PART VIII

Java EE Supporting Technologies

Part VIII explores several technologies that support the Java EE platform. This part contains the following chapters:

- Chapter 43, “Introduction to Java EE Supporting Technologies”
- Chapter 44, “Transactions”
- Chapter 45, “Resources and Resource Adapters”
- Chapter 46, “The Resource Adapter Example”
- Chapter 47, “Java Message Service Concepts”
- Chapter 48, “Java Message Service Examples”
- Chapter 49, “Bean Validation: Advanced Topics”
- Chapter 50, “Using Java EE Interceptors”

◆ ◆ ◆ CHAPTER 43

Introduction to Java EE Supporting Technologies

The Java EE platform includes several technologies and APIs that extend its functionality. These technologies allow applications to access a wide range of services in a uniform manner. These technologies are explained in greater detail in Chapter 44, "Transactions," and Chapter 45, "Resources and Resource Adapters," as well as Chapter 47, "Java Message Service Concepts," Chapter 48, "Java Message Service Examples," and Chapter 49, "Bean Validation: Advanced Topics."

The following topics are addressed here:

- "Transactions in Java EE Applications" on page 789
- "Resources in Java EE Applications" on page 790

Transactions in Java EE Applications

In a Java EE application, a transaction is a series of actions that must all complete successfully, or else all the changes in each action are backed out. Transactions end in either a commit or a rollback.

The Java Transaction API (JTA) allows applications to access transactions in a manner that is independent of specific implementations. JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the transactional application, the Java EE server, and the manager that controls access to the shared resources affected by the transactions.

The JTA defines the `UserTransaction` interface that applications use to start, commit, or roll back transactions. Application components get a `UserTransaction` object through a JNDI lookup by using the name `java:comp/UserTransaction` or by requesting injection of a `UserTransaction` object. An application server uses a number of JTA-defined interfaces to communicate with a transaction manager; a transaction manager uses JTA-defined interfaces to interact with a resource manager.

See Chapter 44, "Transactions," for a more detailed explanation. The JTA 1.1 specification is available at `http://www.oracle.com/technetwork/java/javaee/tech/jta-138684.html`.

Resources in Java EE Applications

A resource is a program object that provides connections to such systems as database servers and messaging systems.

The Java EE Connector Architecture and Resource Adapters

The Java EE Connector architecture enables Java EE components to interact with enterprise information systems (EISs) and EISs to interact with Java EE components. EIS software includes such kinds of systems as enterprise resource planning (ERP), mainframe transaction processing, and nonrelational databases. Connector architecture simplifies the integration of diverse EISs. Each EIS requires only one implementation of the Connector architecture. Because it adheres to the Connector specification, an implementation is portable across all compliant Java EE servers.

The specification defines the contracts for an application server as well as for resource adapters, which are system-level software drivers for specific EIS resources. These standard contracts provide pluggability between application servers and EISs. The Java EE Connector Architecture 1.6 specification defines new system contracts such as Generic Work Context and Security Inflow. The Java EE Connector Architecture 1.6 specification is available at `http://jcp.org/en/jsr/detail?id=322`.

A resource adapter is a Java EE component that implements the Connector architecture for a specific EIS. A resource adapter can choose to support the following levels of transactions:

- `NoTransaction`: No transaction support is provided.
- `LocalTransaction`: Resource manager local transactions are supported.
- `XATransaction`: The resource adapter supports the XA distributed transaction processing model and the JTA `XATransaction` interface.

See Chapter 45, "Resources and Resource Adapters," for a more detailed explanation of resource adapters.

Java Database Connectivity Software

To store, organize, and retrieve data, most applications use relational databases. Java EE applications access relational databases through the JDBC API.

A JDBC resource, or data source, provides applications with a means of connecting to a database. Typically, a JDBC resource is created for each database accessed by the applications deployed in a domain. Transactional access to JDBC resources is available from servlets, JavaServer Faces pages, and enterprise beans. The connection pooling and distributed transaction features are intended for use by JDBC drivers to coordinate with an application server. For more information, see "`DataSource` Objects and Connection Pools" on page 806.

Java Message Service

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

The Java Message Service (JMS) API allows applications to create, send, receive, and read messages. It defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain.

◆ ◆ ◆ CHAPTER 44

Transactions

A typical enterprise application accesses and stores information in one or more databases. Because this information is critical for business operations, it must be accurate, current, and reliable. Data integrity would be lost if multiple programs were allowed to update the same information simultaneously or if a system that failed while processing a business transaction were to leave the affected data only partially updated. By preventing both of these scenarios, software transactions ensure data integrity. Transactions control the concurrent access of data by multiple programs. In the event of a system failure, transactions make sure that after recovery, the data will be in a consistent state.

The following topics are addressed here:

- "What Is a Transaction?" on page 793
- "Container-Managed Transactions" on page 794
- "Bean-Managed Transactions" on page 800
- "Transaction Timeouts" on page 801
- "Updating Multiple Databases" on page 802
- "Transactions in Web Components" on page 803
- "Further Information about Transactions" on page 803

What Is a Transaction?

To emulate a business transaction, a program may need to perform several steps. A financial program, for example, might transfer funds from a checking account to a savings account by using the steps listed in the following pseudocode:

```
begin transaction
    debit checking account
    credit savings account
    update history log
commit transaction
```

Either all or none of the three steps must complete. Otherwise, data integrity is lost. Because the steps within a transaction are a unified whole, a *transaction* is often defined as an indivisible unit of work.

A transaction can end in two ways: with a commit or with a rollback. When a transaction commits, the data modifications made by its statements are saved. If a statement within a transaction fails, the transaction rolls back, undoing the effects of all statements in the transaction. In the pseudocode, for example, if a disk drive were to crash during the `credit` step, the transaction would roll back and undo the data modifications made by the `debit` statement. Although the transaction fails, data integrity would be intact because the accounts still balance.

In the preceding pseudocode, the `begin` and `commit` statements mark the boundaries of the transaction. When designing an enterprise bean, you determine how the boundaries are set by specifying either container-managed or bean-managed transactions.

Container-Managed Transactions

In an enterprise bean with *container-managed transaction demarcation*, the EJB container sets the boundaries of the transactions. You can use container-managed transactions with any type of enterprise bean: session or message-driven. Container-managed transactions simplify development because the enterprise bean code does not explicitly mark the transaction's boundaries. The code does not include statements that begin and end the transaction. By default, if no transaction demarcation is specified, enterprise beans use container-managed transaction demarcation.

Typically, the container begins a transaction immediately before an enterprise bean method starts and commits the transaction just before the method exits. Each method can be associated with a single transaction. Nested or multiple transactions are not allowed within a method.

Container-managed transactions do not require all methods to be associated with transactions. When developing a bean, you can set the transaction attributes to specify which of the bean's methods are associated with transactions.

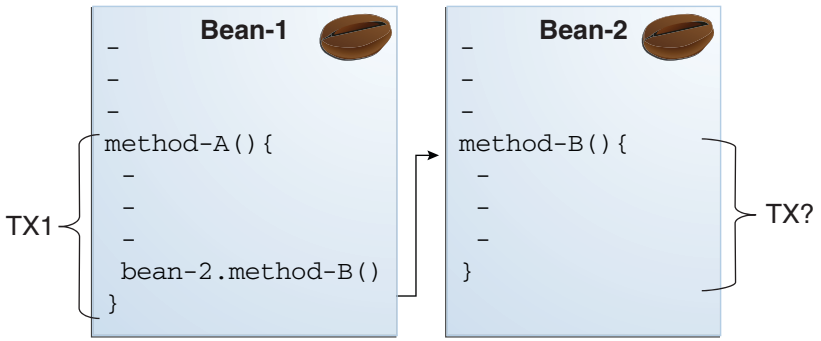
Enterprise beans that use container-managed transaction demarcation must not use any transaction-management methods that interfere with the container's transaction demarcation boundaries. Examples of such methods are the `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection` or the `commit` and `rollback` methods of `javax.jms.Session`. If you require control over the transaction demarcation, you must use application-managed transaction demarcation.

Enterprise beans that use container-managed transaction demarcation also must not use the `javax.transaction.UserTransaction` interface.

Transaction Attributes

A *transaction attribute* controls the scope of a transaction. Figure 44–1 illustrates why controlling the scope is important. In the diagram, `method-A` begins a transaction and then invokes `method-B` of `Bean-2`. When `method-B` executes, does it run within the scope of the transaction started by `method-A`, or does it execute with a new transaction? The answer depends on the transaction attribute of `method-B`.

FIGURE 44–1 Transaction Scope



A transaction attribute can have one of the following values:

- `Required`
- `RequiresNew`
- `Mandatory`
- `NotSupported`
- `Supports`
- `Never`

Required Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.

The `Required` attribute is the implicit transaction attribute for all enterprise bean methods running with container-managed transaction demarcation. You typically do not set the `Required` attribute unless you need to override another transaction attribute. Because transaction attributes are declarative, you can easily change them later.

RequiresNew Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container takes the following steps:

1. Suspends the client's transaction
2. Starts a new transaction
3. Delegates the call to the method
4. Resumes the client's transaction after the method completes

If the client is not associated with a transaction, the container starts a new transaction before running the method.

You should use the `RequiresNew` attribute when you want to ensure that the method always runs within a new transaction.

Mandatory Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container throws a `TransactionRequiredException`.

Use the `Mandatory` attribute if the enterprise bean's method must use the transaction of the client.

NotSupported Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container suspends the client's transaction before invoking the method. After the method has completed, the container resumes the client's transaction.

If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Use the `NotSupported` attribute for methods that don't need transactions. Because transactions involve overhead, this attribute may improve performance.

Supports Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Because the transactional behavior of the method may vary, you should use the `Supports` attribute with caution.

Never Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container throws a `RemoteException`. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Summary of Transaction Attributes

Table 44–1 summarizes the effects of the transaction attributes. Both the `T1` and the `T2` transactions are controlled by the container. A `T1` transaction is associated with the client that calls a method in the enterprise bean. In most cases, the client is another enterprise bean. A `T2` transaction is started by the container just before the method executes.

In the last column of Table 44–1, the word "None" means that the business method does not execute within a transaction controlled by the container. However, the database calls in such a business method might be controlled by the transaction manager of the database management system.

TABLE 44–1 Transaction Attributes and Scope

Transaction Attribute	Client's Transaction	Business Method's Transaction
`Required`	None	T2
`Required`	T1	T1
`RequiresNew`	None	T2
`RequiresNew`	T1	T2
`Mandatory`	None	Error
`Mandatory`	T1	T1
`NotSupported`	None	None
`NotSupported`	T1	None
`Supports`	None	None
`Supports`	T1	T1
`Never`	None	None
`Never`	T1	Error

Setting Transaction Attributes

Transaction attributes are specified by decorating the enterprise bean class or method with a `javax.ejb.TransactionAttribute` annotation and setting it to one of the `javax.ejb.TransactionAttributeType` constants.

If you decorate the enterprise bean class with `@TransactionAttribute`, the specified `TransactionAttributeType` is applied to all the business methods in the class. Decorating a business method with `@TransactionAttribute` applies the `TransactionAttributeType` only to that method. If a `@TransactionAttribute` annotation decorates both the class and the method, the method `TransactionAttributeType` overrides the class `TransactionAttributeType`.

The `TransactionAttributeType` constants shown in Table 44–2 encapsulate the transaction attributes described earlier in this section.

TABLE 44–2 `TransactionAttributeType` Constants

Transaction Attribute	`TransactionAttributeType` Constant
`Required`	`TransactionAttributeType.REQUIRED`
`RequiresNew`	`TransactionAttributeType.REQUIRES_NEW`
`Mandatory`	`TransactionAttributeType.MANDATORY`
`NotSupported`	`TransactionAttributeType.NOT_SUPPORTED`
`Supports`	`TransactionAttributeType.SUPPORTS`
`Never`	`TransactionAttributeType.NEVER`

The following code snippet demonstrates how to use the `@TransactionAttribute` annotation:

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}

    public void thirdMethod() {...}

    public void fourthMethod() {...}
}
```

In this example, the `TransactionBean` class's transaction attribute has been set to `NotSupported`, `firstMethod` has been set to `RequiresNew`, and `secondMethod` has been set to `Required`. Because a `@TransactionAttribute` set on a method overrides the class `@TransactionAttribute`, calls to `firstMethod` will create a new transaction, and calls to `secondMethod` will either run in the current transaction or start a new transaction. Calls to `thirdMethod` or `fourthMethod` do not take place within a transaction.

Rolling Back a Container-Managed Transaction

There are two ways to roll back a container-managed transaction. First, if a system exception is thrown, the container will automatically roll back the transaction. Second, by invoking the `setRollbackOnly` method of the `EJBContext` interface, the bean method instructs the container to roll back the transaction. If the bean throws an application exception, the rollback is not automatic but can be initiated by a call to `setRollbackOnly`.

Synchronizing a Session Bean's Instance Variables

The `SessionSynchronization` interface, which is optional, allows stateful session bean instances to receive transaction synchronization notifications. For example, you could synchronize the instance variables of an enterprise bean with their corresponding values in the database. The container invokes the `SessionSynchronization` methods (`afterBegin`, `beforeCompletion`, and `afterCompletion`) at each of the main stages of a transaction.

The `afterBegin` method informs the instance that a new transaction has begun. The container invokes `afterBegin` immediately before it invokes the business method.

The container invokes the `beforeCompletion` method after the business method has finished but just before the transaction commits. The `beforeCompletion` method is the last opportunity for the session bean to roll back the transaction (by calling `setRollbackOnly`).

The `afterCompletion` method indicates that the transaction has completed. This method has a single `boolean` parameter whose value is `true` if the transaction was committed and `false` if it was rolled back.

Methods Not Allowed in Container-Managed Transactions

You should not invoke any method that might interfere with the transaction boundaries set by the container. The list of prohibited methods follows:

- The `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection`
- The `getUserTransaction` method of `javax.ejb.EJBContext`
- Any method of `javax.transaction.UserTransaction`

You can, however, use these methods to set boundaries in application-managed transactions.

Bean-Managed Transactions

In *bean-managed transaction demarcation*, the code in the session or message-driven bean explicitly marks the boundaries of the transaction. Although beans with container-managed transactions require less coding, they have one limitation: When a method is executing, it can be associated with either a single transaction or no transaction at all. If this limitation will make coding your bean difficult, you should consider using bean-managed transactions.

The following pseudocode illustrates the kind of fine-grained control you can obtain with application-managed transactions. By checking various conditions, the pseudocode decides whether to start or stop certain transactions within the business method:

```
begin transaction
...
    update table-a
...
    if (condition-x)
   commit transaction
    else if (condition-y)
   update table-b
   commit transaction
    else
   rollback transaction
   begin transaction
   update table-c
   commit transaction
```

When coding an application-managed transaction for session or message-driven beans, you must decide whether to use Java Database Connectivity or JTA transactions. The sections that follow discuss both types of transactions.

JTA Transactions

JTA, or the Java Transaction API, allows you to demarcate transactions in a manner that is independent of the transaction manager implementation. GlassFish Server implements the transaction manager with the Java Transaction Service (JTS). However, your code doesn't call the JTS methods directly but instead invokes the JTA methods, which then call the lower-level JTS routines.

A *JTA transaction* is controlled by the Java EE transaction manager. You may want to use a JTA transaction because it can span updates to multiple databases from different vendors. A particular DBMS's transaction manager may not work with heterogeneous databases. However, the Java EE transaction manager does have one limitation: It does not support nested transactions. In other words, it cannot start a transaction for an instance until the preceding transaction has ended.

To demarcate a JTA transaction, you invoke the `begin`, `commit`, and `rollback` methods of the `javax.transaction.UserTransaction` interface.

Returning without Committing

In a stateless session bean with bean-managed transactions, a business method must commit or roll back a transaction before returning. However, a stateful session bean does not have this restriction.

In a stateful session bean with a JTA transaction, the association between the bean instance and the transaction is retained across multiple client calls. Even if each business method called by the client opens and closes the database connection, the association is retained until the instance completes the transaction.

In a stateful session bean with a JDBC transaction, the JDBC connection retains the association between the bean instance and the transaction across multiple calls. If the connection is closed, the association is not retained.

Methods Not Allowed in Bean-Managed Transactions

Do not invoke the `getRollbackOnly` and `setRollbackOnly` methods of the `EJBContext` interface in bean-managed transactions. These methods should be used only in container-managed transactions. For bean-managed transactions, invoke the `getStatus` and `rollback` methods of the `UserTransaction` interface.

Transaction Timeouts

For container-managed transactions, you can use the Administration Console to configure the transaction timeout interval. See "Starting the Administration Console" on page 74.

For enterprise beans with bean-managed JTA transactions, you invoke the `setTransactionTimeout` method of the `UserTransaction` interface.

▼ To Set a Transaction Timeout

1. **In the Administration Console, expand the Configurations node, then expand the server-config node and select Transaction Service.**

2. **On the Transaction Service page, set the value of the Transaction Timeout field to the value of your choice (for example, 5).**

   With this setting, if the transaction has not completed within 5 seconds, the EJB container rolls it back.

   The default value is 0, meaning that the transaction will not time out.

3 **Click Save.**

Updating Multiple Databases

The Java EE transaction manager controls all enterprise bean transactions except for bean-managed JDBC transactions. The Java EE transaction manager allows an enterprise bean to update multiple databases within a transaction. Figure 44–2 and Figure 44–3 show two scenarios for updating multiple databases in a single transaction.

In Figure 44–2, the client invokes a business method in `Bean-A`. The business method begins a transaction, updates Database X, updates Database Y, and invokes a business method in `Bean-B`. The second business method updates Database Z and returns control to the business method in `Bean-A`, which commits the transaction. All three database updates occur in the same transaction.

In Figure 44–3, the client calls a business method in `Bean-A`, which begins a transaction and updates Database X. Then `Bean-A` invokes a method in `Bean-B`, which resides in a remote Java EE server. The method in `Bean-B` updates Database Y. The transaction managers of the Java EE servers ensure that both databases are updated in the same transaction.

FIGURE 44–2 Updating Multiple Databases

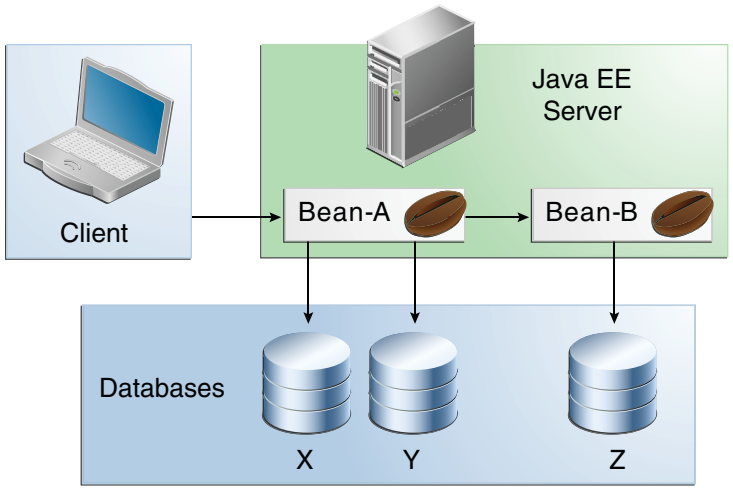
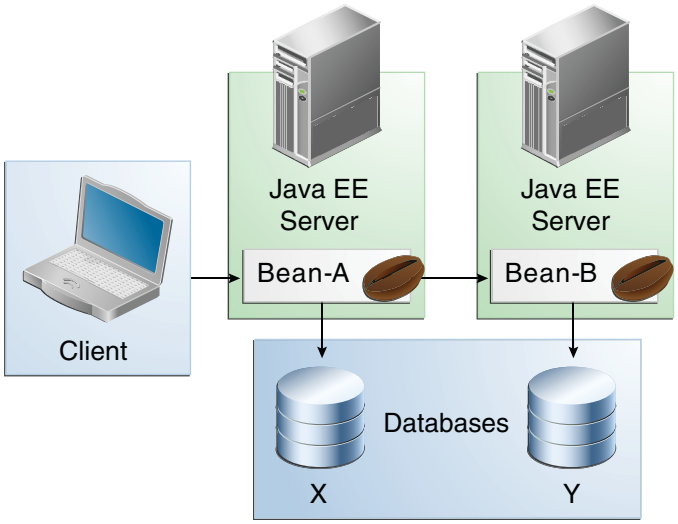


FIGURE 44–3 Updating Multiple Databases across Java EE Servers



Transactions in Web Components

You can demarcate a transaction in a web component by using either the `java.sql.Connection` or the `javax.transaction.UserTransaction` interface. These are the same interfaces that a session bean with bean-managed transactions can use. Transactions demarcated with the `UserTransaction` interface are discussed in "JTA Transactions" on page 800.

Further Information about Transactions

For more information about transactions, see

- Java Transaction API 1.1 specification:

  `http://www.oracle.com/technetwork/java/javaee/tech/jta-138684.html`

◆ ◆ ◆ CHAPTER 45

Resources and Resource Adapters

Java EE components can access a wide variety of resources, including databases, mail sessions, Java Message Service objects, and URLs. The Java EE 6 platform provides mechanisms that allow you to access all these resources in a similar manner. This chapter explains how to get connections to several types of resources.

The following topics are addressed here:

- "Resources and JNDI Naming" on page 805
- "`DataSource` Objects and Connection Pools" on page 806
- "Resource Injection" on page 807
- "Resource Adapters and Contracts" on page 810
- "Metadata Annotations" on page 814
- "Common Client Interface" on page 815
- "Using Resource Adapters With Contexts and Dependency Injection for the Java EE Platform (CDI)" on page 816
- "Further Information about Resources" on page 817

Resources and JNDI Naming

In a distributed application, components need to access other components and resources, such as databases. For example, a servlet might invoke remote methods on an enterprise bean that retrieves information from a database. In the Java EE platform, the Java Naming and Directory Interface (JNDI) naming service enables components to locate other components and resources.

A *resource* is a program object that provides connections to systems, such as database servers and messaging systems. (A Java Database Connectivity resource is sometimes referred to as a data source.) Each resource object is identified by a unique, people-friendly name, called the JNDI name. For example, the JNDI name of the JDBC resource for the Java DB database that is shipped with the GlassFish Server is `jdbc/__default`.

An administrator creates resources in a JNDI namespace. In the GlassFish Server, you can use either the Administration Console or the `asadmin` command to create resources. Applications then use annotations to inject the resources. If an application uses resource injection, the GlassFish Server invokes the JNDI API, and the application is not required to do so. However, it is also possible for an application to locate resources by making direct calls to the JNDI API.

A resource object and its JNDI name are bound together by the naming and directory service. To create a new resource, a new name/object binding is entered into the JNDI namespace. You inject resources by using the `@Resource` annotation in an application.

You can use a deployment descriptor to override the resource mapping that you specify in an annotation. Using a deployment descriptor allows you to change an application by repackaging it rather than by both recompiling the source files and repackaging. However, for most applications, a deployment descriptor is not necessary.

`DataSource` Objects and Connection Pools

To store, organize, and retrieve data, most applications use a relational database. Java EE 6 components may access relational databases through the JDBC API. For information on this API, see `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html`.

In the JDBC API, databases are accessed by using `DataSource` objects. A `DataSource` has a set of properties that identify and describe the real-world data source that it represents. These properties include such information as the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on. In the GlassFish Server, a data source is called a JDBC resource.

Applications access a data source by using a connection, and a `DataSource` object can be thought of as a factory for connections to the particular data source that the `DataSource` instance represents. In a basic `DataSource` implementation, a call to the `getConnection` method returns a connection object that is a physical connection to the data source.

A `DataSource` object may be registered with a JNDI naming service. If so, an application can use the JNDI API to access that `DataSource` object, which can then be used to connect to the data source it represents.

`DataSource` objects that implement connection pooling also produce a connection to the particular data source that the `DataSource` class represents. The connection object that the `getConnection` method returns is a handle to a `PooledConnection` object rather than being a physical connection. An application uses the connection object in the same way that it uses a connection. Connection pooling has no effect on application code except that a pooled connection, like all connections, should always be explicitly closed. When an application closes a connection that is pooled, the connection is returned to a pool of reusable connections. The next time `getConnection` is called, a handle to one of these pooled connections will be returned

if one is available. Because connection pooling avoids creating a new physical connection every time one is requested, applications can run significantly faster.

A JDBC connection pool is a group of reusable connections for a particular database. Because creating each new physical connection is time consuming, the server maintains a pool of available connections to increase performance. When it requests a connection, an application obtains one from the pool. When an application closes a connection, the connection is returned to the pool.

Applications that use the Persistence API specify the `DataSource` object they are using in the `jta-data-source` element of the `persistence.xml` file:

```
<jta-data-source>jdbc/MyOrderDB</jta-data-source>
```

This is typically the only reference to a JDBC object for a persistence unit. The application code does not refer to any JDBC objects.

Resource Injection

The `javax.annotation.Resource` annotation is used to declare a reference to a resource; `@Resource` can decorate a class, a field, or a method. The container will inject the resource referred to by `@Resource` into the component either at runtime or when the component is initialized, depending on whether field/method injection or class injection is used. With field-based and method-based injection, the container will inject the resource when the application is initialized. For class-based injection, the resource is looked up by the application at runtime.

The `@Resource` annotation has the following elements:

- `name`: The JNDI name of the resource
- `type`: The Java language type of the resource
- `authenticationType`: The authentication type to use for the resource
- `shareable`: Indicates whether the resource can be shared
- `mappedName`: A nonportable, implementation-specific name to which the resource should be mapped
- `description`: The description of the resource

The `name` element is the JNDI name of the resource and is optional for field-based and method-based injection. For field-based injection, the default `name` is the field name qualified by the class name. For method-based injection, the default `name` is the JavaBeans property name, based on the method qualified by the class name. The `name` element must be specified for class-based injection.

The type of resource is determined by one of the following:

- The type of the field the `@Resource` annotation is decorating for field-based injection
- The type of the JavaBeans property the `@Resource` annotation is decorating for method-based injection
- The `type` element of `@Resource`

For class-based injection, the `type` element is required.

The `authenticationType` element is used only for connection factory resources, such as the resources of a connector, also called the resource adapter, or data source. This element can be set to one of the `javax.annotation.Resource.AuthenticationType` enumerated type values: `CONTAINER`, the default, and `APPLICATION`.

The `shareable` element is used only for Object Resource Broker (ORB) instance resources or connection factory resource. This element indicates whether the resource can be shared between this component and other components and may be set to `true`, the default, or `false`.

The `mappedName` element is a nonportable, implementation-specific name to which the resource should be mapped. Because the `name` element, when specified or defaulted, is local only to the application, many Java EE servers provide a way of referring to resources across the application server. This is done by setting the `mappedName` element. Use of the `mappedName` element is nonportable across Java EE server implementations.

The `description` element is the description of the resource, typically in the default language of the system on which the application is deployed. This element is used to help identify resources and to help application developers choose the correct resource.

Field-Based Injection

To use field-based resource injection, declare a field and decorate it with the `@Resource` annotation. The container will infer the name and type of the resource if the `name` and `type` elements are not specified. If you do specify the `type` element, it must match the field's `type` declaration.

In the following code, the container infers the `name` of the resource, based on the class name and the field name: `com.example.SomeClass/myDB`. The inferred `type` is `javax.sql.DataSource.class`:

```
package com.example;

public class SomeClass {
    @Resource
    private javax.sql.DataSource myDB;
...
}
```

In the following code, the JNDI name is `customerDB`, and the inferred `type` is `javax.sql.DataSource.class`:

```
package com.example;

public class SomeClass {
    @Resource(name="customerDB")
    private javax.sql.DataSource myDB;
...
}
```

Method-Based Injection

To use method-based injection, declare a setter method and decorate it with the `@Resource` annotation. The container will infer the name and type of the resource if the `name` and `type` elements are not specified. The setter method must follow the JavaBeans conventions for property names: The method name must begin with `set`, have a `void` return type, and only one parameter. If you do specify the `type` element, it must match the field's type declaration.

In the following code, the container infers the `name` of the resource based on the class name and the field name: `com.example.SomeClass/myDB`. The inferred `type` is `javax.sql.DataSource.class`:

```
package com.example;

public class SomeClass {

    private javax.sql.DataSource myDB;
...
    @Resource
    private void setMyDB(javax.sql.DataSource ds) {
        myDB = ds;
    }
...
}
```

In the following code, the JNDI name is `customerDB`, and the inferred `type` is `javax.sql.DataSource.class`:

```
package com.example;

public class SomeClass {

    private javax.sql.DataSource myDB;
...
    @Resource(name="customerDB")
    private void setMyDB(javax.sql.DataSource ds) {
        myDB = ds;
    }
...
}
```

Class-Based Injection

To use class-based injection, decorate the class with a `@Resource` annotation, and set the required `name` and `type` elements:

```
@Resource(name="myMessageQueue",
                type="javax.jms.ConnectionFactory")
public class SomeMessageBean {
...
}
```

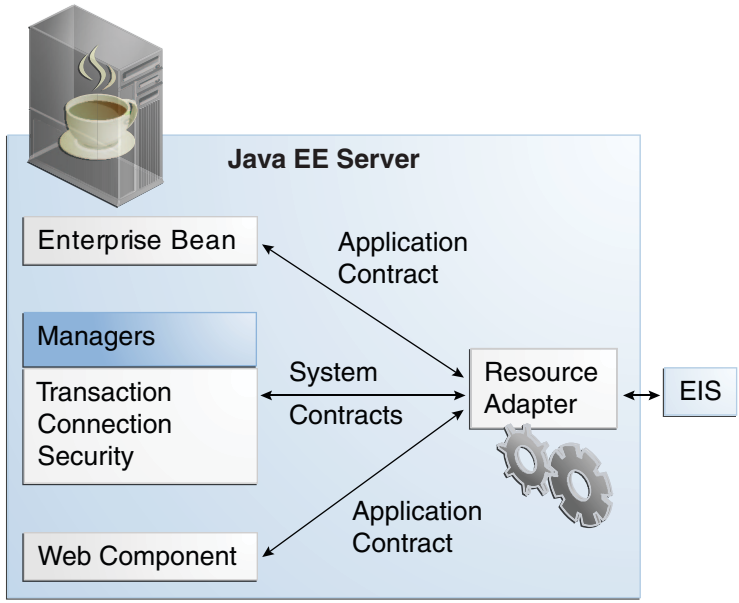
The `@Resources` annotation is used to group together multiple `@Resource` declarations for class-based injection. The following code shows the `@Resources` annotation containing two `@Resource` declarations. One is a Java Message Service message queue, and the other is a JavaMail session:

```
@Resources({
    @Resource(name="myMessageQueue",
                    type="javax.jms.ConnectionFactory"),
    @Resource(name="myMailSession",
                    type="javax.mail.Session")
})
public class SomeMessageBean {
...
}
```

Resource Adapters and Contracts

A resource adapter is a Java EE component that implements the Java EE Connector architecture for a specific EIS. Examples of EISs include enterprise resource planning, mainframe transaction processing, and database systems. In a Java EE server, the Java Message Server and JavaMail also act as EISs that you access using resource adapters. As illustrated in Figure 45–1, the resource adapter facilitates communication between a Java EE application and an EIS.

FIGURE 45–1 Resource Adapters



Stored in a Resource Adapter Archive (RAR) file, a resource adapter can be deployed on any Java EE server, much like a Java EE application. A RAR file may be contained in an Enterprise Archive (EAR) file, or it may exist as a separate file.

A resource adapter is analogous to a JDBC driver. Both provide a standard API through which an application can access a resource that is outside the Java EE server. For a resource adapter, the target system is an EIS; for a JDBC driver, it is a DBMS. Resource adapters and JDBC drivers are rarely created by application developers. In most cases, both types of software are built by vendors that sell tools, servers, or integration software.

The resource adapter mediates communication between the Java EE server and the EIS by means of contracts. The application contract defines the API through which a Java EE component, such as an enterprise bean, accesses the EIS. This API is the only view that the component has of the EIS. The system contracts link the resource adapter to important services that are managed by the Java EE server. The resource adapter itself and its system contracts are transparent to the Java EE component.

Management Contracts

The Java EE Connector Architecture defines system contracts that enable resource adapter lifecycle and thread management.

Lifecycle Management

The Connector Architecture specifies a lifecycle management contract that allows an application server to manage the lifecycle of a resource adapter. This contract provides a mechanism for the application server to bootstrap a resource adapter instance during the deployment or application server startup. This contract also provides a means for the application server to notify the resource adapter instance when it is undeployed or when an orderly shutdown of the application server takes place.

Work Management Contract

The Connector Architecture work management contract ensures that resource adapters use threads in the proper, recommended manner. This contract also enables an application server to manage threads for resource adapters.

Resource adapters that improperly use threads can jeopardize the entire application server environment. For example, a resource adapter might create too many threads or might not properly release threads it has created. Poor thread handling inhibits application server shutdown and impacts the application server's performance because creating and destroying threads are expensive operations.

The work management contract establishes a means for the application server to pool and reuse threads, similar to pooling and reusing connections. By adhering to this contract, the resource adapter does not have to manage threads itself. Instead, the resource adapter has the application server create and provide needed threads. When it is finished with a given thread, the resource adapter returns the thread to the application server. The application server manages the thread, either returning it to a pool for later reuse or destroying it. Handling threads in this manner results in increased application server performance and more efficient use of resources.

In addition to moving thread management to the application server, the Connector Architecture provides a flexible model for a resource adapter that uses threads.

- The requesting thread can choose to block (stop its own execution) until the work thread completes.
- The requesting thread can block while it waits to get the work thread. When the application server provides a work thread, the requesting thread and the work thread execute in parallel.
- The resource adapter can opt to submit the work for the thread to a queue. The thread executes the work from the queue at some later point. The resource adapter continues its own execution from the point it submitted the work to the queue, no matter when the thread executes it.

With the latter two approaches, the submitting thread and the work thread may execute simultaneously or independently. For these approaches, the contract specifies a listener mechanism to notify the resource adapter that the thread has completed its operation. The resource adapter can also specify the execution context for the thread, and the work management contract controls the context in which the thread executes.

Generic Work Context Contract

The work management contract between the application server and a resource adapter enables a resource adapter to do a task, such as communicating with the EIS or delivering messages, by delivering `Work` instances for execution.

A generic work context contract enables a resource adapter to control the contexts in which the `Work` instances that it submits are executed by the application server's `WorkManager`. A generic work context mechanism also enables an application server to support new message inflow and delivery schemes. It also provides a richer contextual `Work` execution environment to the resource adapter while still maintaining control over concurrent behavior in a managed environment.

The generic work context contract standardizes the transaction context and the security context.

Outbound and Inbound Contracts

The Connector Architecture defines the following outbound contracts, system-level contracts between an application server and an EIS that enable outbound connectivity to an EIS.

- The connection management contract supports connection pooling, a technique that enhances application performance and scalability. Connection pooling is transparent to the application, which simply obtains a connection to the EIS.
- The transaction management contract extends the connection management contract and provides support for management of both local and XA transactions.

  A local transaction is limited in scope to a single EIS system, and the EIS resource manager itself manages such transaction. An XA transaction or global transaction can span multiple resource managers. This form of transaction requires transaction coordination by an external transaction manager, typically bundled with an application server. A transaction manager uses a two-phase commit protocol to manage a transaction that spans multiple resource managers or EISs, and uses one-phase commit optimization if only one resource manager is participating in an XA transaction.
- The security management contract provides mechanisms for authentication, authorization, and secure communication between a Java EE server and an EIS to protect the information in the EIS.

  A work security map matches EIS identities to the application server domain's identities.

Inbound contracts are system contracts between a Java EE server and an EIS that enable inbound connectivity from the EIS: pluggability contracts for message providers and contracts for importing transactions.

Metadata Annotations

Java EE Connector Architecture 1.6 introduces a set of annotations to minimize the need for deployment descriptors.

- The `@Connector` annotation can be used by the resource adapter developer to specify that the JavaBeans component is a resource adapter JavaBeans component. This annotation is used for providing metadata about the capabilities of the resource adapter. Optionally, you can provide a JavaBeans component implementing the `ResourceAdapter` interface, as in the following example:

```
@Connector(
   description = "Sample adapter using the JavaMail API",
   displayName = "InboundResourceAdapter",
   vendorName = "My Company, Inc.",
   eisType = "MAIL",
   version = "1.0"
)
public class ResourceAdapterImpl
        implements ResourceAdapter, java.io.Serializable {
    ...
    ...
}
```

- The `@ConnectionDefinition` annotation defines a set of connection interfaces and classes pertaining to a particular connection type, as in the following example:

```
@ConnectionDefinition(
    connectionFactory = JavaMailConnectionFactory.class,
    connectionFactoryImpl = JavaMailConnectionFactoryImpl.class,
    connection = JavaMailConnection.class,
    connectionImpl = JavaMailConnectionImpl.class
)
public class ManagedConnectionFactoryImpl implements
        ManagedConnectionFactory, Serializable {
    ...
}
```

- The `@AdministeredObject` annotation designates a JavaBeans component as an administered object.
- The `@Activation` annotation contains configuration information pertaining to inbound connectivity from an EIS instance, as in the following example:

```
@Activation(
        messageListeners = {JavaMailMessageListener.class}
)
public class ActivationSpecImpl
        implements ActivationSpec, Serializable {
    ...
    @ConfigProperty()
    // serverName property value
    private String serverName = "";

    @ConfigProperty()
    // userName property value
    private String userName = "";
```

```
    @ConfigProperty()
    // password property value
    private String password = "";

    @ConfigProperty()
    // folderName property value
    private String folderName = "INBOX";

    // protocol property value
    @ConfigProperty(
            description = "Normally imap or pop3"
    )
    private String protocol = "imap";
    ...
}
```

- The `@ConfigProperty` annotation can be used on JavaBeans components to provide additional configuration information that may be used by the deployer and resource adapter provider. The preceding example code shows several `@ConfigProperty` annotations.

The specification allows a resource adapter to be developed in mixed-mode form, that is the ability for a resource adapter developer to use both metadata annotations and deployment descriptors in applications. An application assembler or deployer may use the deployment descriptor to override the metadata annotations specified by the resource adapter developer.

The deployment descriptor for a resource adapter, if present, is named `ra.xml`. The `metadata-complete` attribute defines whether the deployment descriptor for the resource adapter module is complete or whether the class files available to the module and packaged with the resource adapter need to be examined for annotations that specify deployment information.

For the complete list of annotations and JavaBeans components introduced in the Java EE 6 platform, see the Java EE Connector architecture 1.6 specification.

Common Client Interface

This section explains how components use the Connector Architecture Common Client Interface (CCI) API and a resource adapter to access data from an EIS. The CCI API defines a set of interfaces and classes whose methods allow a client to perform typical data access operations. The CCI interfaces and classes are as follows:

- `ConnectionFactory`: Provides an application component with a `Connection` instance to an EIS.
- `Connection`: Represents the connection to the underlying EIS.
- `ConnectionSpec`: Provides a means for an application component to pass connection-request-specific properties to the `ConnectionFactory` when making a connection request.
- `Interaction`: Provides a means for an application component to execute EIS functions, such as database stored procedures.

- `InteractionSpec`: Holds properties pertaining to an application component's interaction with an EIS.
- `Record`: The superinterface for the various kinds of record instances. Record instances can be `MappedRecord`, `IndexedRecord`, or `ResultSet` instances, all of which inherit from the `Record` interface.
- `RecordFactory`: Provides an application component with a `Record` instance.
- `IndexedRecord`: Represents an ordered collection of `Record` instances based on the `java.util.List` interface.

A client or application component that uses the CCI to interact with an underlying EIS does so in a prescribed manner. The component must establish a connection to the EIS's resource manager, and it does so using the `ConnectionFactory`. The `Connection` object represents the connection to the EIS and is used for subsequent interactions with the EIS.

The component performs its interactions with the EIS, such as accessing data from a specific table, using an `Interaction` object. The application component defines the Interaction object by using an `InteractionSpec` object. When it reads data from the EIS, such as from database tables, or writes to those tables, the application component does so by using a particular type of `Record` instance: a `MappedRecord`, an `IndexedRecord`, or a `ResultSet` instance.

Note, too, that a client application that relies on a CCI resource adapter is very much like any other Java EE client that uses enterprise bean methods.

Using Resource Adapters With Contexts and Dependency Injection for the Java EE Platform (CDI)

To enable a resource adapter for CDI, provide a `beans.xml` file in the `META-INF` directory of the packaged archive of the resource adapter. For more information about `beans.xml`, see "Configuring a CDI Application" on page 525.

All classes in the resource adapter are available for injection. All classes in the resource adapter can be CDI managed beans except for the following classes:

- **Resource adapter beans**: These beans are classes that are annotated with the `javax.resource.spi.Connector` annotation or are declared as corresponding elements in the resource adapter deployment descriptor, `ra.xml`.
- **Managed connection factory beans**: These beans are classes that are annotated with the `javax.resource.spi.ConnectorDefinition` annotation or the `javax.resource.spi.ConnectorDefinitions` annotation or are declared as corresponding elements in `ra.xml`.
- **Activation specification beans**: These beans are classes that are annotated with the `javax.resource.spi.Activation` annotation or are declared as corresponding elements in `ra.xml`.

- **Administered object beans**: These beans are classes that are annotated with the `javax.resource.spi.AdministeredObject` annotation or are declared as corresponding elements in `ra.xml`.

Further Information about Resources

For more information about resources and annotations, see

- Java EE 6 Platform Specification (JSR 316):

  `http://jcp.org/en/jsr/detail?id=316`
- Java EE Connector architecture 1.6 specification:

  `http://jcp.org/en/jsr/detail?id=322`
- EJB 3.1 specification:

  `http://jcp.org/en/jsr/detail?id=318`
- Common Annotations for the Java Platform:

  `http://www.jcp.org/en/jsr/detail?id=250`

◆ ◆ ◆ CHAPTER 46

The Resource Adapter Example

The `mailconnector` example shows how you can use a resource adapter, a message-driven bean (MDB), and JavaServer Faces technology to create an application that can send email messages and browse for messages. This example uses a sample implementation of the JavaMail API called `mock-javamail`. The resource adapter is deployed separately, while the MDB and the web application are packaged in an EAR file.

The following topics are addressed here:

- "The Resource Adapter" on page 819
- "The Message-Driven Bean" on page 820
- "The Web Application" on page 820
- "Running the `mailconnector` Example" on page 820

The Resource Adapter

The `mailconnector` resource adapter enables the MDB to receive email messages that are delivered to a specific mailbox folder on a mail server. It also provides connection factory objects clients can use to obtain connection objects that allow them to synchronously query email servers for new messages in a specific mailbox folder.

In this example, the MDB activates the resource adapter, but it does not receive email messages. Instead, this example allows users to synchronously query an email server for new messages.

The components of the resource adapter are as follows:

- `mailconnector.ra`: Base class of the `mailconnector` resource adapter
- `mailconnector.ra/inbound`: Classes that implement the inbound resource adapter, which supports delivery of JavaMail messages to MDBs
- `mailconnector.ra/outbound`: Classes that implement the outbound resource adapter, which supports synchronous queries to email servers

- `mailconnector.api`: Interfaces that are implemented by MDBs associated with this resource adapter and by the `Connection` and `ConnectionFactoryinterfaces` provided by the outbound resource adapter
- `mailconnector.share`: JavaBeans class that implements the `ConnectionSpec` interface, allowing properties to be passed to the outbound resource adapter

When the resource adapter is deployed, it uses the Work Management facilities available to resource adapters to start a thread that monitors mailbox folders for new messages. The polling thread of the resource adapter monitors the mailbox folders every 30 seconds for new messages.

The Message-Driven Bean

The `mailconnector` message-driven bean, `JavaMailMessageBean`, activates the resource adapter. When an MDB is deployed, the application server passes the MDB's activation config properties (commented out in this case) to the `mailconnector` resource adapter, which forwards it to the polling thread. When the MDB is undeployed, the application server notifies the resource adapter, which notifies the polling thread to stop monitoring the mail folder associated with the MDB being undeployed.

The MDB is packaged in an EJB JAR file.

The Web Application

The web application in the `mailconnector` example contains an HTML page (`index.html`), Facelets pages, and managed beans that let you log in, send email messages to a mailbox folder, and query for new messages in a mail folder using the connection interfaces provided by the `mailconnector` resource adapter.

The application protects the Facelets pages by using form-based authentication, specified through a security constraint in the `web.xml` file.

The web application is packaged in a WAR file.

Running the `mailconnector` Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `mailconnector` example.

▼ Before You Deploy the `mailconnector` Example

Before you deploy the `mailconnector` application, perform the following steps.

1. Download **mock-javamail-1.9.jar** from **http://download.java.net/maven/2/org/jvnet/mock-javamail/mock-javamail/1.9/.**
2. Copy this JAR file to the directory *as-install*/**lib**.
3. Restart GlassFish Server.
4. Open the GlassFish Server Administration Console in a web browser at **http://localhost:4848**.
5. In the Administration Console, expand the Configurations node, then expand the **server-config** node.
6. Select the Security node.
7. Select the Default Principal to Role Mapping Enabled check box.
8. Click Save.

▼ To Build, Package, and Deploy the mailconnector Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   tut-install/examples/connectors/mailconnector/
3. Select the **mailconnector-ra** folder and click Open Project.
4. In the Projects tab, right-click the **mailconnector-ra** project and select Build.
   This command builds the resource adapter. It also places identical files named mailconnector.rar and mailconnector.jar in the mailconnector directory.
5. In the Projects tab, right-click the **mailconnector-ra** project and select Deploy.
6. From the File menu, choose Open Project.
7. In the Open Project dialog, navigate to:
   tut-install/examples/connectors/mailconnector/
8. Select the **mailconnector-ear** folder.

9. **Select the Open Required Projects check box and click Open Project.**
10. **In the Projects tab, right-click the `mailconnector-ear` project and select Build.**
11. **In a terminal window, navigate to:**

    tut-install`/examples/connectors/mailconnector/mailconnector-ear/`
12. **Enter the following command to create the resources and users:**

    ```
    ant setup
    ```
13. **In NetBeans IDE, in the Projects tab, right-click the `mailconnector-ear` project and select Deploy.**

▼ To Build, Package, and Deploy the `mailconnector` Example Using Ant

1. **In a terminal window, go to:**

   tut-install`/examples/connectors/mailconnector/mailconnector-ear/`
2. **Enter the following command:**

   ```
   ant all
   ```

   This command builds and deploys the `mailconnector-ra` RAR file, sets up users and resources, then builds and deploys the `mailconnector-ear` EAR file. It also places identical files named `mailconnector.rar` and `mailconnector.jar` in the `mailconnector` directory.

▼ To Run the `mailconnector` Example

1. **In a web browser, navigate to the following URL:**

   `http://localhost:8080/mailconnector-war/`
2. **Log in with a user name of either `user1`, `user2`, `user3`, or `user4`. The password is the same as the user name.**

   You can send messages and browse for the messages you sent. The messages you sent are available 30 seconds after you sent them.

   For example, you can log in as `user1` and send a message to `user4`, then log in as `user4` and query for messages. In the form for browsing messages, verify that the fields are correct, then click Browse.

   View the server log to follow the flow of the application. Most classes and methods specify logging information that makes the sequence of events easy to follow.

3. **Before you undeploy the application, in a terminal window, navigate to *tut-install*/examples/connectors/mailconnector/mailconnector-ear/ and enter the following command to remove the resources and users:**

   ```
   ant takedown
   ```

   You cannot undeploy the resource adapter until you run this command.

Next Steps When you clean the application, you can also remove the `mailconnector.rar` and `mailconnector.jar` files from the `mailconnector` directory.

Remove the `mock-javamail-1.9.jar` file from the *as-install*`/lib` directory if you might run any other applications that use the JavaMail API (for example, "The async Example Application" on page 508).

◆ ◆ ◆ CHAPTER 47

Java Message Service Concepts

This chapter provides an introduction to the Java Message Service (JMS) API, a Java API that allows applications to create, send, receive, and read messages using reliable, asynchronous, loosely coupled communication. It covers the following topics:

- "Overview of the JMS API" on page 825
- "Basic JMS API Concepts" on page 828
- "The JMS API Programming Model" on page 832
- "Creating Robust JMS Applications" on page 842
- "Using the JMS API in Java EE Applications" on page 851
- "Further Information about JMS" on page 858

Overview of the JMS API

This overview defines the concept of messaging, describes the JMS API and when it can be used, and explains how the JMS API works within the Java EE platform.

What Is Messaging?

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is *loosely coupled*. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only which message format and which destination to use. In this respect, messaging differs from

tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.

Messaging also differs from electronic mail (email), which is a method of communication between people or between software applications and people. Messaging is used for communication between software applications or software components.

What Is the JMS API?

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain.

The JMS API enables communication that is not only loosely coupled but also:

- **Asynchronous**: A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
- **Reliable**: The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

The current version of the JMS specification is Version 1.1. You can download a copy of the specification from the JMS web site: `http://www.oracle.com/technetwork/java/index-jsp-142945.html`.

When Can You Use the JMS API?

An enterprise application provider is likely to choose a messaging API over a tightly coupled API, such as a remote procedure call (RPC), under the following circumstances.

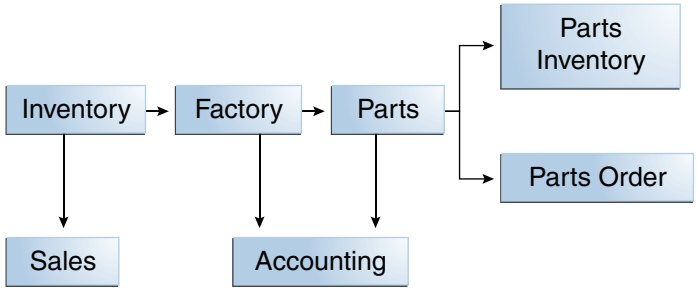
- The provider wants the components not to depend on information about other components' interfaces, so components can be easily replaced.
- The provider wants the application to run whether or not all components are up and running simultaneously.
- The application business model allows a component to send information to another and to continue to operate without receiving an immediate response.

For example, components of an enterprise application for an automobile manufacturer can use the JMS API in situations like these:

- The inventory component can send a message to the factory component when the inventory level for a product goes below a certain level so the factory can make more cars.
- The factory component can send a message to the parts components so the factory can assemble the parts it needs.
- The parts components in turn can send messages to their own inventory and order components to update their inventories and to order new parts from suppliers.
- Both the factory and the parts components can send messages to the accounting component to update budget numbers.
- The business can publish updated catalog items to its sales force.

Using messaging for these tasks allows the various components to interact with one another efficiently, without tying up network or other resources. Figure 47–1 illustrates how this simple example might work.

FIGURE 47–1 Messaging in an Enterprise Application



Manufacturing is only one example of how an enterprise can use the JMS API. Retail applications, financial services applications, health services applications, and many others can make use of messaging.

How Does the JMS API Work with the Java EE Platform?

When the JMS API was introduced in 1998, its most important purpose was to allow Java applications to access existing messaging-oriented middleware (MOM) systems, such as MQSeries from IBM. Since that time, many vendors have adopted and implemented the JMS API, so a JMS product can now provide a complete messaging capability for an enterprise.

Beginning with the 1.3 release of the Java EE platform, the JMS API has been an integral part of the platform, and application developers have been able to use messaging with Java EE components.

The JMS API in the Java EE platform has the following features.

- Application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message. Application clients can in addition receive JMS messages asynchronously. (Applets, however, are not required to support the JMS API.)
- Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages. A JMS provider can optionally implement concurrent processing of messages by message-driven beans.
- Message send and receive operations can participate in distributed transactions, which allow JMS operations and database accesses to take place within a single transaction.

The JMS API enhances the Java EE platform by simplifying enterprise development, allowing loosely coupled, reliable, asynchronous interactions among Java EE components and legacy systems capable of messaging. A developer can easily add new behavior to a Java EE application that has existing business events by adding a new message-driven bean to operate on specific business events. The Java EE platform, moreover, enhances the JMS API by providing support for distributed transactions and allowing for the concurrent consumption of messages. For more information, see the Enterprise JavaBeans specification, v3.1.

The JMS provider can be integrated with the application server using the Java EE Connector architecture. You access the JMS provider through a resource adapter. This capability allows vendors to create JMS providers that can be plugged in to multiple application servers, and it allows application servers to support multiple JMS providers. For more information, see the Java EE Connector architecture specification, v1.6.

Basic JMS API Concepts

This section introduces the most basic JMS API concepts, the ones you must know to get started writing simple application clients that use the JMS API.

The next section introduces the JMS API programming model. Later sections cover more advanced concepts, including the ones you need in order to write applications that use message-driven beans.

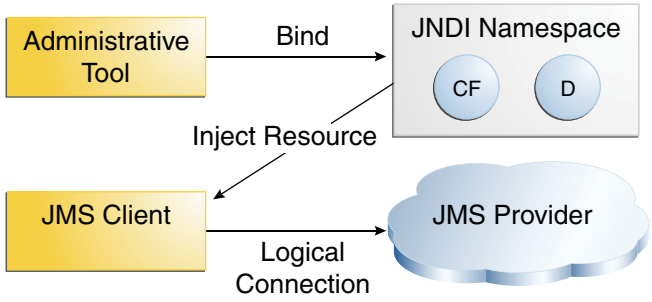
JMS API Architecture

A JMS application is composed of the following parts.

- A *JMS provider* is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the Java EE platform includes a JMS provider.
- *JMS clients* are the programs or components, written in the Java programming language, that produce and consume messages. Any Java EE application component can act as a JMS client.
- *Messages* are the objects that communicate information between JMS clients.
- *Administered objects* are preconfigured JMS objects created by an administrator for the use of clients. The two kinds of JMS administered objects are destinations and connection factories, described in "JMS Administered Objects" on page 833.

Figure 47–2 illustrates the way these parts interact. Administrative tools allow you to bind destinations and connection factories into a JNDI namespace. A JMS client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.

FIGURE 47–2 JMS API Architecture



Messaging Domains

Before the JMS API existed, most messaging products supported either the point-to-point or the publish/subscribe approach to messaging. The JMS specification provides a separate domain for each approach and defines compliance for each domain. A stand-alone JMS provider can implement one or both domains. A Java EE provider must implement both domains.

In fact, most implementations of the JMS API support both the point-to-point and the publish/subscribe domains, and some JMS clients combine the use of both domains in a single application. In this way, the JMS API has extended the power and flexibility of messaging products.

The JMS specification goes one step further: It provides common interfaces that enable you to use the JMS API in a way that is not specific to either domain. The following subsections describe the two messaging domains and the use of the common interfaces.

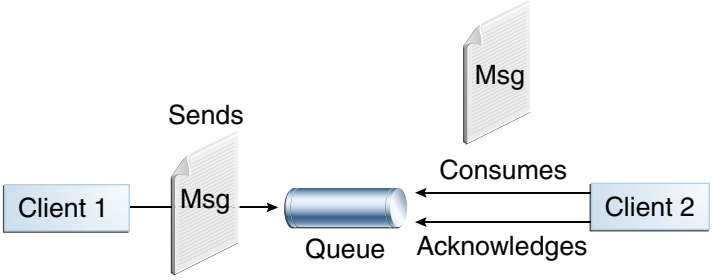
Point-to-Point Messaging Domain

A *point-to-point* (PTP) product or application is built on the concept of message *queues*, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages sent to them until the messages are consumed or expire.

PTP messaging, illustrated in Figure 47–3, has the following characteristics:

- Each message has only one consumer.
- A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
- The receiver acknowledges the successful processing of a message.

FIGURE 47–3 Point-to-Point Messaging



Use PTP messaging when every message you send must be processed successfully by one consumer.

Publish/Subscribe Messaging Domain

In a *publish/subscribe* (pub/sub) product or application, clients address messages to a *topic*, which functions somewhat like a bulletin board. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system

takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.

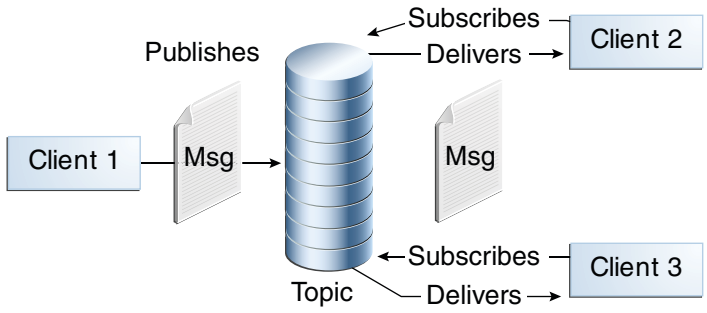
Pub/sub messaging has the following characteristics.

- Each message can have multiple consumers.
- Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

The JMS API relaxes this timing dependency to some extent by allowing subscribers to create *durable subscriptions*, which receive messages sent while the subscribers are not active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients. For more information about durable subscriptions, see "Creating Durable Subscriptions" on page 847.

Use pub/sub messaging when each message can be processed by any number of consumers (or none). Figure 47–4 illustrates pub/sub messaging.

FIGURE 47–4 Publish/Subscribe Messaging



Programming with the Common Interfaces

Version 1.1 of the JMS API allows you to use the same code to send and receive messages under either the PTP or the pub/sub domain. The destinations you use remain domain-specific, and the behavior of the application will depend in part on whether you are using a queue or a topic. However, the code itself can be common to both domains, making your applications flexible and reusable. This tutorial describes and illustrates these common interfaces.

Message Consumption

Messaging products are inherently asynchronous: There is no fundamental timing dependency between the production and the consumption of a message. However, the JMS specification uses this term in a more precise sense. Messages can be consumed in either of two ways:

- **Synchronously**: A subscriber or a receiver explicitly fetches the message from the destination by calling the `receive` method. The `receive` method can block until a message arrives or can time out if a message does not arrive within a specified time limit.
- **Asynchronously**: A client can register a *message listener* with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage` method, which acts on the contents of the message.

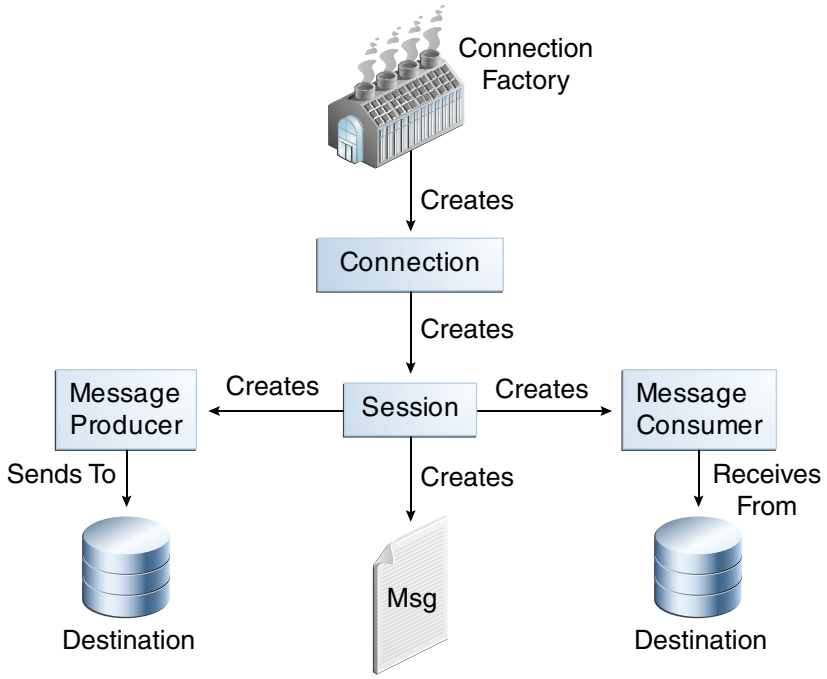
The JMS API Programming Model

The basic building blocks of a JMS application are:

- Administered objects: connection factories and destinations
- Connections
- Sessions
- Message producers
- Message consumers
- Messages

Figure 47–5 shows how all these objects fit together in a JMS client application.

FIGURE 47–5 The JMS API Programming Model



This section describes all these objects briefly and provides sample commands and code snippets that show how to create and use the objects. The last subsection briefly describes JMS API exception handling.

Examples that show how to combine all these objects in applications appear in later sections. For more detail, see the JMS API documentation, part of the Java EE API documentation.

JMS Administered Objects

Two parts of a JMS application, destinations and connection factories, are best maintained administratively rather than programmatically. The technology underlying these objects is likely to be very different from one implementation of the JMS API to another. Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider.

JMS clients access these objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of the JMS API. Ordinarily, an administrator configures administered objects in a JNDI namespace, and JMS clients then access them by using resource injection.

With GlassFish Server, you can use the `asadmin create-jms-resource` command or the Administration Console to create JMS administered objects in the form of connector resources. You can also specify the resources in a file named `glassfish-resources.xml` that you can bundle with an application.

NetBeans IDE provides a wizard that allows you to create JMS resources for GlassFish Server. See "To Create JMS Resources Using NetBeans IDE" on page 863 for details.

JMS Connection Factories

A *connection factory* is the object a client uses to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator. Each connection factory is an instance of the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` interface. To learn how to create connection factories, see "To Create JMS Resources Using NetBeans IDE" on page 863.

At the beginning of a JMS client program, you usually inject a connection factory resource into a `ConnectionFactory` object. For example, the following code fragment specifies a resource whose JNDI name is `jms/ConnectionFactory` and assigns it to a `ConnectionFactory` object:

```
@Resource(lookup = "jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

In a Java EE application, JMS administered objects are normally placed in the `jms` naming subcontext.

JMS Destinations

A *destination* is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging domain, destinations are called queues. In the pub/sub messaging domain, destinations are called topics. A JMS application can use multiple queues or topics (or both). To learn how to create destination resources, see "To Create JMS Resources Using NetBeans IDE" on page 863.

To create a destination using the GlassFish Server, you create a JMS destination resource that specifies a JNDI name for the destination.

In the GlassFish Server implementation of JMS, each destination resource refers to a physical destination. You can create a physical destination explicitly, but if you do not, the Application Server creates it when it is needed and deletes it when you delete the destination resource.

In addition to injecting a connection factory resource into a client program, you usually inject a destination resource. Unlike connection factories, destinations are specific to one domain or the other. To create an application that allows you to use the same code for both topics and queues, you assign the destination to a `Destination` object.

The following code specifies two resources, a queue and a topic. The resource names are mapped to destination resources created in the JNDI namespace.

```
@Resource(lookup = "jms/Queue")
private static Queue queue;

@Resource(lookup = "jms/Topic")
private static Topic topic;
```

With the common interfaces, you can mix or match connection factories and destinations. That is, in addition to using the `ConnectionFactory` interface, you can inject a `QueueConnectionFactory` resource and use it with a `Topic`, and you can inject a `TopicConnectionFactory` resource and use it with a `Queue`. The behavior of the application will depend on the kind of destination you use and not on the kind of connection factory you use.

JMS Connections

A *connection* encapsulates a virtual connection with a JMS provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.

Note – In the Java EE platform, the ability to create multiple sessions from a single connection is limited to application clients. In web and enterprise bean components, a connection can create no more than one session.

Connections implement the `Connection` interface. When you have a `ConnectionFactory` object, you can use it to create a `Connection`:

```
Connection connection = connectionFactory.createConnection();
```

Before an application completes, you must close any connections you have created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers.

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

Before your application can consume messages, you must call the connection's `start` method; for details, see "JMS Message Consumers" on page 837. If you want to stop message delivery temporarily without closing the connection, you call the `stop` method.

JMS Sessions

A *session* is a single-threaded context for producing and consuming messages. You use sessions to create the following:

- Message producers
- Message consumers

- Messages
- Queue browsers
- Temporary queues and topics (see "Creating Temporary Destinations" on page 846)

Sessions serialize the execution of message listeners; for details, see "JMS Message Listeners" on page 838.

A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work. For details, see "Using JMS API Local Transactions" on page 849.

Sessions implement the `Session` interface. After you create a `Connection` object, you use it to create a `Session`:

```
Session session = connection.createSession(false,
     Session.AUTO_ACKNOWLEDGE);
```

The first argument means the session is not transacted; the second means the session automatically acknowledges messages when they have been received successfully. (For more information, see "Controlling Message Acknowledgment" on page 843.)

To create a transacted session, use the following code:

```
Session session = connection.createSession(true, 0);
```

Here, the first argument means the session is transacted; the second indicates that message acknowledgment is not specified for transacted sessions. For more information on transactions, see "Using JMS API Local Transactions" on page 849. For information about the way JMS transactions work in Java EE applications, see "Using the JMS API in Java EE Applications" on page 851.

JMS Message Producers

A *message producer* is an object that is created by a session and used for sending messages to a destination. It implements the `MessageProducer` interface.

You use a `Session` to create a `MessageProducer` for a destination. The following examples show that you can create a producer for a `Destination` object, a `Queue` object, or a `Topic` object.

```
MessageProducer producer = session.createProducer(dest);
MessageProducer producer = session.createProducer(queue);
MessageProducer producer = session.createProducer(topic);
```

You can create an unidentified producer by specifying `null` as the argument to `createProducer`. With an unidentified producer, you do not specify a destination until you send a message.

After you have created a message producer, you can use it to send messages by using the `send` method:

```
producer.send(message);
```

You must first create the messages; see "JMS Messages" on page 839.

If you have created an unidentified producer, use an overloaded `send` method that specifies the destination as the first parameter. For example:

```
MessageProducer anon_prod = session.createProducer(null);
anon_prod.send(dest, message);
```

JMS Message Consumers

A *message consumer* is an object that is created by a session and used for receiving messages sent to a destination. It implements the `MessageConsumer` interface.

A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

For example, you could use a `Session` to create a `MessageConsumer` for a `Destination` object, a `Queue` object, or a `Topic` object:

```
MessageConsumer consumer = session.createConsumer(dest);
MessageConsumer consumer = session.createConsumer(queue);
MessageConsumer consumer = session.createConsumer(topic);
```

You use the `Session.createDurableSubscriber` method to create a durable topic subscriber. This method is valid only if you are using a topic. For details, see "Creating Durable Subscriptions" on page 847.

After you have created a message consumer it becomes active, and you can use it to receive messages. You can use the `close` method for a `MessageConsumer` to make the message consumer inactive. Message delivery does not begin until you start the connection you created by calling its `start` method. (Remember always to call the `start` method; forgetting to start the connection is one of the most common JMS programming errors.)

You use the `receive` method to consume a message synchronously. You can use this method at any time after you call the `start` method:

```
connection.start();
Message m = consumer.receive();
connection.start();
Message m = consumer.receive(1000); // time out after a second
```

To consume a message asynchronously, you use a message listener, as described in the next section.

JMS Message Listeners

A message listener is an object that acts as an asynchronous event handler for messages. This object implements the `MessageListener` interface, which contains one method, `onMessage`. In the `onMessage` method, you define the actions to be taken when a message arrives.

You register the message listener with a specific MessageConsumer by using the `setMessageListener` method. For example, if you define a class named `Listener` that implements the `MessageListener` interface, you can register the message listener as follows:

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

Note – In the Java EE platform, a `MessageListener` can be used only in an application client, not in a web component or enterprise bean.

After you register the message listener, you call the `start` method on the `Connection` to begin message delivery. (If you call `start` before you register the message listener, you are likely to miss messages.)

When message delivery begins, the JMS provider automatically calls the message listener's `onMessage` method whenever a message is delivered. The `onMessage` method takes one argument of type `Message`, which your implementation of the method can cast to any of the other message types (see "Message Bodies" on page 840).

A message listener is not specific to a particular destination type. The same listener can obtain messages from either a queue or a topic, depending on the type of destination for which the message consumer was created. A message listener does, however, usually expect a specific message type and format.

Your `onMessage` method should handle all exceptions. It must not throw checked exceptions, and throwing a `RuntimeException` is considered a programming error.

The session used to create the message consumer serializes the execution of all message listeners registered with the session. At any time, only one of the session's message listeners is running.

In the Java EE platform, a message-driven bean is a special kind of message listener. For details, see "Using Message-Driven Beans to Receive Messages Asynchronously" on page 853.

JMS Message Selectors

If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages that interest it. Message selectors assign the work of filtering messages to the JMS provider rather than to the application. For an example of an application that uses a message selector, see "An Application That Uses the JMS API with a Session Bean" on page 897.

A message selector is a `String` that contains an expression. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax. The message selector in the example selects any message that has a `NewsType` property that is set to the value `'Sports'` or `'Opinion'`:

```
NewsType = 'Sports' OR NewsType = 'Opinion'
```

The `createConsumer` and `createDurableSubscriber` methods allow you to specify a message selector as an argument when you create a message consumer.

The message consumer then receives only messages whose headers and properties match the selector. (See "Message Headers" on page 839, and "Message Properties" on page 840.) A message selector cannot select messages on the basis of the content of the message body.

JMS Messages

The ultimate purpose of a JMS application is to produce and consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

A JMS message can have three parts: a header, properties, and a body. Only the header is required. The following sections describe these parts.

For complete documentation of message headers, properties, and bodies, see the documentation of the `Message` interface in the API documentation.

Message Headers

A JMS message header contains a number of predefined fields that contain values used by both clients and providers to identify and route messages. Table 47–1 lists the JMS message header fields and indicates how their values are set. For example, every message has a unique identifier, which is represented in the header field `JMSMessageID`. The value of another header field, `JMSDestination`, represents the queue or the topic to which the message is sent. Other fields include a timestamp and a priority level.

Each header field has associated setter and getter methods, which are documented in the description of the `Message` interface. Some header fields are intended to be set by a client, but many are set automatically by the `send` or the `publish` method, which overrides any client-set values.

TABLE 47–1 How JMS Message Header Field Values Are Set

Header Field	Set By
`JMSDestination`	`send` or `publish` method
`JMSDeliveryMode`	`send` or `publish` method
`JMSExpiration`	`send` or `publish` method
`JMSPriority`	`send` or `publish` method
`JMSMessageID`	`send` or `publish` method
`JMSTimestamp`	`send` or `publish` method
`JMSCorrelationID`	Client
`JMSReplyTo`	Client
`JMSType`	Client
`JMSRedelivered`	JMS provider

Message Properties

You can create and set properties for messages if you need values in addition to those provided by the header fields. You can use properties to provide compatibility with other messaging systems, or you can use them to create message selectors (see "JMS Message Selectors" on page 839). For an example of setting a property to be used as a message selector, see "An Application That Uses the JMS API with a Session Bean" on page 897.

The JMS API provides some predefined property names that a provider can support. The use of these predefined properties or of user-defined properties is optional.

Message Bodies

The JMS API defines five message body formats, also called message types, which allow you to send and receive data in many different forms and which provide compatibility with existing messaging formats. Table 47–2 describes these message types.

TABLE 47–2 JMS Message Types

Message Type	Body Contains
`TextMessage`	A `java.lang.String` object (for example, the contents of an XML file).

TABLE 47–2 JMS Message Types *(Continued)*

Message Type	Body Contains
`MapMessage`	A set of name-value pairs, with names as `String` objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
`BytesMessage`	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
`StreamMessage`	A stream of primitive values in the Java programming language, filled and read sequentially.
`ObjectMessage`	A `Serializable` object in the Java programming language.
`Message`	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

The JMS API provides methods for creating messages of each type and for filling in their contents. For example, to create and send a `TextMessage`, you might use the following statements:

```
TextMessage message = session.createTextMessage();
message.setText(msg_text);     // msg_text is a String
producer.send(message);
```

At the consuming end, a message arrives as a generic `Message` object and must be cast to the appropriate message type. You can use one or more getter methods to extract the message contents. The following code fragment uses the `getText` method:

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
    TextMessage message = (TextMessage) m;
    System.out.println("Reading message: " + message.getText());
} else {
    // Handle error
}
```

JMS Queue Browsers

Messages sent to a queue remain in the queue until the message consumer for that queue consumes them. The JMS API provides a `QueueBrowser` object that allows you to browse the messages in the queue and display the header values for each message. To create a `QueueBrowser` object, use the `Session.createBrowser` method. For example:

```
QueueBrowser browser = session.createBrowser(queue);
```

See "A Simple Example of Browsing Messages in a Queue" on page 875 for an example of using a `QueueBrowser` object.

The `createBrowser` method allows you to specify a message selector as a second argument when you create a `QueueBrowser`. For information on message selectors, see "JMS Message Selectors" on page 839.

The JMS API provides no mechanism for browsing a topic. Messages usually disappear from a topic as soon as they appear: If there are no message consumers to consume them, the JMS provider removes them. Although durable subscriptions allow messages to remain on a topic while the message consumer is not active, no facility exists for examining them.

JMS Exception Handling

The root class for exceptions thrown by JMS API methods is `JMSException`. Catching `JMSException` provides a generic way of handling all exceptions related to the JMS API.

The `JMSException` class includes the following subclasses, described in the API documentation:

- `IllegalStateException`
- `InvalidClientIDException`
- `InvalidDestinationException`
- `InvalidSelectorException`
- `JMSSecurityException`
- `MessageEOFException`
- `MessageFormatException`
- `MessageNotReadableException`
- `MessageNotWriteableException`
- `ResourceAllocationException`
- `TransactionInProgressException`
- `TransactionRolledBackException`

All the examples in the tutorial catch and handle `JMSException` when it is appropriate to do so.

Creating Robust JMS Applications

This section explains how to use features of the JMS API to achieve the level of reliability and performance your application requires. Many people choose to implement JMS applications because they cannot tolerate dropped or duplicate messages and because they require that every message be received once and only once. The JMS API provides this functionality.

The most reliable way to produce a message is to send a `PERSISTENT` message within a transaction. JMS messages are `PERSISTENT` by default. A *transaction* is a unit of work into which you can group a series of operations, such as message sends and receives, so that the operations either all succeed or all fail. For details, see "Specifying Message Persistence" on page 845 and "Using JMS API Local Transactions" on page 849.

The most reliable way to consume a message is to do so within a transaction, either from a queue or from a durable subscription to a topic. For details, see "Creating Temporary Destinations" on page 846, "Creating Durable Subscriptions" on page 847, and "Using JMS API Local Transactions" on page 849.

For other applications, a lower level of reliability can reduce overhead and improve performance. You can send messages with varying priority levels (see "Setting Message Priority Levels" on page 845) and you can set them to expire after a certain length of time (see "Allowing Messages to Expire" on page 846).

The JMS API provides several ways to achieve various kinds and degrees of reliability. This section divides them into two categories, basic and advanced.

The following sections describe these features as they apply to JMS clients. Some of the features work differently in Java EE applications; in these cases, the differences are noted here and are explained in detail in "Using the JMS API in Java EE Applications" on page 851.

Using Basic Reliability Mechanisms

The basic mechanisms for achieving or affecting reliable message delivery are as follows:

- **Controlling message acknowledgment**: You can specify various levels of control over message acknowledgment.
- **Specifying message persistence**: You can specify that messages are persistent, meaning they must not be lost in the event of a provider failure.
- **Setting message priority levels**: You can set various priority levels for messages, which can affect the order in which the messages are delivered.
- **Allowing messages to expire**: You can specify an expiration time for messages so they will not be delivered if they are obsolete.
- **Creating temporary destinations**: You can create temporary destinations that last only for the duration of the connection in which they are created.

Controlling Message Acknowledgment

Until a JMS message has been acknowledged, it is not considered to be successfully consumed. The successful consumption of a message ordinarily takes place in three stages.

1. The client receives the message.
2. The client processes the message.
3. The message is acknowledged. Acknowledgment is initiated either by the JMS provider or by the client, depending on the session acknowledgment mode.

In transacted sessions (see "Using JMS API Local Transactions" on page 849), acknowledgment happens automatically when a transaction is committed. If a transaction is rolled back, all consumed messages are redelivered.

In nontransacted sessions, when and how a message is acknowledged depend on the value specified as the second argument of the `createSession` method. The three possible argument values are as follows:

- `Session.AUTO_ACKNOWLEDGE`: The session automatically acknowledges a client's receipt of a message either when the client has successfully returned from a call to `receive` or when the `MessageListener` it has called to process the message returns successfully.

  A synchronous receive in an `AUTO_ACKNOWLEDGE` session is the one exception to the rule that message consumption is a three-stage process as described earlier. In this case, the receipt and acknowledgment take place in one step, followed by the processing of the message.

- `Session.CLIENT_ACKNOWLEDGE`: A client acknowledges a message by calling the message's `acknowledge` method. In this mode, acknowledgment takes place on the session level: Acknowledging a consumed message automatically acknowledges the receipt of *all* messages that have been consumed by its session. For example, if a message consumer consumes ten messages and then acknowledges the fifth message delivered, all ten messages are acknowledged.

  Note – In the Java EE platform, a `CLIENT_ACKNOWLEDGE` session can be used only in an application client, not in a web component or enterprise bean.

- `Session.DUPS_OK_ACKNOWLEDGE`: This option instructs the session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should be used only by consumers that can tolerate duplicate messages. (If the JMS provider redelivers a message, it must set the value of the `JMSRedelivered` message header to `true`.) This option can reduce session overhead by minimizing the work the session does to prevent duplicates.

If messages have been received from a queue but not acknowledged when a session terminates, the JMS provider retains them and redelivers them when a consumer next accesses the queue. The provider also retains unacknowledged messages for a terminated session that has a durable `TopicSubscriber`. (See "Creating Durable Subscriptions" on page 847.) Unacknowledged messages for a nondurable `TopicSubscriber` are dropped when the session is closed.

If you use a queue or a durable subscription, you can use the `Session.recover` method to stop a nontransacted session and restart it with its first unacknowledged message. In effect, the session's series of delivered messages is reset to the point after its last acknowledged message. The messages it now delivers may be different from those that were originally delivered, if messages have expired or if higher-priority messages have arrived. For a nondurable `TopicSubscriber`, the provider may drop unacknowledged messages when its session is recovered.

The sample program in "A Message Acknowledgment Example" on page 887 demonstrates two ways to ensure that a message will not be acknowledged until processing of the message is complete.

Specifying Message Persistence

The JMS API supports two delivery modes specifying whether messages are lost if the JMS provider fails. These delivery modes are fields of the `DeliveryMode` interface.

- The `PERSISTENT` delivery mode, the default, instructs the JMS provider to take extra care to ensure that a message is not lost in transit in case of a JMS provider failure. A message sent with this delivery mode is logged to stable storage when it is sent.
- The `NON_PERSISTENT` delivery mode does not require the JMS provider to store the message or otherwise guarantee that it is not lost if the provider fails.

You can specify the delivery mode in either of two ways.

- You can use the `setDeliveryMode` method of the `MessageProducer` interface to set the delivery mode for all messages sent by that producer. For example, the following call sets the delivery mode to `NON_PERSISTENT` for a producer:

  ```
  producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
  ```

- You can use the long form of the `send` or the `publish` method to set the delivery mode for a specific message. The second argument sets the delivery mode. For example, the following `send` call sets the delivery mode for `message` to `NON_PERSISTENT`:

  ```
  producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
  ```

  The third and fourth arguments set the priority level and expiration time, which are described in the next two subsections.

If you do not specify a delivery mode, the default is `PERSISTENT`. Using the `NON_PERSISTENT` delivery mode may improve performance and reduce storage overhead, but you should use it only if your application can afford to miss messages.

Setting Message Priority Levels

You can use message priority levels to instruct the JMS provider to deliver urgent messages first. You can set the priority level in either of two ways.

- You can use the `setPriority` method of the `MessageProducer` interface to set the priority level for all messages sent by that producer. For example, the following call sets a priority level of 7 for a producer:

  ```
  producer.setPriority(7);
  ```

- You can use the long form of the `send` or the `publish` method to set the priority level for a specific message. The third argument sets the priority level. For example, the following `send` call sets the priority level for `message` to 3:

  ```
  producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
  ```

The ten levels of priority range from 0 (lowest) to 9 (highest). If you do not specify a priority level, the default level is 4. A JMS provider tries to deliver higher-priority messages before lower-priority ones but does not have to deliver messages in exact order of priority.

Allowing Messages to Expire

By default, a message never expires. If a message will become obsolete after a certain period, however, you may want to set an expiration time. You can do this in either of two ways.

- You can use the `setTimeToLive` method of the `MessageProducer` interface to set a default expiration time for all messages sent by that producer. For example, the following call sets a time to live of one minute for a producer:

  ```
  producer.setTimeToLive(60000);
  ```

- You can use the long form of the `send` or the `publish` method to set an expiration time for a specific message. The fourth argument sets the expiration time in milliseconds. For example, the following `send` call sets a time to live of 10 seconds:

  ```
  producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
  ```

If the specified `timeToLive` value is `0`, the message never expires.

When the message is sent, the specified `timeToLive` is added to the current time to give the expiration time. Any message not delivered before the specified expiration time is destroyed. The destruction of obsolete messages conserves storage and computing resources.

Creating Temporary Destinations

Normally, you create JMS destinations (queues and topics) administratively rather than programmatically. Your JMS provider includes a tool to create and remove destinations, and it is common for destinations to be long-lasting.

The JMS API also enables you to create destinations (`TemporaryQueue` and `TemporaryTopic` objects) that last only for the duration of the connection in which they are created. You create these destinations dynamically using the `Session.createTemporaryQueue` and the `Session.createTemporaryTopic` methods.

The only message consumers that can consume from a temporary destination are those created by the same connection that created the destination. Any message producer can send to the temporary destination. If you close the connection to which a temporary destination belongs, the destination is closed and its contents are lost.

You can use temporary destinations to implement a simple request/reply mechanism. If you create a temporary destination and specify it as the value of the `JMSReplyTo` message header field when you send a message, then the consumer of the message can use the value of the `JMSReplyTo` field as the destination to which it sends a reply. The consumer can also reference the original request by setting the `JMSCorrelationID` header field of the reply message to the value of the `JMSMessageID` header field of the request. For example, an `onMessage` method can create a session so that it can send a reply to the message it receives. It can use code such as the following:

```
producer = session.createProducer(msg.getJMSReplyTo());
replyMsg = session.createTextMessage("Consumer " +
    "processed message: " + msg.getText());
replyMsg.setJMSCorrelationID(msg.getJMSMessageID());
producer.send(replyMsg);
```

For more examples, see Chapter 48, "Java Message Service Examples."

Using Advanced Reliability Mechanisms

The more advanced mechanisms for achieving reliable message delivery are the following:

- **Creating durable subscriptions**: You can create durable topic subscriptions, which receive messages published while the subscriber is not active. Durable subscriptions offer the reliability of queues to the publish/subscribe message domain.
- **Using local transactions**: You can use local transactions, which allow you to group a series of sends and receives into an atomic unit of work. Transactions are rolled back if they fail at any time.

Creating Durable Subscriptions

To ensure that a pub/sub application receives all published messages, use `PERSISTENT` delivery mode for the publishers and durable subscriptions for the subscribers.

The `Session.createConsumer` method creates a nondurable subscriber if a topic is specified as the destination. A nondurable subscriber can receive only messages that are published while it is active.

At the cost of higher overhead, you can use the `Session.createDurableSubscriber` method to create a durable subscriber. A durable subscription can have only one active subscriber at a time.

A durable subscriber registers a durable subscription by specifying a unique identity that is retained by the JMS provider. Subsequent subscriber objects that have the same identity resume the subscription in the state in which it was left by the preceding subscriber. If a durable subscription has no active subscriber, the JMS provider retains the subscription's messages until they are received by the subscription or until they expire.

You establish the unique identity of a durable subscriber by setting the following:

- A client ID for the connection
- A topic and a subscription name for the subscriber

You set the client ID administratively for a client-specific connection factory using either the command line or the Administration Console.

After using this connection factory to create the connection and the session, you call the `createDurableSubscriber` method with two arguments: the topic and a string that specifies the name of the subscription:

```
String subName = "MySub";
MessageConsumer topicSubscriber =
     session.createDurableSubscriber(myTopic, subName);
```

The subscriber becomes active after you start the `Connection` or `TopicConnection`. Later, you might close the subscriber:

```
topicSubscriber.close();
```

The JMS provider stores the messages sent or published to the topic, as it would store messages sent to a queue. If the program or another application calls `createDurableSubscriber` using the same connection factory and its client ID, the same topic, and the same subscription name, then the subscription is reactivated and the JMS provider delivers any messages that were published while the subscriber was inactive.

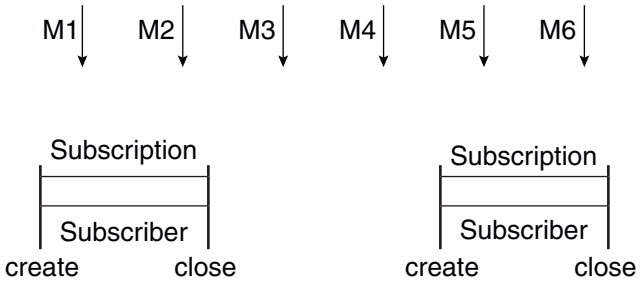
To delete a durable subscription, first close the subscriber, then use the `unsubscribe` method with the subscription name as the argument:

```
topicSubscriber.close();
session.unsubscribe("MySub");
```

The `unsubscribe` method deletes the state the provider maintains for the subscriber.

Figure 47–6 and Figure 47–7 show the difference between a nondurable and a durable subscriber. With an ordinary, nondurable subscriber, the subscriber and the subscription begin and end at the same point and are, in effect, identical. When a subscriber is closed, the subscription also ends. Here, `create` stands for a call to `Session.createConsumer` with a `Topic` argument, and `close` stands for a call to `MessageConsumer.close`. Any messages published to the topic between the time of the first `close` and the time of the second `create` are not consumed by the subscriber. In Figure 47–6, the subscriber consumes messages M1, M2, M5, and M6, but messages M3 and M4 are lost.

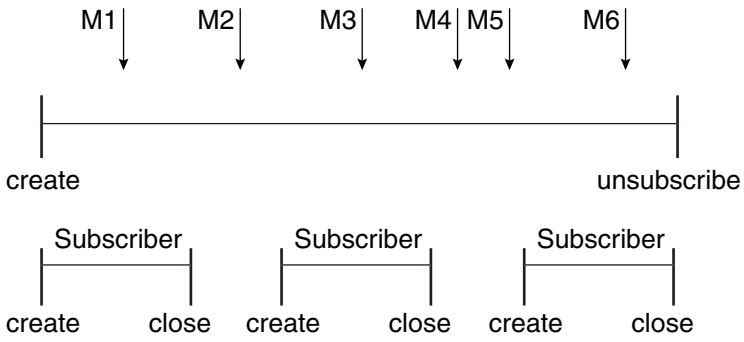
FIGURE 47–6 Nondurable Subscribers and Subscriptions



With a durable subscriber, the subscriber can be closed and re-created, but the subscription continues to exist and to hold messages until the application calls the `unsubscribe` method. In

Figure 47–7, `create` stands for a call to `Session.createDurableSubscriber`, `close` stands for a call to `MessageConsumer.close`, and `unsubscribe` stands for a call to `Session.unsubscribe`. Messages published while the subscriber is closed are received when the subscriber is created again, so even though messages M2, M4, and M5 arrive while the subscriber is closed, they are not lost.

FIGURE 47–7 A Durable Subscriber and Subscription



See "A Message Acknowledgment Example" on page 887, "A Durable Subscription Example" on page 889, and "An Application That Uses the JMS API with a Session Bean" on page 897 for examples of Java EE applications that use durable subscriptions.

Using JMS API Local Transactions

You can group a series of operations into an atomic unit of work called a *transaction*. If any one of the operations fails, the transaction can be rolled back, and the operations can be attempted again from the beginning. If all the operations succeed, the transaction can be committed.

In a JMS client, you can use local transactions to group message sends and receives. The JMS API `Session` interface provides `commit` and `rollback` methods you can use in a JMS client. A transaction commit means that all produced messages are sent and all consumed messages are acknowledged. A transaction rollback means that all produced messages are destroyed and all consumed messages are recovered and redelivered unless they have expired (see "Allowing Messages to Expire" on page 846).

A transacted session is always involved in a transaction. As soon as the `commit` or the `rollback` method is called, one transaction ends and another transaction begins. Closing a transacted session rolls back its transaction in progress, including any pending sends and receives.

In an Enterprise JavaBeans component, you cannot use the `Session.commit` and `Session.rollback` methods. Instead, you use distributed transactions, described in "Using the JMS API in Java EE Applications" on page 851.

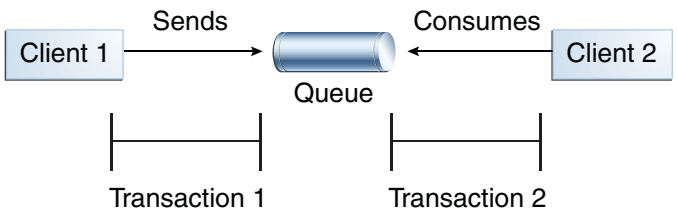
You can combine several sends and receives in a single JMS API local transaction. If you do so, you need to be careful about the order of the operations. You will have no problems if the transaction consists of all sends or all receives, or if the receives all come before the sends. However, if you try to use a request/reply mechanism, in which you send a message and then try to receive a reply to that message in the same transaction, the program will hang, because the send cannot take place until the transaction is committed. The following code fragment illustrates the problem:

```
// Don't do this!
outMsg.setJMSReplyTo(replyQueue);
producer.send(outQueue, outMsg);
consumer = session.createConsumer(replyQueue);
inMsg = consumer.receive();
session.commit();
```

Because a message sent during a transaction is not actually sent until the transaction is committed, the transaction cannot contain any receives that depend on that message's having been sent.

In addition, the production and the consumption of a message cannot both be part of the same transaction. The reason is that the transactions take place between the clients and the JMS provider, which intervenes between the production and the consumption of the message. Figure 47–8 illustrates this interaction.

FIGURE 47–8 Using JMS API Local Transactions



The sending of one or more messages to one or more destinations by Client 1 can form a single transaction, because it forms a single set of interactions with the JMS provider using a single session. Similarly, the receiving of one or more messages from one or more destinations by Client 2 also forms a single transaction using a single session. But because the two clients have no direct interaction and are using two different sessions, no transactions can take place between them.

Another way of putting this is that the act of producing and/or consuming messages in a session can be transactional, but the act of producing and consuming a specific message across different sessions cannot be transactional.

This is the fundamental difference between messaging and synchronized processing. Instead of tightly coupling the sending and receiving of data, message producers and consumers use an alternative approach to reliability, built on a JMS provider's ability to supply a once-and-only-once message delivery guarantee.

When you create a session, you specify whether it is transacted. The first argument to the `createSession` method is a `boolean` value. A value of `true` means the session is transacted; a value of `false` means it is not transacted. The second argument to this method is the acknowledgment mode, which is relevant only to nontransacted sessions (see "Controlling Message Acknowledgment" on page 843). If the session is transacted, the second argument is ignored, so it is a good idea to specify `0` to make the meaning of your code clear. For example:

```
session = connection.createSession(true, 0);
```

The `commit` and the `rollback` methods for local transactions are associated with the session. You can combine queue and topic operations in a single transaction if you use the same session to perform the operations. For example, you can use the same session to receive a message from a queue and send a message to a topic in the same transaction.

You can pass a client program's session to a message listener's constructor function and use it to create a message producer. In this way, you can use the same session for receives and sends in asynchronous message consumers.

"A Local Transaction Example" on page 891 provides an example of using JMS API local transactions.

Using the JMS API in Java EE Applications

This section describes how using the JMS API in enterprise bean applications or web applications differs from using it in application clients.

A general rule in the Java EE platform specification applies to all Java EE components that use the JMS API within EJB or web containers: Application components in the web and EJB containers must not attempt to create more than one active (not closed) `Session` object per connection.

This rule does not apply to application clients. The application client container supports the creation of multiple sessions for each connection.

Using `@Resource` Annotations in Enterprise Bean or Web Components

When you use the `@Resource` annotation in an application client component, you normally declare the JMS resource static:

```
@Resource(lookup = "jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(lookup = "jms/Queue")
private static Queue queue;
```

However, when you use this annotation in a session bean, a message-driven bean, or a web component, do *not* declare the resource static:

```
@Resource(lookup = "jms/ConnectionFactory")
private ConnectionFactory connectionFactory;

@Resource(lookup = "jms/Topic")
private Topic topic;
```

If you declare the resource static in these components, runtime errors will result.

Using Session Beans to Produce and to Synchronously Receive Messages

An application that produces messages or synchronously receives them can use a session bean to perform these operations. The example in "An Application That Uses the JMS API with a Session Bean" on page 897 uses a stateless session bean to publish messages to a topic.

Because a blocking synchronous receive ties up server resources, it is not a good programming practice to use such a `receive` call in an enterprise bean. Instead, use a timed synchronous receive, or use a message-driven bean to receive messages asynchronously. For details about blocking and timed synchronous receives, see "Writing the Clients for the Synchronous Receive Example" on page 860.

Using the JMS API in an enterprise bean is in many ways similar to using it in an application client. The main differences are the areas of resource management and transactions.

Managing JMS Resources in Session Beans

The JMS API resources are a JMS API connection and a JMS API session. In general, it is important to release JMS resources when they are no longer being used. Here are some useful practices to follow:

- If you wish to maintain a JMS API resource only for the life span of a business method, it is a good idea to close the resource in a `finally` block within the method.
- If you would like to maintain a JMS API resource for the life span of an enterprise bean instance, it is a good idea to use a `@PostConstruct` callback method to create the resource and to use a `@PreDestroy` callback method to close the resource. If you use a stateful session bean and you wish to maintain the JMS API resource in a cached state, you must close the resource in a `@PrePassivate` callback method and set its value to `null`, and you must create it again in a `@PostActivate` callback method.

Managing Transactions in Session Beans

Instead of using local transactions, you use container-managed transactions for bean methods that perform sends or receives, allowing the EJB container to handle transaction demarcation. Because container-managed transactions are the default, you do not have to use an annotation to specify them.

You can use bean-managed transactions and the `javax.transaction.UserTransaction` interface's transaction demarcation methods, but you should do so only if your application has special requirements and you are an expert in using transactions. Usually, container-managed transactions produce the most efficient and correct behavior. This tutorial does not provide any examples of bean-managed transactions.

Using Message-Driven Beans to Receive Messages Asynchronously

The sections "What Is a Message-Driven Bean?" on page 439 and "How Does the JMS API Work with the Java EE Platform?" on page 827 describe how the Java EE platform supports a special kind of enterprise bean, the message-driven bean, which allows Java EE applications to process JMS messages asynchronously. Session beans allow you to send messages and to receive them synchronously but not asynchronously.

A message-driven bean is a message listener that can reliably consume messages from a queue or a durable subscription. The messages can be sent by any Java EE component (from an application client, another enterprise bean, or a web component) or from an application or a system that does not use Java EE technology.

Like a message listener in an application client, a message-driven bean contains an `onMessage` method that is called automatically when a message arrives. Like a message listener, a message-driven bean class can implement helper methods invoked by the `onMessage` method to aid in message processing.

A message-driven bean, however, differs from an application client's message listener in the following ways:

- Certain setup tasks are performed by the EJB container.
- The bean class uses the `@MessageDriven` annotation to specify properties for the bean or the connection factory, such as a destination type, a durable subscription, a message selector, or an acknowledgment mode. The examples in Chapter 48, "Java Message Service Examples," show how the JMS resource adapter works in the GlassFish Server.

The EJB container automatically performs several setup tasks that a stand-alone client must perform:

- Creating a message consumer to receive the messages. Instead of creating a message consumer in your source code, you associate the message-driven bean with a destination and a connection factory at deployment time. If you want to specify a durable subscription or use a message selector, you do this at deployment time also.
- Registering the message listener. You must not call `setMessageListener`.
- Specifying a message acknowledgment mode. The default mode, `AUTO_ACKNOWLEDGE`, is used unless it is overridden by a property setting.

If JMS is integrated with the application server using a resource adapter, the JMS resource adapter handles these tasks for the EJB container.

Your message-driven bean class must implement the `javax.jms.MessageListener` interface and the `onMessage` method.

It may implement a `@PostConstruct` callback method to create a connection, and a `@PreDestroy` callback method to close the connection. Typically, it implements these methods if it produces messages or performs synchronous receives from another destination.

The bean class commonly injects a `MessageDrivenContext` resource, which provides some additional methods you can use for transaction management.

The main difference between a message-driven bean and a session bean is that a message-driven bean has no local or remote interface. Instead, it has only a bean class.

A message-driven bean is similar in some ways to a stateless session bean: Its instances are relatively short-lived and retain no state for a specific client. The instance variables of the message-driven bean instance can contain some state across the handling of client messages: for example, a JMS API connection, an open database connection, or an object reference to an enterprise bean object.

Like a stateless session bean, a message-driven bean can have many interchangeable instances running at the same time. The container can pool these instances to allow streams of messages to be processed concurrently. The container attempts to deliver messages in chronological order when that would not impair the concurrency of message processing, but no guarantees are made as to the exact order in which messages are delivered to the instances of the message-driven bean class. Because concurrency can affect the order in which messages are delivered, you should write your applications to handle messages that arrive out of sequence.

For example, your application could manage conversations by using application-level sequence numbers. An application-level conversation control mechanism with a persistent conversation state could cache later messages until earlier messages have been processed.

Another way to ensure order is to have each message or message group in a conversation require a confirmation message that the sender blocks on receipt of. This forces the responsibility for order back onto the sender and more tightly couples senders to the progress of message-driven beans.

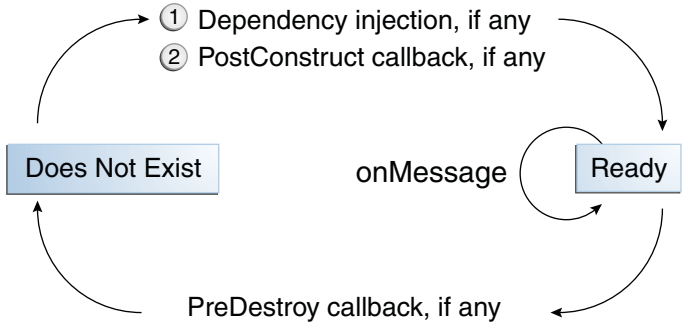
To create a new instance of a message-driven bean, the container does the following:

- Instantiates the bean
- Performs any required resource injection
- Calls the `@PostConstruct` callback method, if it exists

To remove an instance of a message-driven bean, the container calls the `@PreDestroy` callback method.

Figure 47–9 shows the lifecycle of a message-driven bean.

FIGURE 47–9 Lifecycle of a Message-Driven Bean



Managing Distributed Transactions

JMS client applications use JMS API local transactions (described in "Using JMS API Local Transactions" on page 849), which allow the grouping of sends and receives within a specific JMS session. Java EE applications commonly use distributed transactions to ensure the integrity of accesses to external resources. For example, distributed transactions allow multiple applications to perform atomic updates on the same database, and they allow a single application to perform atomic updates on multiple databases.

In a Java EE application that uses the JMS API, you can use transactions to combine message sends or receives with database updates and other resource manager operations. You can access resources from multiple application components within a single transaction. For example, a servlet can start a transaction, access multiple databases, invoke an enterprise bean that sends a JMS message, invoke another enterprise bean that modifies an EIS system using the Connector architecture, and finally commit the transaction. Your application cannot, however, both send a JMS message and receive a reply to it within the same transaction; the restriction described in "Using JMS API Local Transactions" on page 849 still applies.

Distributed transactions within the EJB container can be either of two kinds:

- **Container-managed transactions**: The EJB container controls the integrity of your transactions without your having to call `commit` or `rollback`. Container-managed transactions are recommended for Java EE applications that use the JMS API. You can specify appropriate transaction attributes for your enterprise bean methods.

  Use the `Required` transaction attribute (the default) to ensure that a method is always part of a transaction. If a transaction is in progress when the method is called, the method will be part of that transaction; if not, a new transaction will be started before the method is called and will be committed when the method returns.
- **Bean-managed transactions**: You can use these in conjunction with the `javax.transaction.UserTransaction` interface, which provides its own `commit` and `rollback` methods you can use to delimit transaction boundaries. Bean-managed transactions are recommended only for those who are experienced in programming transactions.

You can use either container-managed transactions or bean-managed transactions with message-driven beans. To ensure that all messages are received and handled within the context of a transaction, use container-managed transactions and use the `Required` transaction attribute (the default) for the `onMessage` method. This means that if there is no transaction in progress, a new transaction will be started before the method is called and will be committed when the method returns.

When you use container-managed transactions, you can call the following `MessageDrivenContext` methods:

- `setRollbackOnly`: Use this method for error handling. If an exception occurs, `setRollbackOnly` marks the current transaction so that the only possible outcome of the transaction is a rollback.
- `getRollbackOnly`: Use this method to test whether the current transaction has been marked for rollback.

If you use bean-managed transactions, the delivery of a message to the `onMessage` method takes place outside the distributed transaction context. The transaction begins when you call the `UserTransaction.begin` method within the `onMessage` method, and it ends when you call `UserTransaction.commit` or `UserTransaction.rollback`. Any call to the `Connection.createSession` method must take place within the transaction. If you call `UserTransaction.rollback`, the message is not redelivered, whereas calling `setRollbackOnly` for container-managed transactions does cause a message to be redelivered.

Neither the JMS API specification nor the Enterprise JavaBeans specification (available from `http://jcp.org/en/jsr/detail?id=318`) specifies how to handle calls to JMS API methods outside transaction boundaries. The Enterprise JavaBeans specification does state that the EJB container is responsible for acknowledging a message that is successfully processed by the `onMessage` method of a message-driven bean that uses bean-managed transactions. Using bean-managed transactions allows you to process the message by using more than one transaction or to have some parts of the message processing take place outside a transaction context. In most cases, however, container-managed transactions provide greater reliability and are therefore preferable.

When you create a session in an enterprise bean, the container ignores the arguments you specify, because it manages all transactional properties for enterprise beans. It is still a good idea to specify arguments of `true` and `0` to the `createSession` method to make this situation clear:

```
session = connection.createSession(true, 0);
```

When you use container-managed transactions, you normally use the `Required` transaction attribute (the default) for your enterprise bean's business methods.

You do not specify a message acknowledgment mode when you create a message-driven bean that uses container-managed transactions. The container acknowledges the message automatically when it commits the transaction.

If a message-driven bean uses bean-managed transactions, the message receipt cannot be part of the bean-managed transaction, so the container acknowledges the message outside the transaction.

If the `onMessage` method throws a `RuntimeException`, the container does not acknowledge processing the message. In that case, the JMS provider will redeliver the unacknowledged message in the future.

Using the JMS API with Application Clients and Web Components

An application client in a Java EE application can use the JMS API in much the same way that a stand-alone client program does. It can produce messages, and it can consume messages by using either synchronous receives or message listeners. See Chapter 25, "A Message-Driven Bean Example," for an example of an application client that produces messages. For an example of using an application client to produce and to consume messages, see "An Application Example That Deploys a Message-Driven Bean on Two Servers" on page 916.

The Java EE platform specification does not impose strict constraints on how web components should use the JMS API. In the GlassFish Server, a web component can send messages and consume them synchronously but cannot consume them asynchronously.

Because a blocking synchronous receive ties up server resources, it is not a good programming practice to use such a `receive` call in a web component. Instead, use a timed synchronous receive. For details about blocking and timed synchronous receives, see "Writing the Clients for the Synchronous Receive Example" on page 860.

Further Information about JMS

For more information about JMS, see:

- Java Message Service web site:

  `http://www.oracle.com/technetwork/java/index-jsp-142945.html`
- Java Message Service specification, version 1.1, available from:

  `http://www.oracle.com/technetwork/java/docs-136352.html`

◆ ◆ ◆ CHAPTER 48

Java Message Service Examples

This chapter provides examples that show how to use the JMS API in various kinds of Java EE applications. It covers the following topics:

- "Writing Simple JMS Applications" on page 860
- "Writing Robust JMS Applications" on page 886
- "An Application That Uses the JMS API with a Session Bean" on page 897
- "An Application That Uses the JMS API with an Entity" on page 902
- "An Application Example That Consumes Messages from a Remote Server" on page 910
- "An Application Example That Deploys a Message-Driven Bean on Two Servers" on page 916

The examples are in the *tut-install*`/examples/jms/` directory.

The steps to build and run each example are as follows:

1. Use NetBeans IDE or Ant to compile and package the example.
2. Use NetBeans IDE or Ant to deploy the example and create resources for it.
3. Use NetBeans IDE, the `appclient` command, or Ant to run the client.

Each example has a `build.xml` file that refers to files in the *tut-install*`/examples/bp-project/` directory.

Each example has a `setup/glassfish-resources.xml` file that is used to create resources for the example.

See Chapter 25, "A Message-Driven Bean Example," for a simpler example of a Java EE application that uses the JMS API.

Writing Simple JMS Applications

This section shows how to create, package, and run simple JMS clients that are packaged as application clients and deployed to a Java EE server. The clients demonstrate the basic tasks a JMS application must perform:

- Creating a connection and a session
- Creating message producers and consumers
- Sending and receiving messages

In a Java EE application, some of these tasks are performed, in whole or in part, by the container. If you learn about these tasks, you will have a good basis for understanding how a JMS application works on the Java EE platform.

Each example uses two clients: one that sends messages and one that receives them. You can run the clients in NetBeans IDE or in two terminal windows.

When you write a JMS client to run in an enterprise bean application, you use many of the same methods in much the same sequence as for an application client. However, there are some significant differences. "Using the JMS API in Java EE Applications" on page 851 describes these differences, and this chapter provides examples that illustrate them.

The examples for this section are in the *tut-install*`/examples/jms/simple/` directory, under the following four subdirectories:

```
producer
synchconsumer
asynchconsumer
messagebrowser
```

A Simple Example of Synchronous Message Receives

This section describes the sending and receiving clients in an example that uses the `receive` method to consume messages synchronously. This section then explains how to compile, package, and run the clients using the GlassFish Server.

The following subsections describe the steps in creating and running the example.

Writing the Clients for the Synchronous Receive Example

The sending client, `producer/src/java/Producer.java`, performs the following steps:

1. Injects resources for a connection factory, queue, and topic:

```
@Resource(lookup = "jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(lookup = "jms/Queue")private static Queue queue;
@Resource(lookup = "jms/Topic")private static Topic topic;
```

2. Retrieves and verifies command-line arguments that specify the destination type and the number of arguments:

```
final int NUM_MSGS;
String destType = args[0];
System.out.println("Destination type is " + destType);
if ( ! ( destType.equals("queue") || destType.equals("topic") ) ) {
    System.err.println("Argument must be \"queue\" or " + "\"topic\"");
    System.exit(1);
}
if (args.length == 2){
    NUM_MSGS = (new Integer(args[1])).intValue();
}
else {
    NUM_MSGS = 1;
}
```

3. Assigns either the queue or the topic to a destination object, based on the specified destination type:

```
Destination dest = null;
try {
    if (destType.equals("queue")) {
        dest = (Destination) queue;
    } else {
        dest = (Destination) topic;
    }
}
catch (Exception e) {
    System.err.println("Error setting destination: " + e.toString());
    e.printStackTrace();
    System.exit(1);
}
```

4. Creates a `Connection` and a `Session`:

```
Connection connection = connectionFactory.createConnection();
Session session = connection.createSession(
            false,
            Session.AUTO_ACKNOWLEDGE);
```

5. Creates a `MessageProducer` and a `TextMessage`:

```
MessageProducer producer = session.createProducer(dest);
TextMessage message = session.createTextMessage();
```

6. Sends one or more messages to the destination:

```
for (int i = 0; i < NUM_MSGS; i++) {
    message.setText("This is message " + (i + 1) + " from producer");
    System.out.println("Sending message: " + message.getText());
    producer.send(message);
}
```

7. Sends an empty control message to indicate the end of the message stream:

```
producer.send(session.createMessage());
```

   Sending an empty message of no specified type is a convenient way to indicate to the consumer that the final message has arrived.

8. Closes the connection in a `finally` block, automatically closing the session and `MessageProducer`:

```
} finally {
    if (connection != null) {
        try { connection.close(); }
        catch (JMSException e) { }
    }
}
```

The receiving client, `synchconsumer/src/java/SynchConsumer.java`, performs the following steps:

1. Injects resources for a connection factory, queue, and topic.
2. Assigns either the queue or the topic to a destination object, based on the specified destination type.
3. Creates a `Connection` and a `Session`.
4. Creates a `MessageConsumer`:

```
consumer = session.createConsumer(dest);
```

5. Starts the connection, causing message delivery to begin:

```
connection.start();
```

6. Receives the messages sent to the destination until the end-of-message-stream control message is received:

```
while (true) {
    Message m = consumer.receive(1);
    if (m != null) {
        if (m instanceof TextMessage) {
            message = (TextMessage) m;
            System.out.println("Reading message: " + message.getText());
        } else {
            break;
        }
    }
}
```

   Because the control message is not a `TextMessage`, the receiving client terminates the `while` loop and stops receiving messages after the control message arrives.

7. Closes the connection in a `finally` block, automatically closing the session and `MessageConsumer`.

The `receive` method can be used in several ways to perform a synchronous receive. If you specify no arguments or an argument of `0`, the method blocks indefinitely until a message arrives:

```
Message m = consumer.receive();
Message m = consumer.receive(0);
```

For a simple client, this may not matter. But if you do not want your application to consume system resources unnecessarily, use a timed synchronous receive. Do one of the following:

- Call the `receive` method with a timeout argument greater than `0`:

```
Message m = consumer.receive(1); // 1 millisecond
```

- Call the `receiveNoWait` method, which receives a message only if one is available:

```
Message m = consumer.receiveNoWait();
```

The `SynchConsumer` client uses an indefinite `while` loop to receive messages, calling `receive` with a timeout argument. Calling `receiveNoWait` would have the same effect.

Starting the JMS Provider

When you use the GlassFish Server, your JMS provider is the GlassFish Server. Start the server as described in "Starting and Stopping the GlassFish Server" on page 73.

JMS Administered Objects for the Synchronous Receive Example

This example uses the following JMS administered objects:

- A connection factory
- Two destination resources: a topic and a queue

NetBeans IDE and the Ant tasks for the JMS examples create needed JMS resources when you deploy the applications, using a file named `setup/glassfish-resources.xml`. This file is most easily created using NetBeans IDE, although you can create it by hand.

You can also use the `asadmin create-jms-resource` command to create resources, the `asadmin list-jms-resources` command to display their names, and the `asadmin delete-jms-resource` command to remove them.

▼ To Create JMS Resources Using NetBeans IDE

Follow these steps to create a JMS resource in GlassFish Server using NetBeans IDE. Repeat these steps for each resource you need.

The example applications in this chapter already have the resources, so you will need to follow these steps only when you create your own applications.

1. **Right-click the project for which you want to create resources and choose New, then choose Other.**

   The New File wizard opens.

2. **Under Categories, select GlassFish.**

3. **Under File Types, select JMS Resource.**

   The General Attributes - JMS Resource page opens.

4. **In the JNDI Name field, type the name of the resource.**

   By convention, JMS resource names begin with `jms/`.

5. **Select the radio button for the resource type.**

   Normally, this is either `javax.jms.Queue`, `javax.jms.Topic`, or `javax.jms.ConnectionFactory`.

6. **Click Next.**

   The JMS Properties page opens.

7. **For a queue or topic, type a name for a physical queue in the Value field for the Name property.**

   You can type any value for this required field.

   Connection factories have no required properties. In a few situations, discussed in later sections, you may need to specify a property.

8. **Click Finish.**

   A file named `glassfish-resources.xml` is created in your project, in a directory named `setup`. In the project pane, you can find it under the Server Resources node. If this file exists, resources are created automatically by NetBeans IDE when you deploy the project.

▼ To Delete JMS Resources Using NetBeans IDE

1. **In the Services pane, expand the Servers node, then expand the GlassFish Server 3+ node.**
2. **Expand the Resources node, then expand the Connector Resources node.**
3. **Expand the Admin Object Resources node.**
4. **Right-click any destination you want to remove and select Unregister.**
5. **Expand the Connector Connection Pools node.**
6. **Right-click any connection factory you want to remove and select Unregister.**

   Every connection factory has both a connector connection pool and an associated connector resource. When you remove the connector connection pool, the resource is removed automatically. You can verify the removal by expanding the Connector Resources node.

Running the Clients for the Synchronous Receive Example

To run these examples using the GlassFish Server, package each one in an application client JAR file. The application client JAR file requires a manifest file, located in the `src/conf` directory for each example, along with the `.class` file.

The `build.xml` file for each example contains Ant targets that compile, package, and deploy the example. The targets place the `.class` file for the example in the `build/jar` directory. Then the targets use the `jar` command to package the class file and the manifest file in an application client JAR file.

Because the examples use the common interfaces, you can run them using either a queue or a topic.

▼ To Build and Package the Clients for the Synchronous Receive Example Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**
   tut-install`/examples/jms/simple/`
3. **Select the `producer` folder.**
4. **Select the Open as Main Project check box.**
5. **Click Open Project.**
6. **In the Projects tab, right-click the project and select Build.**
7. **From the File menu, choose Open Project again.**
8. **Select the `synchconsumer` folder.**
9. **Select the Open as Main Project check box.**
10. **Click Open Project.**
11. **In the Projects tab, right-click the project and select Build.**

▼ To Deploy and Run the Clients for the Synchronous Receive Example Using NetBeans IDE

1. **Deploy and run the `Producer` example:**
   a. **Right-click the `producer` project and select Properties.**
   b. **Select Run from the Categories tree.**

c. **In the Arguments field, type the following:**

```
queue 3
```

d. **Click OK.**

e. **Right-click the project and select Run.**

The output of the program looks like this (along with some additional output):

```
Destination type is queue
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
```

The messages are now in the queue, waiting to be received.

Note – When you run an application client, the command often takes a long time to complete.

---

2 **Now deploy and run the `SynchConsumer` example:**

a. **Right-click the `synchconsumer` project and select Properties.**

b. **Select Run from the Categories tree.**

c. **In the Arguments field, type the following:**

```
queue
```

d. **Click OK.**

e. **Right-click the project and select Run.**

The output of the program looks like this (along with some additional output):

```
Destination type is queue
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
```

3 **Now try running the programs in the opposite order. Right-click the `synchconsumer` project and select Run.**

The Output pane displays the destination type and then appears to hang, waiting for messages.

4 **Right-click the `producer` project and select Run.**

When the messages have been sent, the `SynchConsumer` client receives them and exits. The Output pane shows the output of both programs, in two different tabs.

5. **Now run the `Producer` example using a topic instead of a queue.**

   a. **Right-click the `producer` project and select Properties.**

   b. **Select Run from the Categories tree.**

   c. **In the Arguments field, type the following:**

      ```
      topic 3
      ```

   d. **Click OK.**

   e. **Right-click the project and select Run.**

      The output looks like this (along with some additional output):

      ```
      Destination type is topic
      Sending message: This is message 1 from producer
      Sending message: This is message 2 from producer
      Sending message: This is message 3 from producer
      ```

6. **Now run the `SynchConsumer` example using the topic.**

   a. **Right-click the `synchconsumer` project and select Properties.**

   b. **Select Run from the Categories tree.**

   c. **In the Arguments field, type the following:**

      ```
      topic
      ```

   d. **Click OK.**

   e. **Right-click the project and select Run.**

      The result, however, is different. Because you are using a topic, messages that were sent before you started the consumer cannot be received. (See "Publish/Subscribe Messaging Domain" on page 830 for details.) Instead of receiving the messages, the program appears to hang.

7. **Run the `Producer` example again. Right-click the `producer` project and select Run.**

   Now the `SynchConsumer` example receives the messages:

   ```
   Destination type is topic
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer
   ```

▼ To Build and Package the Clients for the Synchronous Receive Example Using Ant

1. **In a terminal window, go to the `producer` directory:**

   ```
   cd producer
   ```

2. **Type the following command:**

   ```
   ant
   ```

3. **In a terminal window, go to the `synchconsumer` directory:**

   ```
   cd ../synchconsumer
   ```

4. **Type the following command:**

   ```
   ant
   ```

   The targets place the application client JAR file in the `dist` directory for each example.

▼ To Deploy and Run the Clients for the Synchronous Receive Example Using Ant and the `appclient` Command

You can run the clients using the `appclient` command. The `build.xml` file for each project includes a target that creates resources, deploys the client, and then retrieves the client stubs that the `appclient` command uses. Each of the clients takes one or more command-line arguments: a destination type and, for `Producer`, a number of messages.

To build, deploy, and run the `Producer` and `SynchConsumer` examples using Ant and the `appclient` command, follow these steps.

To run the clients, you need two terminal windows.

1. **In a terminal window, go to the `producer` directory:**

   ```
   cd ../producer
   ```

2. **Create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:**

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.

3. **Run the `Producer` program, sending three messages to the queue:**

   ```
   appclient -client client-jar/producerClient.jar queue 3
   ```

   The output of the program looks like this (along with some additional output):

   ```
   Destination type is queue
   Sending message: This is message 1 from producer
   ```

```
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
```

The messages are now in the queue, waiting to be received.

Note – When you run an application client, the command often takes a long time to complete.

---

4. **In the same window, go to the `synchconsumer` directory:**

   ```
   cd ../synchconsumer
   ```

5. **Deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:**

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.

6. **Run the `SynchConsumer` client, specifying the queue:**

   ```
   appclient -client client-jar/synchconsumerClient.jar queue
   ```

   The output of the client looks like this (along with some additional output):

   ```
   Destination type is queue
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer
   ```

7. **Now try running the clients in the opposite order. Run the `SynchConsumer` client:**

   ```
   appclient -client client-jar/synchconsumerClient.jar queue
   ```

   The client displays the destination type and then appears to hang, waiting for messages.

8. **In a different terminal window, run the `Producer` client.**

   ```
   cd tut-install/examples/jms/simple/producer
   appclient -client client-jar/producerClient.jar queue 3
   ```

   When the messages have been sent, the `SynchConsumer` client receives them and exits.

9. **Now run the `Producer` client using a topic instead of a queue:**

   ```
   appclient -client client-jar/producerClient.jar topic 3
   ```

   The output of the client looks like this (along with some additional output):

   ```
   Destination type is topic
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer
   ```

10. **Now run the `SynchConsumer` client using the topic:**

    ```
    appclient -client client-jar/synchconsumerClient.jar topic
    ```

The result, however, is different. Because you are using a topic, messages that were sent before you started the consumer cannot be received. (See "Publish/Subscribe Messaging Domain" on page 830 for details.) Instead of receiving the messages, the client appears to hang.

11 Run the `Producer` client again.

Now the `SynchConsumer` client receives the messages (along with some additional output):

```
Destination type is topic
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
```

A Simple Example of Asynchronous Message Consumption

This section describes the receiving clients in an example that uses a message listener to consume messages asynchronously. This section then explains how to compile and run the clients using the GlassFish Server.

Writing the Clients for the Asynchronous Receive Example

The sending client is `producer/src/java/Producer.java`, the same client used in the example in "A Simple Example of Synchronous Message Receives" on page 860.

An asynchronous consumer normally runs indefinitely. This one runs until the user types the character `q` or `Q` to stop the client.

The receiving client, `asynchconsumer/src/java/AsynchConsumer.java`, performs the following steps:

1. Injects resources for a connection factory, queue, and topic.
2. Assigns either the queue or the topic to a destination object, based on the specified destination type.
3. Creates a `Connection` and a `Session`.
4. Creates a `MessageConsumer`.
5. Creates an instance of the `TextListener` class and registers it as the message listener for the `MessageConsumer`:

   ```
   listener = new TextListener();consumer.setMessageListener(listener);
   ```
6. Starts the connection, causing message delivery to begin.
7. Listens for the messages published to the destination, stopping when the user types the character `q` or `Q`:

   ```
   System.out.println("To end program, type Q or q, " + "then <return>");
   inputStreamReader = new InputStreamReader(System.in);
   while (!((answer == 'q') || (answer == 'Q'))) {
   ```

```
        try {
            answer = (char) inputStreamReader.read();
        } catch (IOException e) {
            System.out.println("I/O exception: " + e.toString());
        }
    }
```

8. Closes the connection, which automatically closes the session and `MessageConsumer`.

The message listener, `asynchconsumer/src/java/TextListener.java`, follows these steps:

1. When a message arrives, the `onMessage` method is called automatically.
2. The `onMessage` method converts the incoming message to a `TextMessage` and displays its content. If the message is not a text message, it reports this fact:

```
public void onMessage(Message message) {
    TextMessage msg = null;
    try {
        if (message instanceof TextMessage) {
            msg = (TextMessage) message;
             System.out.println("Reading message: " + msg.getText());
        } else {
             System.out.println("Message is not a " + "TextMessage");
        }
    } catch (JMSException e) {
        System.out.println("JMSException in onMessage(): " + e.toString());
    } catch (Throwable t) {
        System.out.println("Exception in onMessage():" + t.getMessage());
    }
}
```

For this example, you will use the connection factory and destinations you created for "A Simple Example of Synchronous Message Receives" on page 860.

▼ To Build and Package the `AsynchConsumer` Client Using NetBeans IDE

1 From the File menu, choose Open Project.

2 In the Open Project dialog, navigate to:

tut-install`/examples/jms/simple/`

3 Select the `asynchconsumer` folder.

4 Select the Open as Main Project check box.

5 Click Open Project.

6 In the Projects tab, right-click the project and select Build.

▼ To Deploy and Run the Clients for the Asynchronous Receive Example Using NetBeans IDE

1. **Run the `AsynchConsumer` example:**

   a. **Right-click the `asynchconsumer` project and select Properties.**

   b. **Select Run from the Categories tree.**

   c. **In the Arguments field, type the following:**

      ```
      topic
      ```

   d. **Click OK.**

   e. **Right-click the project and select Run.**

      The client displays the following lines and appears to hang:

      ```
      Destination type is topic
      To end program, type Q or q, then <return>
      ```

2. **Now run the `Producer` example:**

   a. **Right-click the `producer` project and select Properties.**

   b. **Select Run from the Categories tree.**

   c. **In the Arguments field, type the following:**

      ```
      topic 3
      ```

   d. **Click OK.**

   e. **Right-click the project and select Run.**

      The output of the client looks like this:

      ```
      Destination type is topic
      Sending message: This is message 1 from producer
      Sending message: This is message 2 from producer
      Sending message: This is message 3 from producer
      ```

      In the other tab, the `AsynchConsumer` client displays the following:

      ```
      Destination type is topic
      To end program, type Q or q, then <return>
      Reading message: This is message 1 from producer
      Reading message: This is message 2 from producer
      Reading message: This is message 3 from producer
      Message is not a TextMessage
      ```

The last line appears because the client has received the non-text control message sent by the `Producer` client.

3. **Type `Q` or `q` in the Output window and press Return to stop the client.**

4. **Now run the `Producer` client using a queue.**

   In this case, as with the synchronous example, you can run the `Producer` client first, because there is no timing dependency between the sender and the receiver.

   a. **Right-click the `producer` project and select Properties.**

   b. **Select Run from the Categories tree.**

   c. **In the Arguments field, type the following:**

      ```
      queue 3
      ```

   d. **Click OK.**

   e. **Right-click the project and select Run.**

      The output of the client looks like this:

      ```
      Destination type is queue
      Sending message: This is message 1 from producer
      Sending message: This is message 2 from producer
      Sending message: This is message 3 from producer
      ```

5. **Run the `AsynchConsumer` client.**

   a. **Right-click the `asynchconsumer` project and select Properties.**

   b. **Select Run from the Categories tree.**

   c. **In the Arguments field, type the following:**

      ```
      queue
      ```

   d. **Click OK.**

   e. **Right-click the project and select Run.**

      The output of the client looks like this:

      ```
      Destination type is queue
      To end program, type Q or q, then <return>
      Reading message: This is message 1 from producer
      Reading message: This is message 2 from producer
      Reading message: This is message 3 from producer
      Message is not a TextMessage
      ```

6. **Type `Q` or `q` in the Output window and press Return to stop the client.**

▼ To Build and Package the `AsynchConsumer` Client Using Ant

1. **In a terminal window, go to the `asynchconsumer` directory:**

   ```
   cd ../asynchconsumer
   ```

2. **Type the following command:**

   ```
   ant
   ```

   The targets package both the main class and the message listener class in the JAR file and place the file in the `dist` directory for the example.

▼ To Deploy and Run the Clients for the Asynchronous Receive Example Using Ant and the `appclient` Command

1. **Deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:**

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.

2. **Run the `AsynchConsumer` client, specifying the `topic` destination type.**

   ```
   appclient -client client-jar/asynchconsumerClient.jar topic
   ```

   The client displays the following lines (along with some additional output) and appears to hang:

   ```
   Destination type is topic
   To end program, type Q or q, then <return>
   ```

3. **In the terminal window where you ran the `Producer` client previously, run the client again, sending three messages.**

   ```
   appclient -client client-jar/producerClient.jar topic 3
   ```

   The output of the client looks like this (along with some additional output):

   ```
   Destination type is topic
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer
   ```

   In the other window, the `AsynchConsumer` client displays the following (along with some additional output):

   ```
   Destination type is topic
   To end program, type Q or q, then <return>
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer
   Message is not a TextMessage
   ```

The last line appears because the client has received the non-text control message sent by the `Producer` client.

4. **Type `Q` or `q` and press Return to stop the client.**

5. **Now run the clients using a queue.**

   In this case, as with the synchronous example, you can run the `Producer` client first, because there is no timing dependency between the sender and receiver:

   ```
   appclient -client client-jar/producerClient.jar queue 3
   ```

   The output of the client looks like this:

   ```
   Destination type is queue
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer
   ```

6. **Run the `AsynchConsumer` client:**

   ```
   appclient -client client-jar/asynchconsumerClient.jar queue
   ```

   The output of the client looks like this (along with some additional output):

   ```
   Destination type is queue
   To end program, type Q or q, then <return>
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer
   Message is not a TextMessage
   ```

7. **Type `Q` or `q` to stop the client.**

A Simple Example of Browsing Messages in a Queue

This section describes an example that creates a `QueueBrowser` object to examine messages on a queue, as described in "JMS Queue Browsers" on page 841. This section then explains how to compile, package, and run the example using the GlassFish Server.

Writing the Client for the `QueueBrowser` Example

To create a `QueueBrowser` for a queue, you call the `Session.createBrowser` method with the queue as the argument. You obtain the messages in the queue as an `Enumeration` object. You can then iterate through the `Enumeration` object and display the contents of each message.

The `messagebrowser/src/java/MessageBrowser.java` client performs the following steps:

1. Injects resources for a connection factory and a queue.
2. Creates a `Connection` and a `Session`.

3. Creates a `QueueBrowser`:

```
QueueBrowser browser = session.createBrowser(queue);
```

4. Retrieves the `Enumeration` that contains the messages:

```
Enumeration msgs = browser.getEnumeration();
```

5. Verifies that the `Enumeration` contains messages, then displays the contents of the messages:

```
if ( !msgs.hasMoreElements() ) {
    System.out.println("No messages in queue");
} else {
    while (msgs.hasMoreElements()) {
        Message tempMsg = (Message)msgs.nextElement();
        System.out.println("Message: " + tempMsg);
    }
}
```

6. Closes the connection, which automatically closes the session and the `QueueBrowser`.

The format in which the message contents appear is implementation-specific. In the GlassFish Server, the message format looks something like this:

```
Message contents:
Text: This is message 3 from producer
Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID(): ID:14-128.149.71.199(f9:86:a2:d5:46:9b)-40814-1255980521747
getJMSTimestamp(): 1129061034355
getJMSCorrelationID(): null
JMSReplyTo: null
JMSDestination: PhysicalQueue
getJMSDeliveryMode(): PERSISTENT
getJMSRedelivered(): false
getJMSType(): null
getJMSExpiration(): 0
getJMSPriority(): 4
Properties: null
```

For this example, you will use the connection factory and queue you created for "A Simple Example of Synchronous Message Receives" on page 860.

▼ To Run the `MessageBrowser` Client Using NetBeans IDE

To build, package, deploy, and run the `MessageBrowser` example using NetBeans IDE, follow these steps.

You also need the `Producer` example to send the message to the queue, and one of the consumer clients to consume the messages after you inspect them. If you did not do so already, package these examples.

1 From the File menu, choose Open Project.

2 In the Open Project dialog, navigate to:

tut-install`/examples/jms/simple/`

3. **Select the `messagebrowser` folder.**

4. **Select the Open as Main Project check box.**

5. **Click Open Project.**

6. **In the Projects tab, right-click the project and select Build.**

7. **Run the `Producer` client, sending one message to the queue:**

   a. **Right-click the `producer` project and select Properties.**

   b. **Select Run from the Categories tree.**

   c. **In the Arguments field, type the following:**

      ```
      queue
      ```

   d. **Click OK.**

   e. **Right-click the project and select Run.**

      The output of the client looks like this:

      ```
      Destination type is queue
      Sending message: This is message 1 from producer
      ```

8. **Run the `MessageBrowser` client. Right-click the `messagebrowser` project and select Run.**

   The output of the client looks something like this:

   ```
   Message:
   Text: This is message 1 from producer
   Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
   getJMSMessageID(): ID:12-128.149.71.199(8c:34:4a:1a:1b:b8)-40883-1255980521747
   getJMSTimestamp(): 1129062957611
   getJMSCorrelationID(): null
   JMSReplyTo: null
   JMSDestination: PhysicalQueue
   getJMSDeliveryMode(): PERSISTENT
   getJMSRedelivered(): false
   getJMSType(): null
   getJMSExpiration(): 0
   getJMSPriority(): 4
   Properties: null
   Message:
   Class: com.sun.messaging.jmq.jmsclient.MessageImpl
   getJMSMessageID(): ID:13-128.149.71.199(8c:34:4a:1a:1b:b8)-40883-1255980521747
   getJMSTimestamp(): 1129062957616
   getJMSCorrelationID(): null
   JMSReplyTo: null
   JMSDestination: PhysicalQueue
   getJMSDeliveryMode(): PERSISTENT
   getJMSRedelivered(): false
   ```

```
getJMSType(): null
getJMSExpiration(): 0
getJMSPriority(): 4
Properties: null
```

The first message is the `TextMessage`, and the second is the non-text control message.

9 Run the `SynchConsumer` client to consume the messages.

a. Right-click the `synchconsumer` project and select Properties.

b. Select Run from the Categories tree.

c. In the Arguments field, type the following:

```
queue
```

d. Click OK.

e. Right-click the project and select Run.

The output of the client looks like this:

```
Destination type is queue
Reading message: This is message 1 from producer
```

▼ To Run the `MessageBrowser` Client Using Ant and the `appclient` Command

To build, package, deploy, and run the `MessageBrowser` example using Ant, follow these steps.

You also need the `Producer` example to send the message to the queue, and one of the consumer clients to consume the messages after you inspect them. If you did not do so already, package these examples.

To run the clients, you need two terminal windows.

1 In a terminal window, go to the `messagebrowser` directory.

```
cd ../messagebrowser
```

2 Type the following command:

```
ant
```

The targets place the application client JAR file in the `dist` directory for the example.

3 In another terminal window, go to the `producer` directory.

4 Run the `Producer` client, sending one message to the queue:

```
appclient -client client-jar/producerClient.jar queue
```

The output of the client looks like this (along with some additional output):

```
Destination type is queue
Sending message: This is message 1 from producer
```

5 **Go to the `messagebrowser` directory.**

6 **Deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:**

```
ant getclient
```

Ignore the message that states that the application is deployed at a URL.

7 **Because this example takes no command-line arguments, you can run the `MessageBrowser` client using the following command:**

```
ant run
```

Alternatively, you can type the following command:

```
appclient -client client-jar/messagebrowserClient.jar
```

The output of the client looks something like this (along with some additional output):

```
Message:
Text: This is message 1 from producer
Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID(): ID:12-128.149.71.199(8c:34:4a:1a:1b:b8)-40883-1255980521747
getJMSTimestamp(): 1255980521747
getJMSCorrelationID(): null
JMSReplyTo: null
JMSDestination: PhysicalQueue
getJMSDeliveryMode(): PERSISTENT
getJMSRedelivered(): false
getJMSType(): null
getJMSExpiration(): 0
getJMSPriority(): 4
Properties: null
Message:
Class: com.sun.messaging.jmq.jmsclient.MessageImpl
getJMSMessageID(): ID:13-128.149.71.199(8c:34:4a:1a:1b:b8)-40883-1255980521767
getJMSTimestamp(): 1255980521767
getJMSCorrelationID(): null
JMSReplyTo: null
JMSDestination: PhysicalQueue
getJMSDeliveryMode(): PERSISTENT
getJMSRedelivered(): false
getJMSType(): null
getJMSExpiration(): 0
getJMSPriority(): 4
Properties: null
```

The first message is the `TextMessage`, and the second is the non-text control message.

8 **Go to the `synchconsumer` directory.**

9 **Run the `SynchConsumer` client to consume the messages:**

```
appclient -client client-jar/synchconsumerClient.jar queue
```

The output of the client looks like this (along with some additional output):

```
Destination type is queue
Reading message: This is message 1 from producer
```

Running JMS Clients on Multiple Systems

JMS clients that use the GlassFish Server can exchange messages with each other when they are running on different systems in a network. The systems must be visible to each other by name (the UNIX host name or the Microsoft Windows computer name) and must both be running the GlassFish Server.

Note – Any mechanism for exchanging messages between systems is specific to the Java EE server implementation. This tutorial describes how to use the GlassFish Server for this purpose.

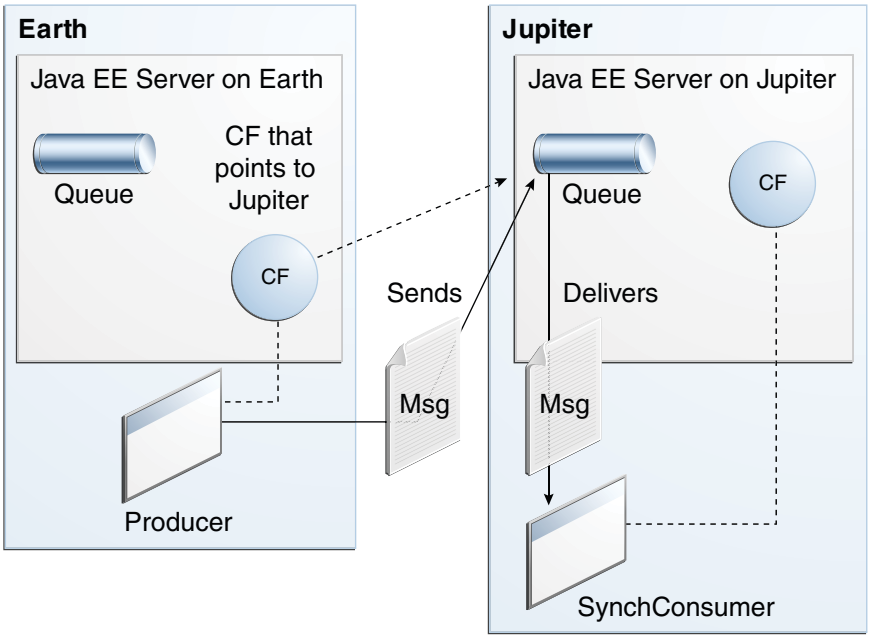
Suppose you want to run the `Producer` client on one system, `earth`, and the `SynchConsumer` client on another system, `jupiter`. Before you can do so, you need to perform these tasks:

1. Create two new connection factories
2. Change the name of the default JMS host on one system
3. Edit the source code for the two examples
4. Recompile and repackage the examples

Note – A limitation in the JMS provider in the GlassFish Server may cause a runtime failure to create a connection to systems that use the Dynamic Host Configuration Protocol (DHCP) to obtain an IP address. You can, however, create a connection *from* a system that uses DHCP *to* a system that does not use DHCP. In the examples in this tutorial, `earth` can be a system that uses DHCP, and `jupiter` can be a system that does not use DHCP.

When you run the clients, they will work as shown in Figure 48–1. The client run on `earth` needs the queue on `earth` only so the resource injection will succeed. The connection, session, and message producer are all created on `jupiter` using the connection factory that points to `jupiter`. The messages sent from `earth` will be received on `jupiter`.

FIGURE 48–1 Sending Messages from One System to Another



For examples showing how to deploy more complex applications on two different systems, see "An Application Example That Consumes Messages from a Remote Server" on page 910 and "An Application Example That Deploys a Message-Driven Bean on Two Servers" on page 916.

▼ To Create Administered Objects for Multiple Systems

To run these clients, you must do the following:

- Create a new connection factory on both `earth` and `jupiter`
- Create a destination resource on both `earth` and `jupiter`

You do not have to install the tutorial examples on both systems, but you must be able to access the file system where it is installed. You may find it more convenient to install the tutorial examples on both systems if the two use different operating systems (for example, Windows and UNIX). Otherwise you will have to edit the *tut-install*`/examples/bp-project/build.properties` file and change the location of the `javaee.home` property each time you build or run a client on a different system.

1. **Start the GlassFish Server on `earth`.**
2. **Start the GlassFish Server on `jupiter`.**

3. **To create a new connection factory on `jupiter`, follow these steps:**

   a. **From a command shell on `jupiter`, go to the *tut-install*`/examples/jms/simple/producer/` directory.**

   b. **Type the following command:**

      ```
      ant create-local-factory
      ```

      The `create-local-factory` target, defined in the `build.xml` file for the `Producer` example, creates a connection factory named `jms/JupiterConnectionFactory`.

4. **To create a new connection factory on `earth` that points to the connection factory on `jupiter`, follow these steps:**

   a. **From a command shell on `earth`, go to the *tut-install*`/examples/jms/simple/producer/` directory.**

   b. **Type the following command:**

      ```
      ant create-remote-factory -Dsys=remote-system-name
      ```

      Replace *remote-system-name* with the actual name of the remote system.

      The `create-remote-factory` target, defined in the `build.xml` file for the `Producer` example, also creates a connection factory named `jms/JupiterConnectionFactory`. In addition, it sets the `AddressList` property for this factory to the name of the remote system.

Additional resources will be created when you deploy the application, if they have not been created before.

The reason the `glassfish-resources.xml` file does not specify `jms/JupiterConnectionFactory` is that on `earth` the connection factory requires the `AddressList` property setting, whereas on `jupiter` it does not. You can examine the targets in the `build.xml` file for details.

Changing the Default Host Name

By default, the default host name for the JMS service on the GlassFish Server is `localhost`. To access the JMS service from another system, however, you must change the host name. You can change it either to the actual host name or to `0.0.0.0`.

You can change the default host name using either the Administration Console or the `asadmin` command.

▼ To Change the Default Host Name Using the Administration Console

1. **On `jupiter`, start the Administration Console by opening a browser at `http://localhost:4848/`.**

2. **In the navigation tree, expand the Configurations node, then expand the server-config node.**
3. **Under the server-config node, expand the Java Message Service node.**
4. **Under the Java Message Service node, expand the JMS Hosts node.**
5. **Under the JMS Hosts node, select `default_JMS_host`.**
   The Edit JMS Host page opens.
6. **In the Host field, type the name of the system, or type `0.0.0.0`.**
7. **Click Save.**
8. **Restart the GlassFish Server.**

▼ To Change the Default Host Name Using the `asadmin` Command

1. **Specify a command like one of the following:**

   ```
   asadmin set server-config.jms-service.jms-host.default_JMS_host.host="0.0.0.0"
   ```

   ```
   asadmin set server-config.jms-service.jms-host.default_JMS_host.host="hostname"
   ```

2. **Restart the GlassFish Server.**

▼ To Run the Clients Using NetBeans IDE

These steps assume you have the tutorial installed on both of the systems you are using and that you are able to access the file system of `jupiter` from `earth` or vice versa. You will edit the source files to specify the new connection factory. Then you will rebuild and run the clients.

1. **To edit the source files, follow these steps:**

   a. **On `earth`, open the following file in NetBeans IDE:**
      tut-install`/examples/jms/simple/producer/src/java/Producer.java`

   b. **Find the following line:**

      ```
      @Resource(lookup = "jms/ConnectionFactory")
      ```

   c. **Change the line to the following:**

      ```
      @Resource(lookup = "jms/JupiterConnectionFactory")
      ```

   d. **Save the file.**

e. **On `jupiter`, open the following file in NetBeans IDE:**

   tut-install`/examples/jms/simple/synchconsumer/src/java/SynchConsumer.java`

f. **Repeat Step b and Step c, then save the file.**

2 **To recompile and repackage the `Producer` example on `earth`, right-click the `producer` project and select Clean and Build.**

3 **To recompile and repackage the `SynchConsumer` example on `jupiter`, right-click the `synchconsumer` project and select Clean and Build.**

4 **On `earth`, deploy and run `Producer`. Follow these steps:**

a. **Right-click the `producer` project and select Properties.**

b. **Select Run from the Categories tree.**

c. **In the Arguments field, type the following:**

   `queue 3`

d. **Click OK.**

e. **Right-click the project and select Run.**

   The output looks like this (along with some additional output):

```
Destination type is topic
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
```

5 **On `jupiter`, run `SynchConsumer`. Follow these steps:**

a. **Right-click the `synchconsumer` project and select Properties.**

b. **Select Run from the Categories tree.**

c. **In the Arguments field, type the following:**

   `queue`

d. **Click OK.**

e. **Right-click the project and select Run.**

   The output of the program looks like this (along with some additional output):

```
Destination type is queue
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
```

```
Reading message: This is message 3 from producer
```

▼ To Run the Clients Using Ant and the `appclient` Command

These steps assume you have the tutorial installed on both of the systems you are using and that you are able to access the file system of `jupiter` from `earth` or vice versa. You will edit the source files to specify the new connection factory. Then you will rebuild and run the clients.

1. **To edit the source files, follow these steps:**

   a. **On `earth`, open the following file in a text editor:**

      tut-install`/examples/jms/simple/producer/src/java/Producer.java`

   b. **Find the following line:**

      ```
      @Resource(lookup = "jms/ConnectionFactory")
      ```

   c. **Change the line to the following:**

      ```
      @Resource(lookup = "jms/JupiterConnectionFactory")
      ```

   d. **Save and close the file.**

   e. **On `jupiter`, open the following file in a text editor:**

      tut-install`/examples/jms/simple/synchconsumer/src/java/SynchConsumer.java`

   f. **Repeat Step b and Step c, then save and close the file.**

2. **To recompile and repackage the `Producer` example on `earth`, type the following:**

   ```
   ant
   ```

3. **To recompile and repackage the `SynchConsumer` example on `jupiter`, go to the `synchconsumer` directory and type the following:**

   ```
   ant
   ```

4. **On `earth`, deploy and run `Producer`. Follow these steps:**

   a. **On `earth`, from the `producer` directory, create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:**

      ```
      ant getclient
      ```

      Ignore the message that states that the application is deployed at a URL.

   b. **To run the client, type the following:**

      ```
      appclient -client client-jar/producerClient.jar queue 3
      ```

The output looks like this (along with some additional output):

```
Destination type is topic
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
```

5 **On `jupiter`, run `SynchConsumer`. Follow these steps:**

a. **From the `synchconsumer` directory, create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:**

```
ant getclient
```

Ignore the message that states that the application is deployed at a URL.

b. **To run the client, type the following:**

```
appclient -client client-jar/synchconsumerClient.jar queue
```

The output of the program looks like this (along with some additional output):

```
Destination type is queue
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
```

Undeploying and Cleaning the Simple JMS Examples

After you finish running the examples, you can undeploy them and remove the build artifacts.

You can also use the `asadmin delete-jms-resource` command to delete the destinations and connection factories you created. However, it is recommended that you keep them, because they will be used in most of the examples later in this chapter. After you have created them, they will be available whenever you restart the GlassFish Server.

Writing Robust JMS Applications

The following examples show how to use some of the more advanced features of the JMS API.

A Message Acknowledgment Example

The `AckEquivExample.java` client shows how both of the following scenarios ensure that a message will not be acknowledged until processing of it is complete:

- Using an asynchronous message consumer (a message listener) in an `AUTO_ACKNOWLEDGE` session
- Using a synchronous receiver in a `CLIENT_ACKNOWLEDGE` session

Note – In the Java EE platform, message listeners and `CLIENT_ACKNOWLEDGE` sessions can be used only in application clients, as in this example.

With a message listener, the automatic acknowledgment happens when the `onMessage` method returns (that is, after message processing has finished). With a synchronous receiver, the client acknowledges the message after processing is complete. If you use `AUTO_ACKNOWLEDGE` with a synchronous receive, the acknowledgment happens immediately after the `receive` call; if any subsequent processing steps fail, the message cannot be redelivered.

The example is in the following directory:

tut-install`/examples/jms/advanced/ackequivexample/src/java/`

The example contains an `AsynchSubscriber` class with a `TextListener` class, a `MultiplePublisher` class, a `SynchReceiver` class, a `SynchSender` class, a `main` method, and a method that runs the other classes' threads.

The example uses the following objects:

- `jms/ConnectionFactory`, `jms/Queue`, and `jms/Topic`: resources that you created for "A Simple Example of Synchronous Message Receives" on page 860.
- `jms/ControlQueue`: an additional queue
- `jms/DurableConnectionFactory`: a connection factory with a client ID (see "Creating Durable Subscriptions" on page 847 for more information)

The new queue and connection factory are created at deployment time.

You can use either NetBeans IDE or Ant to build, package, deploy, and run `ackequivexample`.

▼ To Run `ackequivexample` Using NetBeans IDE

1 To build and package the client, follow these steps.

a. From the File menu, choose Open Project.

b. **In the Open Project dialog, navigate to:**

tut-install`/examples/jms/advanced/`

c. **Select the `ackequivexample` folder.**

d. **Select the Open as Main Project check box.**

e. **Click Open Project.**

f. **In the Projects tab, right-click the project and select Build.**

2 **To run the client, right-click the `ackequivexample` project and select Run.**

The client output looks something like this (along with some additional output):

```
Queue name is jms/ControlQueue
Queue name is jms/Queue
Topic name is jms/Topic
Connection factory name is jms/DurableConnectionFactory
  SENDER: Created client-acknowledge session
  SENDER: Sending message: Here is a client-acknowledge message
  RECEIVER: Created client-acknowledge session
  RECEIVER: Processing message: Here is a client-acknowledge message
  RECEIVER: Now I'll acknowledge the message
SUBSCRIBER: Created auto-acknowledge session
SUBSCRIBER: Sending synchronize message to control queue
PUBLISHER: Created auto-acknowledge session
PUBLISHER: Receiving synchronize messages from control queue; count = 1
PUBLISHER: Received synchronize message;  expect 0 more
PUBLISHER: Publishing message: Here is an auto-acknowledge message 1
PUBLISHER: Publishing message: Here is an auto-acknowledge message 2
SUBSCRIBER: Processing message: Here is an auto-acknowledge message 1
PUBLISHER: Publishing message: Here is an auto-acknowledge message 3
SUBSCRIBER: Processing message: Here is an auto-acknowledge message 2
SUBSCRIBER: Processing message: Here is an auto-acknowledge message 3
```

3 **After you run the client, you can delete the destination resource `jms/ControlQueue` by using the following command:**

```
asadmin delete-jms-resource jms/ControlQueue
```

You will need the other resources for other examples.

▼ To Run `ackequivexample` Using Ant

1 **In a terminal window, go to the following directory:**

tut-install`/examples/jms/advanced/ackequivexample/`

2 **To compile and package the client, type the following command:**

```
ant
```

3. **To create needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs, type the following command:**

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.

4. **Because this example takes no command-line arguments, you can run the client using the following command:**

   ```
   ant run
   ```

   Alternatively, you can type the following command:

   ```
   appclient -client client-jar/ackequivexampleClient.jar
   ```

   The client output looks something like this (along with some additional output):

   ```
   Queue name is jms/ControlQueue
   Queue name is jms/Queue
   Topic name is jms/Topic
   Connection factory name is jms/DurableConnectionFactory
     SENDER: Created client-acknowledge session
     SENDER: Sending message: Here is a client-acknowledge message
     RECEIVER: Created client-acknowledge session
     RECEIVER: Processing message: Here is a client-acknowledge message
     RECEIVER: Now I'll acknowledge the message
   SUBSCRIBER: Created auto-acknowledge session
   SUBSCRIBER: Sending synchronize message to control queue
   PUBLISHER: Created auto-acknowledge session
   PUBLISHER: Receiving synchronize messages from control queue; count = 1
   PUBLISHER: Received synchronize message;  expect 0 more
   PUBLISHER: Publishing message: Here is an auto-acknowledge message 1
   PUBLISHER: Publishing message: Here is an auto-acknowledge message 2
   SUBSCRIBER: Processing message: Here is an auto-acknowledge message 1
   PUBLISHER: Publishing message: Here is an auto-acknowledge message 3
   SUBSCRIBER: Processing message: Here is an auto-acknowledge message 2
   SUBSCRIBER: Processing message: Here is an auto-acknowledge message 3
   ```

5. **After you run the client, you can delete the destination resource `jms/ControlQueue` by using the following command:**

   ```
   asadmin delete-jms-resource jms/ControlQueue
   ```

   You will need the other resources for other examples.

A Durable Subscription Example

`DurableSubscriberExample.java` shows how durable subscriptions work. It demonstrates that a durable subscription is active even when the subscriber is not active. The example contains a `DurableSubscriber` class, a `MultiplePublisher` class, a `main` method, and a method that instantiates the classes and calls their methods in sequence.

The example is in the *tut-install*`/examples/jms/advanced/durablesubscriberexample/src/java/` directory.

The example begins in the same way as any publish/subscribe client: The subscriber starts, the publisher publishes some messages, and the subscriber receives them. At this point, the subscriber closes itself. The publisher then publishes some messages while the subscriber is not active. The subscriber then restarts and receives those messages.

You can use either NetBeans IDE or Ant to build, package, deploy, and run `durablesubscriberexample`.

▼ To Run `durablesubscriberexample` Using NetBeans IDE

1 To compile and package the client, follow these steps:

a. From the File menu, choose Open Project.

b. In the Open Project dialog, navigate to:

tut-install`/examples/jms/advanced/`

c. Select the `durablesubscriberexample` folder.

d. Select the Open as Main Project check box.

e. Click Open Project.

f. In the Projects tab, right-click the project and select Build.

2 To run the client, right-click the `durablesubscriberexample` project and select Run.

The output looks something like this (along with some additional output):

```
Connection factory without client ID is jms/ConnectionFactory
Connection factory with client ID is jms/DurableConnectionFactory
Topic name is jms/Topic
Starting subscriber
PUBLISHER: Publishing message: Here is a message 1
SUBSCRIBER: Reading message: Here is a message 1
PUBLISHER: Publishing message: Here is a message 2
SUBSCRIBER: Reading message: Here is a message 2
PUBLISHER: Publishing message: Here is a message 3
SUBSCRIBER: Reading message: Here is a message 3
Closing subscriber
PUBLISHER: Publishing message: Here is a message 4
PUBLISHER: Publishing message: Here is a message 5
PUBLISHER: Publishing message: Here is a message 6
Starting subscriber
SUBSCRIBER: Reading message: Here is a message 4
SUBSCRIBER: Reading message: Here is a message 5
SUBSCRIBER: Reading message: Here is a message 6
```

```
Closing subscriber
Unsubscribing from durable subscription
```

3. **After you run the client, you can delete the connection factory `jms/DurableConnectionFactory` by using the following command:**

   ```
   asadmin delete-jms-resource jms/DurableConnectionFactory
   ```

▼ To Run `durablesubscriberexample` Using Ant

1. **In a terminal window, go to the following directory:**

   tut-install`/examples/jms/advanced/durablesubscriberexample/`

2. **To compile and package the client, type the following command:**

   ```
   ant
   ```

3. **To create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs, type the following command:**

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.

4. **Because this example takes no command-line arguments, you can run the client using the following command:**

   ```
   ant run
   ```

   Alternatively, you can type the following command:

   ```
   appclient -client client-jar/durablesubscriberexampleClient.jar
   ```

5. **After you run the client, you can delete the connection factory `jms/DurableConnectionFactory` by using the following command:**

   ```
   asadmin delete-jms-resource jms/DurableConnectionFactory
   ```

A Local Transaction Example

`TransactedExample.java` demonstrates the use of transactions in a JMS client application. The example is in the *tut-install*`/examples/jms/advanced/transactedexample/src/java/` directory.

This example shows how to use a queue and a topic in a single transaction as well as how to pass a session to a message listener's constructor function. The example represents a highly simplified e-commerce application in which the following actions occur.

1. A retailer sends a `MapMessage` to the vendor order queue, ordering a quantity of computers, and waits for the vendor's reply:

```
producer = session.createProducer(vendorOrderQueue);
outMessage = session.createMapMessage();
outMessage.setString("Item", "Computer(s)");
outMessage.setInt("Quantity", quantity);
outMessage.setJMSReplyTo(retailerConfirmQueue);
producer.send(outMessage);
System.out.println("Retailer: ordered " + quantity + " computer(s)");
orderConfirmReceiver = session.createConsumer(retailerConfirmQueue);
connection.start();
```

2. The vendor receives the retailer's order message and sends an order message to the supplier order topic in one transaction. This JMS transaction uses a single session, so you can combine a receive from a queue with a send to a topic. Here is the code that uses the same session to create a consumer for a queue and a producer for a topic:

   ```
   vendorOrderReceiver = session.createConsumer(vendorOrderQueue);
   supplierOrderProducer = session.createProducer(supplierOrderTopic);
   ```

   The following code receives the incoming message, sends an outgoing message, and commits the session. The message processing has been removed to keep the sequence simple:

   ```
   inMessage = vendorOrderReceiver.receive();
   // Process the incoming message and format the outgoing
   // message
   ...
   supplierOrderProducer.send(orderMessage);
   ...
   session.commit();
   ```

   For simplicity, there are only two suppliers, one for CPUs and one for hard drives.

3. Each supplier receives the order from the order topic, checks its inventory, and then sends the items ordered to the queue named in the order message's `JMSReplyTo` field. If it does not have enough of the item in stock, the supplier sends what it has. The synchronous receive from the topic and the send to the queue take place in one JMS transaction.

   ```
   receiver = session.createConsumer(orderTopic);
   ...
   inMessage = receiver.receive();
   if (inMessage instanceof MapMessage) {
       orderMessage = (MapMessage) inMessage;
   }
   // Process message
   MessageProducer producer =
       session.createProducer((Queue) orderMessage.getJMSReplyTo());
   outMessage = session.createMapMessage();
   // Add content to message
   producer.send(outMessage);
   // Display message contentssession.commit();
   ```

4. The vendor receives the suppliers' replies from its confirmation queue and updates the state of the order. Messages are processed by an asynchronous message listener; this step shows the use of JMS transactions with a message listener.

   ```
   MapMessage component = (MapMessage) message;
   ...
   ```

```
orderNumber = component.getInt("VendorOrderNumber");
Order order = Order.getOrder(orderNumber).processSubOrder(component);
session.commit();
```

5. When all outstanding replies are processed for a given order, the vendor message listener sends a message notifying the retailer whether it can fulfill the order.

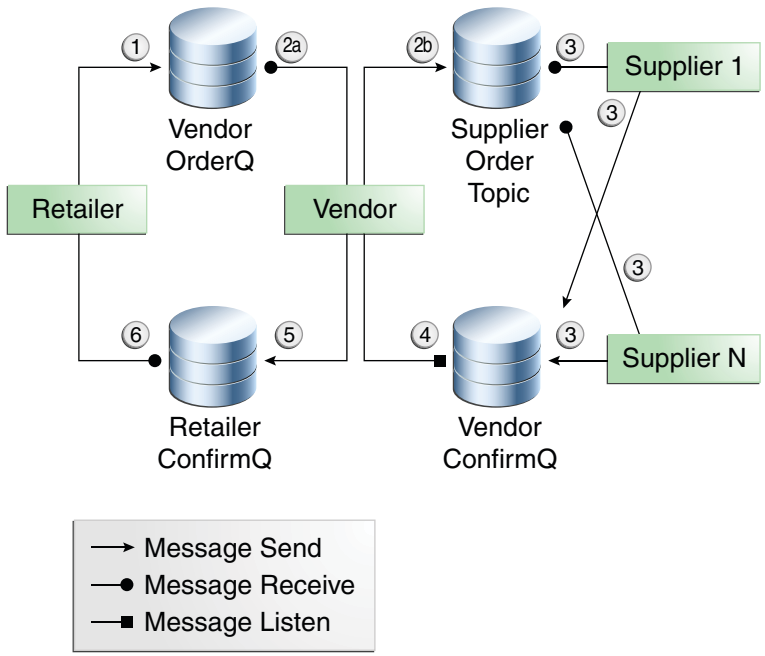
```
Queue replyQueue = (Queue) order.order.getJMSReplyTo();
MessageProducer producer = session.createProducer(replyQueue);
MapMessage retailerConfirmMessage = session.createMapMessage();
// Format the message
producer.send(retailerConfirmMessage);
session.commit();
```

6. The retailer receives the message from the vendor:

```
inMessage = (MapMessage) orderConfirmReceiver.receive();
```

Figure 48–2 illustrates these steps.

FIGURE 48–2 Transactions: JMS Client Example



The example contains five classes: `GenericSupplier`, `Order`, `Retailer`, `Vendor`, and `VendorMessageListener`. The example also contains a `main` method and a method that runs the threads of the `Retailer`, `Vendor`, and two supplier classes.

All the messages use the `MapMessage` message type. Synchronous receives are used for all message reception except when the vendor processes the replies of the suppliers. These replies are processed asynchronously and demonstrate how to use transactions within a message listener.

At random intervals, the `Vendor` class throws an exception to simulate a database problem and cause a rollback.

All classes except `Retailer` use transacted sessions.

The example uses three queues named `jms/AQueue`, `jms/BQueue`, and `jms/CQueue`, and one topic named `jms/OTopic`.

You can use either NetBeans IDE or Ant to build, package, deploy, and run `transactedexample`.

▼ To Run `transactedexample` Using NetBeans IDE

1. **In a terminal window, go to the following directory:**
   tut-install`/examples/jms/advanced/transactedexample/`
2. **To compile and package the client, follow these steps:**
   a. **From the File menu, choose Open Project.**
   b. **In the Open Project dialog, navigate to:**
      tut-install`/examples/jms/advanced/`
   c. **Select the `transactedexample` folder.**
   d. **Select the Open as Main Project check box.**
   e. **Click Open Project.**
   f. **In the Projects tab, right-click the project and select Build.**
3. **To deploy and run the client, follow these steps:**
   a. **Right-click the `transactedexample` project and select Properties.**
   b. **Select Run from the Categories tree.**
   c. **In the Arguments field, type a number that specifies the number of computers to order:**
      3

d. Click OK.

e. Right-click the project and select Run.

The output looks something like this (along with some additional output):

```
Quantity to be ordered is 3
Retailer: ordered 3 computer(s)
Vendor: Retailer ordered 3 Computer(s)
Vendor: ordered 3 CPU(s) and hard drive(s)
CPU Supplier: Vendor ordered 3 CPU(s)
CPU Supplier: sent 3 CPU(s)
  CPU Supplier: committed transaction
  Vendor: committed transaction 1
Hard Drive Supplier: Vendor ordered 3 Hard Drive(s)
Hard Drive Supplier: sent 1 Hard Drive(s)
Vendor: Completed processing for order 1
  Hard Drive Supplier: committed transaction
Vendor: unable to send 3 computer(s)
  Vendor: committed transaction 2
Retailer: Order not filled
Retailer: placing another order
Retailer: ordered 6 computer(s)
Vendor: JMSException occurred: javax.jms.JMSException:
Simulated database concurrent access exception
javax.jms.JMSException: Simulated database concurrent access exception
        at TransactedExample$Vendor.run(Unknown Source)
  Vendor: rolled back transaction 1
Vendor: Retailer ordered 6 Computer(s)
Vendor: ordered 6 CPU(s) and hard drive(s)
CPU Supplier: Vendor ordered 6 CPU(s)
Hard Drive Supplier: Vendor ordered 6 Hard Drive(s)
CPU Supplier: sent 6 CPU(s)
  CPU Supplier: committed transaction
Hard Drive Supplier: sent 6 Hard Drive(s)
  Hard Drive Supplier: committed transaction
  Vendor: committed transaction 1
Vendor: Completed processing for order 2
Vendor: sent 6 computer(s)
Retailer: Order filled
  Vendor: committed transaction 2
```

4 After you run the client, you can delete the destination resources in NetBeans IDE or by using the following commands:

```
asadmin delete-jms-resource jms/AQueue
asadmin delete-jms-resource jms/BQueue
asadmin delete-jms-resource jms/CQueue
asadmin delete-jms-resource jms/OTopic
```

▼ To Run `transactedexample` Using Ant and the `appclient` Command

1 In a terminal window, go to the following directory:

tut-install`/examples/jms/advanced/transactedexample/`

2. **To build and package the client, type the following command:**

   ```
   ant
   ```

3. **Create needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:**

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.

4. **Use a command like the following to run the client.**

   The argument specifies the number of computers to order.

   ```
   appclient -client client-jar/transactedexampleClient.jar 3
   ```

   The output looks something like this (along with some additional output):

```
Quantity to be ordered is 3
Retailer: ordered 3 computer(s)
Vendor: Retailer ordered 3 Computer(s)
Vendor: ordered 3 CPU(s) and hard drive(s)
CPU Supplier: Vendor ordered 3 CPU(s)
CPU Supplier: sent 3 CPU(s)
  CPU Supplier: committed transaction
  Vendor: committed transaction 1
Hard Drive Supplier: Vendor ordered 3 Hard Drive(s)
Hard Drive Supplier: sent 1 Hard Drive(s)
Vendor: Completed processing for order 1
  Hard Drive Supplier: committed transaction
Vendor: unable to send 3 computer(s)
  Vendor: committed transaction 2
Retailer: Order not filled
Retailer: placing another order
Retailer: ordered 6 computer(s)
Vendor: JMSException occurred: javax.jms.JMSException:
Simulated database concurrent access exception
javax.jms.JMSException: Simulated database concurrent access exception
        at TransactedExample$Vendor.run(Unknown Source)
  Vendor: rolled back transaction 1
Vendor: Retailer ordered 6 Computer(s)
Vendor: ordered 6 CPU(s) and hard drive(s)
CPU Supplier: Vendor ordered 6 CPU(s)
Hard Drive Supplier: Vendor ordered 6 Hard Drive(s)
CPU Supplier: sent 6 CPU(s)
  CPU Supplier: committed transaction
Hard Drive Supplier: sent 6 Hard Drive(s)
  Hard Drive Supplier: committed transaction
  Vendor: committed transaction 1
Vendor: Completed processing for order 2
Vendor: sent 6 computer(s)
Retailer: Order filled
  Vendor: committed transaction 2
```

5 **After you run the client, you can delete the destination resources by using the following commands:**

```
asadmin delete-jms-resource jms/AQueue
asadmin delete-jms-resource jms/BQueue
asadmin delete-jms-resource jms/CQueue
asadmin delete-jms-resource jms/OTopic
```

An Application That Uses the JMS API with a Session Bean

This section explains how to write, compile, package, deploy, and run an application that uses the JMS API in conjunction with a session bean. The application contains the following components:

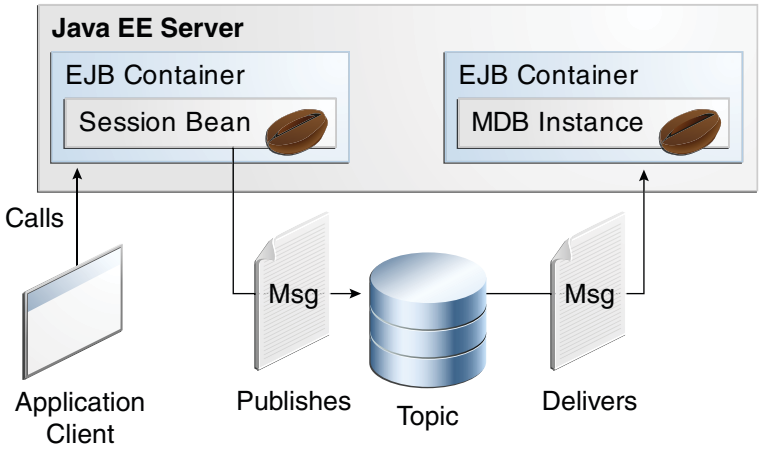
- An application client that invokes a session bean
- A session bean that publishes several messages to a topic
- A message-driven bean that receives and processes the messages using a durable topic subscriber and a message selector

You will find the source files for this section in the *tut-install*`/examples/jms/clientsessionmdb/` directory. Path names in this section are relative to this directory.

Writing the Application Components for the `clientsessionmdb` Example

This application demonstrates how to send messages from an enterprise bean (in this case, a session bean) rather than from an application client, as in the example in Chapter 25, "A Message-Driven Bean Example." Figure 48–3 illustrates the structure of this application.

FIGURE 48–3 An Enterprise Bean Application: Client to Session Bean to Message-Driven Bean



The Publisher enterprise bean in this example is the enterprise-application equivalent of a wire-service news feed that categorizes news events into six news categories. The message-driven bean could represent a newsroom, where the sports desk, for example, would set up a subscription for all news events pertaining to sports.

The application client in the example injects the Publisher enterprise bean's remote home interface and then calls the bean's business method. The enterprise bean creates 18 text messages. For each message, it sets a `String` property randomly to one of six values representing the news categories and then publishes the message to a topic. The message-driven bean uses a message selector for the property to limit which of the published messages it receives.

Coding the Application Client: `MyAppClient.java`

The application client, `clientsessionmdb-app-client/src/java/MyAppClient.java`, performs no JMS API operations and so is simpler than the client in Chapter 25, "A Message-Driven Bean Example." The client uses dependency injection to obtain the Publisher enterprise bean's business interface:

```
@EJB(name="PublisherRemote")
static private PublisherRemote publisher;
```

The client then calls the bean's business method twice.

Coding the Publisher Session Bean

The Publisher bean is a stateless session bean that has one business method. The Publisher bean uses a remote interface rather than a local interface because it is accessed from the application client.

The remote interface, `clientsessionmdb-ejb/src/java/sb/PublisherRemote.java`, declares a single business method, `publishNews`.

The bean class, `clientsessionmdb-ejb/src/java/sb/PublisherBean.java`, implements the `publishNews` method and its helper method `chooseType`. The bean class also injects `SessionContext`, `ConnectionFactory`, and `Topic` resources and implements `@PostConstruct` and `@PreDestroy` callback methods. The bean class begins as follows:

```
@Stateless
@Remote({PublisherRemote.class})
public class PublisherBean implements PublisherRemote {

    @Resource
    private SessionContext sc;

    @Resource(lookup = "jms/ConnectionFactory")
    private ConnectionFactory connectionFactory;

    @Resource(lookup = "jms/Topic")
    private Topic topic;
    ...
```

The `@PostConstruct` callback method of the bean class, `makeConnection`, creates the `Connection` used by the bean. The business method `publishNews` creates a `Session` and a `MessageProducer` and publishes the messages.

The `@PreDestroy` callback method, `endConnection`, deallocates the resources that were allocated by the `@PostConstruct` callback method. In this case, the method closes the `Connection`.

Coding the Message-Driven Bean: `MessageBean.java`

The message-driven bean class, `clientsessionmdb-ejb/src/java/mdb/MessageBean.java`, is almost identical to the one in Chapter 25, "A Message-Driven Bean Example." However, the `@MessageDriven` annotation is different, because instead of a queue, the bean is using a topic with a durable subscription, and it is also using a message selector. Therefore, the annotation sets the activation config properties `messageSelector`, `subscriptionDurability`, `clientId`, and `subscriptionName`, as follows:

```
@MessageDriven(mappedName = "jms/Topic", activationConfig =  {
    @ActivationConfigProperty(propertyName = "messageSelector",
            propertyValue = "NewsType = 'Sports' OR NewsType = 'Opinion'")
    , @ActivationConfigProperty(propertyName = "subscriptionDurability",
            propertyValue = "Durable")
    , @ActivationConfigProperty(propertyName = "clientId",
            propertyValue = "MyID")
    , @ActivationConfigProperty(propertyName = "subscriptionName",
            propertyValue = "MySub")
    })
```

Note – For a message-driven bean, the destination is specified with the `mappedName` element instead of the `lookup` element.

The JMS resource adapter uses these properties to create a connection factory for the message-driven bean that allows the bean to use a durable subscriber.

Creating Resources for the `clientsessionmdb` Example

This example uses the topic named `jms/Topic` and the connection factory `jms/ConnectionFactory`, which are also used in previous examples. If you deleted the connection factory or topic, they will be recreated when you deploy the example.

Running the `clientsessionmdb` Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `clientsessionmdb` example.

▼ To Run the `clientsessionmdb` Example Using NetBeans IDE

1. **To compile and package the project, follow these steps:**
   a. **From the File menu, choose Open Project.**
   b. **In the Open Project dialog, navigate to:**
      tut-install`/examples/jms/`
   c. **Select the `clientsessionmdb` folder.**
   d. **Select the Open as Main Project check box and the Open Required Projects check box.**
   e. **Click Open Project.**
   f. **In the Projects tab, right-click the `clientsessionmdb` project and select Build.**
      This task creates the following:
      - An application client JAR file that contains the client class file and the session bean's remote interface, along with a manifest file that specifies the main class and places the EJB JAR file in its classpath
      - An EJB JAR file that contains both the session bean and the message-driven bean

- An application EAR file that contains the two JAR files

2. **Right-click the project and select Run.**

   This command creates any needed resources, deploys the project, returns a JAR file named `clientsessionmdbClient.jar`, and then executes it.

   The output of the application client in the Output pane looks like this (preceded by application client container output):

```
To view the bean output,
 check <install_dir>/domains/domain1/logs/server.log.
```

   The output from the enterprise beans appears in the server log (*domain-dir*`/logs/server.log`), wrapped in logging information. The Publisher session bean sends two sets of 18 messages numbered 0 through 17. Because of the message selector, the message-driven bean receives only the messages whose `NewsType` property is `Sports` or `Opinion`.

▼ To Run the `clientsessionmdb` Example Using Ant

1. **Go to the following directory:**

   tut-install`/examples/jms/clientsessionmdb/`

2. **To compile the source files and package the application, use the following command:**

```
ant
```

   The `ant` command creates the following:

   - An application client JAR file that contains the client class file and the session bean's remote interface, along with a manifest file that specifies the main class and places the EJB JAR file in its classpath
   - An EJB JAR file that contains both the session bean and the message-driven bean
   - An application EAR file that contains the two JAR files

   The `clientsessionmdb.ear` file is created in the `dist` directory.

3. **To create any needed resources, deploy the application, and run the client, use the following command:**

```
ant run
```

   Ignore the message that states that the application is deployed at a URL.

   The client displays these lines (preceded by application client container output):

```
To view the bean output,
 check <install_dir>/domains/domain1/logs/server.log.
```

The output from the enterprise beans appears in the server log file, wrapped in logging information. The Publisher session bean sends two sets of 18 messages numbered 0 through 17. Because of the message selector, the message-driven bean receives only the messages whose `NewsType` property is `Sports` or `Opinion`.

An Application That Uses the JMS API with an Entity

This section explains how to write, compile, package, deploy, and run an application that uses the JMS API with an entity. The application uses the following components:

- An application client that both sends and receives messages
- Two message-driven beans
- An entity class

You will find the source files for this section in the *tut-install*`/examples/jms/clientmdbentity/` directory. Path names in this section are relative to this directory.

Overview of the `clientmdbentity` Example Application

This application simulates, in a simplified way, the work flow of a company's human resources (HR) department when it processes a new hire. This application also demonstrates how to use the Java EE platform to accomplish a task that many JMS applications need to perform.

A JMS client must often wait for several messages from various sources. It then uses the information in all these messages to assemble a message that it then sends to another destination. The common term for this process is *joining messages*. Such a task must be transactional, with all the receives and the send as a single transaction. If not all the messages are received successfully, the transaction can be rolled back. For an application client example that illustrates this task, see "A Local Transaction Example" on page 891.

A message-driven bean can process only one message at a time in a transaction. To provide the ability to join messages, an application can have the message-driven bean store the interim information in an entity. The entity can then determine whether all the information has been received; when it has, the entity can report this back to one of the message-driven beans, which then creates and sends the message to the other destination. After it has completed its task, the entity can be removed.

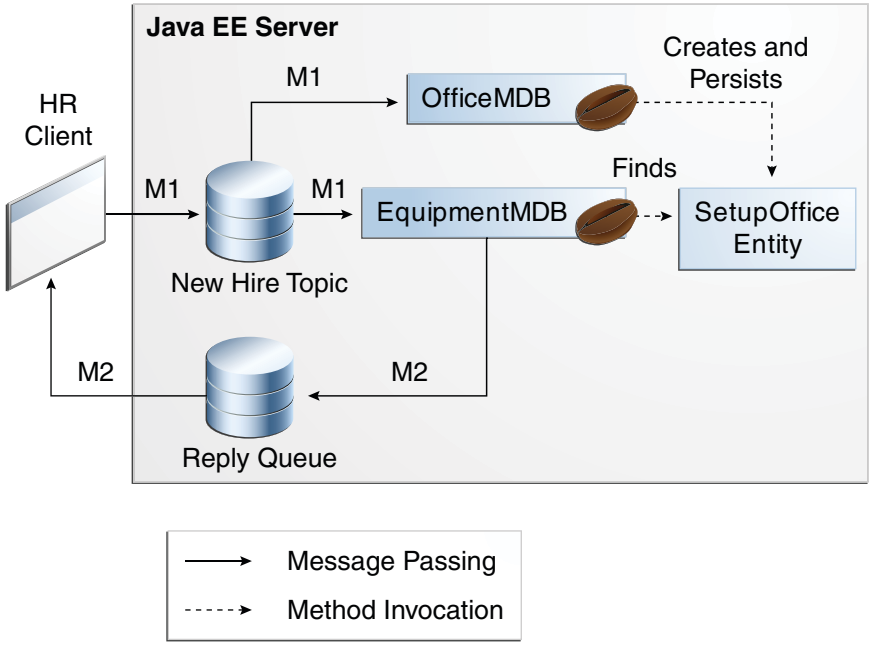
The basic steps of the application are as follows.

1. The HR department's application client generates an employee ID for each new hire and then publishes a message (M1) containing the new hire's name, employee ID, and position. The client then creates a temporary queue, `ReplyQueue`, with a message listener that waits for a reply to the message. (See "Creating Temporary Destinations" on page 846 for more information.)
2. Two message-driven beans process each message: One bean, `OfficeMDB`, assigns the new hire's office number, and the other bean, `EquipmentMDB`, assigns the new hire's equipment. The first bean to process the message creates and persists an entity named `SetupOffice`, then calls a business method of the entity to store the information it has generated. The second bean locates the existing entity and calls another business method to add its information.
3. When both the office and the equipment have been assigned, the entity business method returns a value of `true` to the message-driven bean that called the method. The message-driven bean then sends to the reply queue a message (M2) describing the assignments. Then it removes the entity. The application client's message listener retrieves the information.

Figure 48–4 illustrates the structure of this application. Of course, an actual HR application would have more components; other beans could set up payroll and benefits records, schedule orientation, and so on.

Figure 48–4 assumes that `OfficeMDB` is the first message-driven bean to consume the message from the client. `OfficeMDB` then creates and persists the `SetupOffice` entity and stores the office information. `EquipmentMDB` then finds the entity, stores the equipment information, and learns that the entity has completed its work. `EquipmentMDB` then sends the message to the reply queue and removes the entity.

FIGURE 48–4 An Enterprise Bean Application: Client to Message-Driven Beans to Entity



Writing the Application Components for the `clientmdbentity` Example

Writing the components of the application involves coding the application client, the message-driven beans, and the entity class.

Coding the Application Client: `HumanResourceClient.java`

The application client, `clientmdbentity-app-client/src/java/HumanResourceClient.java`, performs the following steps:

1. Injects `ConnectionFactory` and `Topic` resources
2. Creates a `TemporaryQueue` to receive notification of processing that occurs, based on new-hire events it has published
3. Creates a `MessageConsumer` for the `TemporaryQueue`, sets the `MessageConsumer`'s message listener, and starts the connection
4. Creates a `MessageProducer` and a `MapMessage`
5. Creates five new employees with randomly generated names, positions, and ID numbers (in sequence) and publishes five messages containing this information

The message listener, `HRListener`, waits for messages that contain the assigned office and equipment for each employee. When a message arrives, the message listener displays the information received and determines whether all five messages have arrived. When they have, the message listener notifies the `main` method, which then exits.

Coding the Message-Driven Beans for the `clientmdbentity` Example

This example uses two message-driven beans:

- `clientmdbentity-ejb/src/java/eb/EquipmentMDB.java`
- `clientmdbentity-ejb/src/java/eb/OfficeMDB.java`

The beans take the following steps:

1. They inject `MessageDrivenContext` and `ConnectionFactory` resources.
2. The `onMessage` method retrieves the information in the message. The `EquipmentMDB`'s `onMessage` method chooses equipment, based on the new hire's position; the `OfficeMDB`'s `onMessage` method randomly generates an office number.
3. After a slight delay to simulate real world processing hitches, the `onMessage` method calls a helper method, `compose`.
4. The `compose` method takes the following steps:
    a. It either creates and persists the `SetupOffice` entity or finds it by primary key.
    b. It uses the entity to store the equipment or the office information in the database, calling either the `doEquipmentList` or the `doOfficeNumber` business method.
    c. If the business method returns `true`, meaning that all of the information has been stored, it creates a connection and a session, retrieves the reply destination information from the message, creates a `MessageProducer`, and sends a reply message that contains the information stored in the entity.
    d. It removes the entity.

Coding the Entity Class for the `clientmdbentity` Example

The `SetupOffice` class, `clientmdbentity-ejb/src/java/eb/SetupOffice.java`, is an entity class. The entity and the message-driven beans are packaged together in an EJB JAR file. The entity class is declared as follows:

```
@Entity
public class SetupOffice implements Serializable {
```

The class contains a no-argument constructor and a constructor that takes two arguments, the employee ID and name. It also contains getter and setter methods for the employee ID, name, office number, and equipment list. The getter method for the employee ID has the `@Id` annotation to indicate that this field is the primary key:

```
@Id
public String getEmployeeId() {
    return id;
}
```

The class also implements the two business methods, `doEquipmentList` and `doOfficeNumber`, and their helper method, `checkIfSetupComplete`.

The message-driven beans call the business methods and the getter methods.

The `persistence.xml` file for the entity specifies the most basic settings:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
               http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="clientmdbentity-ejbPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/__default</jta-data-source>
    <class>eb.SetupOffice</class>
    <properties>
      <property name="eclipselink.ddl-generation"
                value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Creating Resources for the `clientmdbentity` Example

This example uses the connection factory `jms/ConnectionFactory` and the topic `jms/Topic`, both of which you used in "An Application That Uses the JMS API with a Session Bean" on page 897. It also uses the JDBC resource named `jdbc/__default`, which is enabled by default when you start the GlassFish Server.

If you deleted the connection factory or topic, they will be created when you deploy the example.

Running the `clientmdbentity` Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `clientmdbentity` example.

▼ To Run the `clientmdbentity` Example Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**
   tut-install`/examples/jms/`
3. **Select the `clientmdbentity` folder.**
4. **Select the Open as Main Project check box and the Open Required Projects check box.**
5. **Click Open Project.**
6. **In the Projects tab, right-click the `clientmdbentity` project and select Build.**
   This task creates the following:
   - An application client JAR file that contains the client class and listener class files, along with a manifest file that specifies the main class
   - An EJB JAR file that contains the message-driven beans and the entity class, along with the `persistence.xml` file
   - An application EAR file that contains the two JAR files along with an `application.xml` file
7. **If the Java DB database is not already running, follow these steps:**
   a. **Click the Services tab.**
   b. **Expand the Databases node.**
   c. **Right-click the Java DB node and select Start Server.**
8. **In the Projects tab, right-click the project and select Run.**
   This command creates any needed resources, deploys the project, returns a client JAR file named `clientmdbentityClient.jar`, and then executes it.

   The output of the application client in the Output pane looks something like this:

```
PUBLISHER: Setting hire ID to 50, name Bill Tudor, position Programmer
PUBLISHER: Setting hire ID to 51, name Carol Jones, position Senior Programmer
PUBLISHER: Setting hire ID to 52, name Mark Wilson, position Manager
PUBLISHER: Setting hire ID to 53, name Polly Wren, position Senior Programmer
PUBLISHER: Setting hire ID to 54, name Joe Lawrence, position Director
Waiting for 5 message(s)
New hire event processed:
  Employee ID: 52
  Name: Mark Wilson
  Equipment: PDA
  Office number: 294
```

```
Waiting for 4 message(s)
New hire event processed:
  Employee ID: 53
  Name: Polly Wren
  Equipment: Laptop
  Office number: 186
Waiting for 3 message(s)
New hire event processed:
  Employee ID: 54
  Name: Joe Lawrence
  Equipment: Java Phone
  Office number: 135
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 50
  Name: Bill Tudor
  Equipment: Desktop System
  Office number: 200
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 51
  Name: Carol Jones
  Equipment: Laptop
  Office number: 262
```

The output from the message-driven beans and the entity class appears in the server log, wrapped in logging information.

For each employee, the application first creates the entity and then finds it. You may see runtime errors in the server log, and transaction rollbacks may occur. The errors occur if both of the message-driven beans discover at the same time that the entity does not yet exist, so they both try to create it. The first attempt succeeds, but the second fails because the bean already exists. After the rollback, the second message-driven bean tries again and succeeds in finding the entity. Container-managed transactions allow the application to run correctly, in spite of these errors, with no special programming.

▼ To Run the `clientmdbentity` Example Using Ant

1 Go to the following directory:

tut-install`/examples/jms/clientmdbentity/`

2 To compile the source files and package the application, use the following command:

```
ant
```

The `ant` command creates the following:

- An application client JAR file that contains the client class and listener class files, along with a manifest file that specifies the main class
- An EJB JAR file that contains the message-driven beans and the entity class, along with the `persistence.xml` file
- An application EAR file that contains the two JAR files along with an `application.xml` file

3 To create any needed resources, deploy the application, and run the client, use the following command:

```
ant run
```

This command starts the database server if it is not already running, then deploys and runs the application.

Ignore the message that states that the application is deployed at a URL.

The output in the terminal window looks something like this (preceded by application client container output):

```
running application client container.
PUBLISHER: Setting hire ID to 50, name Bill Tudor, position Programmer
PUBLISHER: Setting hire ID to 51, name Carol Jones, position Senior Programmer
PUBLISHER: Setting hire ID to 52, name Mark Wilson, position Manager
PUBLISHER: Setting hire ID to 53, name Polly Wren, position Senior Programmer
PUBLISHER: Setting hire ID to 54, name Joe Lawrence, position Director
Waiting for 5 message(s)
New hire event processed:
  Employee ID: 52
  Name: Mark Wilson
  Equipment: PDA
  Office number: 294
Waiting for 4 message(s)
New hire event processed:
  Employee ID: 53
  Name: Polly Wren
  Equipment: Laptop
  Office number: 186
Waiting for 3 message(s)
New hire event processed:
  Employee ID: 54
  Name: Joe Lawrence
  Equipment: Java Phone
  Office number: 135
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 50
  Name: Bill Tudor
  Equipment: Desktop System
  Office number: 200
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 51
  Name: Carol Jones
  Equipment: Laptop
  Office number: 262
```

The output from the message-driven beans and the entity class appears in the server log, wrapped in logging information.

For each employee, the application first creates the entity and then finds it. You may see runtime errors in the server log, and transaction rollbacks may occur. The errors occur if both of the message-driven beans discover at the same time that the entity does not yet exist, so they both try to create it. The first attempt succeeds, but the second fails because the bean already

exists. After the rollback, the second message-driven bean tries again and succeeds in finding the entity. Container-managed transactions allow the application to run correctly, in spite of these errors, with no special programming.

An Application Example That Consumes Messages from a Remote Server

This section and the following section explain how to write, compile, package, deploy, and run a pair of Java EE modules that run on two Java EE servers and that use the JMS API to interchange messages with each other. It is a common practice to deploy different components of an enterprise application on different systems within a company, and these examples illustrate on a small scale how to do this for an application that uses the JMS API.

The two examples work in slightly different ways. In the first example, the deployment information for a message-driven bean specifies the remote server from which it will *consume* messages. In the next example, described in "An Application Example That Deploys a Message-Driven Bean on Two Servers" on page 916, the same message-driven bean is deployed on two different servers, so it is the client module that specifies the servers (one local, one remote) to which it is *sending* messages.

This first example divides the example in Chapter 25, "A Message-Driven Bean Example," into two modules: one containing the application client, and the other containing the message-driven bean.

You will find the source files for this section in the *tut-install*`/examples/jms/consumeremote/` directory. Path names in this section are relative to this directory.

Overview of the `consumeremote` Example Modules

This example is very similar to the one in Chapter 25, "A Message-Driven Bean Example," except for the fact that it is packaged as two separate modules:

- One module contains the application client, which runs on the remote system and sends three messages to a queue.
- The other module contains the message-driven bean, which is deployed on the local server and consumes the messages from the queue on the remote server.

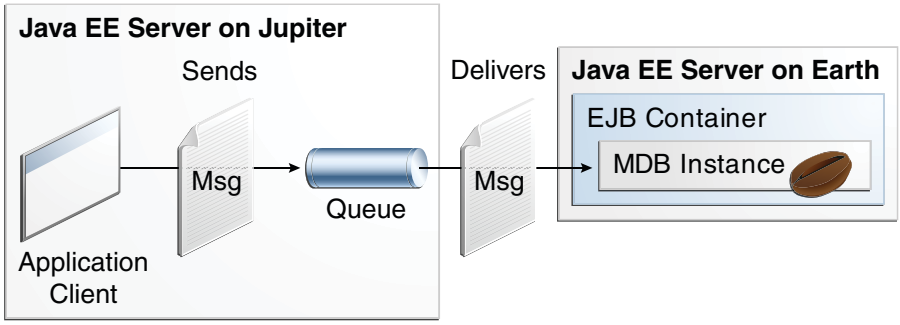
The basic steps of the modules are as follows:

1. The administrator starts two Java EE servers, one on each system.
2. On the local server, the administrator deploys the message-driven bean module, which specifies the remote server where the client is deployed.
3. On the remote server, the administrator places the client JAR file.

4. The client module sends three messages to a queue.
5. The message-driven bean consumes the messages.

Figure 48–5 illustrates the structure of this application. You can see that it is almost identical to Figure 25–1 except that there are two Java EE servers. The queue used is the one on the remote server; the queue must also exist on the local server for resource injection to succeed.

FIGURE 48–5 A Java EE Application That Consumes Messages from a Remote Server



Writing the Module Components for the `consumeremote` Example

Writing the components of the modules involves

- Coding the application client
- Coding the message-driven bean

The application client, `jupiterclient/src/java/SimpleClient.java`, is almost identical to the one in "The `simplemessage` Application Client" on page 492.

Similarly, the message-driven bean, `earthmdb/src/java/MessageBean.java`, is almost identical to the one in "The Message-Driven Bean Class" on page 493. The only significant difference is that the activation config properties include one property that specifies the name of the remote system. You need to edit the source file to specify the name of your system.

Creating Resources for the `consumeremote` Example

The application client can use any connection factory that exists on the remote server; in this example, it uses `jms/ConnectionFactory`. Both components use the queue named `jms/Queue`,

which you created for "A Simple Example of Synchronous Message Receives" on page 860. The message-driven bean does not need a previously created connection factory; the resource adapter creates one for it.

Any missing resources will be created when you deploy the example.

Using Two Application Servers for the `consumeremote` Example

As in "Running JMS Clients on Multiple Systems" on page 880, the two servers are referred to as `earth` and `jupiter`.

The GlassFish Server must be running on both systems.

Before you can run the example, you must change the default name of the JMS host on `jupiter`, as described in "To Change the Default Host Name Using the Administration Console" on page 882. If you have already performed this task, you do not have to repeat it.

Which system you use to package and deploy the modules and which system you use to run the client depend on your network configuration (specifically, which file system you can access remotely). These instructions assume you can access the file system of `jupiter` from `earth` but cannot access the file system of `earth` from `jupiter`. (You can use the same systems for `jupiter` and `earth` that you used in "Running JMS Clients on Multiple Systems" on page 880.)

You can package both modules on `earth` and deploy the message-driven bean there. The only action you perform on `jupiter` is running the client module.

Running the `consumeremote` Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `consumeremote` example.

▼ To Run the `consumeremote` Example Using NetBeans IDE

To edit the message-driven bean source file and then package, deploy, and run the modules using NetBeans IDE, follow these steps.

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

   tut-install`/examples/jms/consumeremote/`
3. **Select the `earthmdb` folder.**

4. **Select the Open as Main Project check box.**
5. **Click Open Project.**
6. **Edit the `MessageBean.java` file as follows:**
   a. **In the Projects tab, expand the `earthmdb`, Source Packages, and `mdb` nodes, then double-click `MessageBean.java`.**
   b. **Find the following line within the `@MessageDriven` annotation:**

```
@ActivationConfigProperty(propertyName = "addressList",
    propertyValue = "remotesystem"),
```

   c. **Replace `remotesystem` with the name of your remote system.**
7. **Right-click the `earthmdb` project and select Build.**
   This command creates a JAR file that contains the bean class file.
8. **From the File menu, choose Open Project.**
9. **Select the `jupiterclient` folder.**
10. **Select the Open as Main Project check box.**
11. **Click Open Project.**
12. **In the Projects tab, right-click the `jupiterclient` project and select Build.**
    This command creates a JAR file that contains the client class file and a manifest file.
13. **Right-click the `earthmdb` project and select Deploy.**
14. **To copy the `jupiterclient` module to the remote system, follow these steps:**
    a. **Change to the directory `jupiterclient/dist`:**

```
cd tut-install/examples/jms/consumeremote/jupiterclient/dist
```

    b. **Type a command like the following:**

```
cp jupiterclient.jar F:/
```

    That is, copy the client JAR file to a location on the remote file system. You can use the file system graphical user interface on your system instead of the command line.

15 To run the application client, follow these steps:

a. If you did not previously create the queue and connection factory on the remote system (`jupiter`), go to the *tut-install*`/examples/jms/consumeremote/jupiterclient/` directory on the remote system and type the following command:

```
ant add-resources
```

b. Go to the directory on the remote system (`jupiter`) where you copied the client JAR file.

c. To deploy the client module and retrieve the client stubs, use the following command:

```
asadmin deploy --retrieve . jupiterclient.jar
```

This command deploys the client JAR file and retrieves the client stubs in a file named `jupiterclientClient.jar`

d. To run the client, use the following command:

```
appclient -client jupiterclientClient.jar
```

On `jupiter`, the output of the `appclient` command looks like this (preceded by application client container output):

```
Sending message: This is message 1 from jupiterclient
Sending message: This is message 2 from jupiterclient
Sending message: This is message 3 from jupiterclient
```

On `earth`, the output in the server log looks something like this (preceded by logging information):

```
MESSAGE BEAN: Message received: This is message 1 from jupiterclient
MESSAGE BEAN: Message received: This is message 2 from jupiterclient
MESSAGE BEAN: Message received: This is message 3 from jupiterclient
```

e. To undeploy the client after you finish running it, use the following command:

```
asadmin undeploy jupiterclient
```

▼ To Run the `consumeremote` Example Using Ant

To edit the message-driven bean source file and then package, deploy, and run the modules using Ant, follow these steps.

1 Open the following file in an editor:

tut-install`/examples/jms/consumeremote/earthmdb/src/java/mdb/MessageBean.java`

2 Find the following line within the `@MessageDriven` annotation:

```
@ActivationConfigProperty(propertyName = "addressList",
    propertyValue = "remotesystem"),
```

3 Replace `remotesystem` with the name of your remote system, then save and close the file.

4. **Go to the following directory:**

   tut-install`/examples/jms/consumeremote/earthmdb/`

5. **Type the following command:**

   ```
   ant
   ```

   This command creates a JAR file that contains the bean class file.

6. **Type the following command:**

   ```
   ant deploy
   ```

7. **Go to the `jupiterclient` directory:**

   ```
   cd ../jupiterclient
   ```

8. **Type the following command:**

   ```
   ant
   ```

   This command creates a JAR file that contains the client class file and a manifest file.

9. **To copy the `jupiterclient` module to the remote system, follow these steps:**

   a. **Change to the directory `jupiterclient/dist`:**

      ```
      cd ../jupiterclient/dist
      ```

   b. **Type a command like the following:**

      ```
      cp jupiterclient.jar F:/
      ```

      That is, copy the client JAR file to a location on the remote file system.

10. **To run the application client, follow these steps:**

    a. **If you did not previously create the queue and connection factory on the remote system (`jupiter`), go to the *tut-install*`/examples/jms/consumeremote/jupiterclient/` directory on the remote system and type the following command:**

       ```
       ant add-resources
       ```

    b. **Go to the directory on the remote system (`jupiter`) where you copied the client JAR file.**

    c. **To deploy the client module and retrieve the client stubs, use the following command:**

       ```
       asadmin deploy --retrieve . jupiterclient.jar
       ```

       This command deploys the client JAR file and retrieves the client stubs in a file named `jupiterclientClient.jar`

    d. **To run the client, use the following command:**

       ```
       appclient -client jupiterclientClient.jar
       ```

On `jupiter`, the output of the `appclient` command looks like this (preceded by application client container output):

```
Sending message: This is message 1 from jupiterclient
Sending message: This is message 2 from jupiterclient
Sending message: This is message 3 from jupiterclient
```

On `earth`, the output in the server log looks something like this (preceded by logging information):

```
MESSAGE BEAN: Message received: This is message 1 from jupiterclient
MESSAGE BEAN: Message received: This is message 2 from jupiterclient
MESSAGE BEAN: Message received: This is message 3 from jupiterclient
```

e. To undeploy the client after you finish running it, use the following command:

```
asadmin undeploy jupiterclient
```

An Application Example That Deploys a Message-Driven Bean on Two Servers

This section, like the preceding one, explains how to write, compile, package, deploy, and run a pair of Java EE modules that use the JMS API and run on two Java EE servers. These modules are slightly more complex than the ones in the first example.

The modules use the following components:

- An application client that is deployed on the local server. It uses two connection factories, an ordinary one and one configured to communicate with the remote server, to create two publishers and two subscribers and to publish and consume messages.
- A message-driven bean that is deployed twice: once on the local server, and once on the remote one. It processes the messages and sends replies.

In this section, the term *local server* means the server on which both the application client and the message-driven bean are deployed (`earth` in the preceding example). The term *remote server* means the server on which only the message-driven bean is deployed (`jupiter` in the preceding example).

You will find the source files for this section in the *tut-install*`/examples/jms/sendremote/` directory. Path names in this section are relative to this directory.

Overview of the `sendremote` Example Modules

This pair of modules is somewhat similar to the modules in "An Application Example That Consumes Messages from a Remote Server" on page 910 in that the only components are a

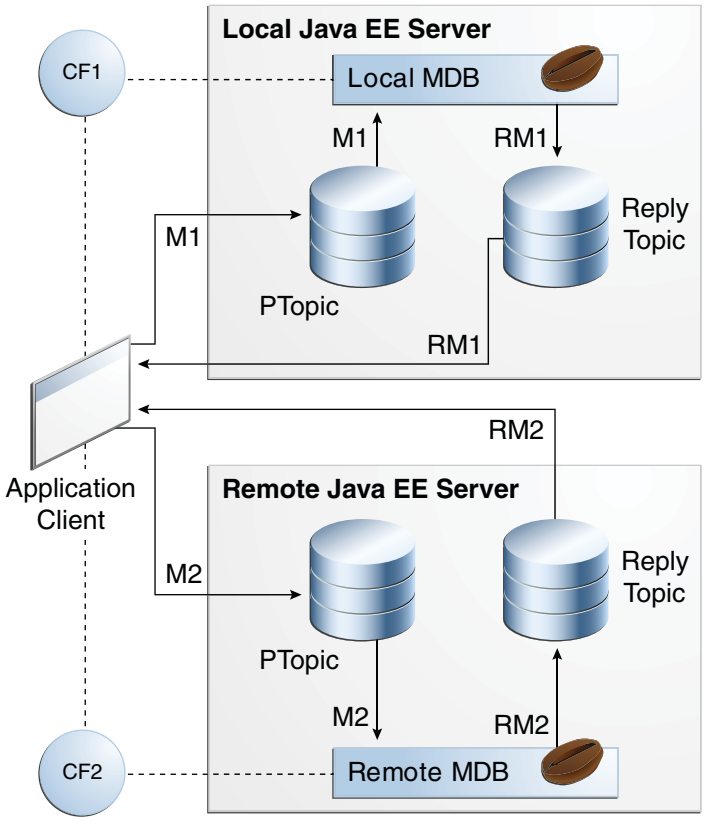
client and a message-driven bean. However, the modules here use these components in more complex ways. One module consists of the application client. The other module contains only the message-driven bean and is deployed twice, once on each server.

The basic steps of the modules are as follows.

1. You start two Java EE servers, one on each system.
2. On the local server (`earth`), you create two connection factories: one local and one that communicates with the remote server (`jupiter`). On the remote server, you create a connection factory that has the same name as the one that communicates with the remote server.
3. The application client looks up the two connection factories (the local one and the one that communicates with the remote server) to create two connections, sessions, publishers, and subscribers. The subscribers use a message listener.
4. Each publisher publishes five messages.
5. Each of the local and the remote message-driven beans receives five messages and sends replies.
6. The client's message listener consumes the replies.

Figure 48–6 illustrates the structure of this application. M1 represents the first message sent using the local connection factory, and RM1 represents the first reply message sent by the local MDB. M2 represents the first message sent using the remote connection factory, and RM2 represents the first reply message sent by the remote MDB.

FIGURE 48–6 A Java EE Application That Sends Messages to Two Servers



Writing the Module Components for the `sendremote` Example

Writing the components of the modules involves coding the application client and the message-driven bean.

Coding the Application Client: `MultiAppServerClient.java`

The application client class, `multiclient/src/java/MultiAppServerClient.java`, does the following.

1. It injects resources for two connection factories and a topic.
2. For each connection factory, it creates a connection, a publisher session, a publisher, a subscriber session, a subscriber, and a temporary topic for replies.
3. Each subscriber sets its message listener, `ReplyListener`, and starts the connection.

4. Each publisher publishes five messages and creates a list of the messages the listener should expect.
5. When each reply arrives, the message listener displays its contents and removes it from the list of expected messages.
6. When all the messages have arrived, the client exits.

Coding the Message-Driven Bean: `ReplyMsgBean.java`

The message-driven bean class, `replybean/src/ReplyMsgBean.java`, does the following:

1. Uses the `@MessageDriven` annotation:

```
@MessageDriven(mappedName = "jms/Topic")
```

2. Injects resources for the `MessageDrivenContext` and for a connection factory. It does not need a destination resource because it uses the value of the incoming message's `JMSReplyTo` header as the destination.
3. Uses a `@PostConstruct` callback method to create the connection, and a `@PreDestroy` callback method to close the connection.

The `onMessage` method of the message-driven bean class does the following:

1. Casts the incoming message to a `TextMessage` and displays the text
2. Creates a connection, a session, and a publisher for the reply message
3. Publishes the message to the reply topic
4. Closes the connection

On both servers, the bean will consume messages from the topic `jms/Topic`.

Creating Resources for the `sendremote` Example

This example uses the connection factory named `jms/ConnectionFactory` and the topic named `jms/Topic`. These objects must exist on both the local and the remote servers.

This example uses an additional connection factory, `jms/JupiterConnectionFactory`, which communicates with the remote system; you created it in "To Create Administered Objects for Multiple Systems" on page 881. This connection factory must exist on the local server.

The `build.xml` file for the `multiclient` module contains targets you can use to create these resources if you deleted them previously.

To create the resource needed only on the local system, use the following command:

```
ant create-remote-factory -Dsys=remote-system-name
```

The other resources will be created when you deploy the application.

▼ To Enable Deployment on the Remote System

GlassFish Server by default does not allow deployment on a remote system. You must create a password for the administrator on the remote system, then enable secure administration on that system. After that, you will be able to deploy the message-driven bean on the remote system.

1. **On `jupiter`, start the Administration Console by opening a browser at `http://localhost:4848/`.**
2. **In the navigation tree, expand the Configurations node, then expand the server-config node.**
3. **Expand the Security node.**
4. **Expand the Realms node.**
5. **Select the admin-realm node.**
6. **On the Edit Realm page, click Manage Users.**
7. **In the File Users table, click admin in the User ID column.**
8. **On the Edit File Realm Users page, type a password (for example, jmsadmin) in the New Password and Confirm New Password fields, then click Save.**
9. **In the navigation tree, click the Server (Admin Server) node.**
10. **On the General Information page, click Secure Administration.**
11. **Click Enable Secure Admin, accepting the default values for the alias and instance.**
12. **The server on `jupiter` will stop and restart automatically. Log in to the Administration Console with the `admin` user ID and the password you created and verify that the settings are correct.**

▼ To Use Two Application Servers for the `sendremote` Example

If you are using NetBeans IDE, you need to add the remote server in order to deploy the message-driven bean there. To do so, follow these steps.

1. **In NetBeans IDE, click the Services tab.**

2. **Right-click the Servers node and select Add Server. In the Add Server Instance dialog, follow these steps:**

   a. **Select GlassFish Server 3+ from the Server list.**

   b. **In the Name field, specify a name slightly different from that of the local server, such as `GlassFish Server 3+ (2)`.**

   c. **Click Next.**

   d. **For the Server Location, browse to the location of the GlassFish Server on the remote system. This location must be visible from the local system.**

   e. **Click Next.**

   f. **Select the Register Remote Domain radio button.**

   g. **In the Host Name field, type the name of the remote system.**

   h. **Click Finish.**

   i. **In the dialog that appears, enter the user name (admin) and the password you created.**

Next Steps Before you can run the example, you must change the default name of the JMS host on `jupiter`, as described in "To Change the Default Host Name Using the Administration Console" on page 882. If you have already performed this task, you do not have to repeat it.

Running the `sendremote` Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `sendremote` example.

▼ To Run the `sendremote` Example Using NetBeans IDE

1. **To build the `replybean` module, follow these steps:**

   a. **From the File menu, choose Open Project.**

   b. **In the Open Project dialog, navigate to:**

      tut-install`/examples/jms/sendremote/`

   c. **Select the `replybean` folder.**

d. **Select the Open as Main Project check box.**

e. **Click Open Project.**

f. **In the Projects tab, right-click the `replybean` project and select Build.**
   This command creates a JAR file that contains the bean class file.

2 **To build the `multiclient` module, follow these steps:**

a. **From the File menu, choose Open Project.**

b. **Select the `multiclient` folder.**

c. **Select the Open as Main Project check box.**

d. **Click Open Project.**

e. **In the Projects tab, right-click the `multiclient` project and select Build.**
   This command creates a JAR file that contains the client class file and a manifest file.

3 **To create any needed resources and deploy the `multiclient` module on the local server, follow these steps:**

a. **Right-click the `multiclient` project and select Properties.**

b. **Select Run from the Categories tree.**

c. **From the Server list, select GlassFish Server 3+ (the local server).**

d. **Click OK.**

e. **Right-click the `multiclient` project and select Deploy.**
   You can use the Services tab to verify that `multiclient` is deployed as an App Client Module on the local server.

4 **To deploy the `replybean` module on the local and remote servers, follow these steps:**

a. **Right-click the `replybean` project and select Properties.**

b. **Select Run from the Categories tree.**

c. **From the Server list, select GlassFish Server 3+ (the local server).**

d. Click OK.

e. Right-click the `replybean` project and select Deploy.

f. Right-click the `replybean` project again and select Properties.

g. Select Run from the Categories tree.

h. From the Server list, select GlassFish Server 3+ (2) (the remote server).

i. Click OK.

j. Right-click the `replybean` project and select Deploy.

You can use the Services tab to verify that `replybean` is deployed as an EJB Module on both servers.

5 To run the application client, right-click the `multiclient` project and select Run.

This command returns a JAR file named `multiclientClient.jar` and then executes it.

On the local system, the output of the `appclient` command looks something like this:

```
running application client container.
...
Sent message: text: id=1 to local app server
Sent message: text: id=2 to remote app server
ReplyListener: Received message: id=1, text=ReplyMsgBean processed message:
text: id=1 to local app server
Sent message: text: id=3 to local app server
ReplyListener: Received message: id=3, text=ReplyMsgBean processed message:
text: id=3 to local app server
ReplyListener: Received message: id=2, text=ReplyMsgBean processed message:
text: id=2 to remote app server
Sent message: text: id=4 to remote app server
ReplyListener: Received message: id=4, text=ReplyMsgBean processed message:
text: id=4 to remote app server
Sent message: text: id=5 to local app server
ReplyListener: Received message: id=5, text=ReplyMsgBean processed message:
text: id=5 to local app server
Sent message: text: id=6 to remote app server
ReplyListener: Received message: id=6, text=ReplyMsgBean processed message:
text: id=6 to remote app server
Sent message: text: id=7 to local app server
ReplyListener: Received message: id=7, text=ReplyMsgBean processed message:
text: id=7 to local app server
Sent message: text: id=8 to remote app server
ReplyListener: Received message: id=8, text=ReplyMsgBean processed message:
text: id=8 to remote app server
Sent message: text: id=9 to local app server
ReplyListener: Received message: id=9, text=ReplyMsgBean processed message:
text: id=9 to local app server
Sent message: text: id=10 to remote app server
ReplyListener: Received message: id=10, text=ReplyMsgBean processed message:
text: id=10 to remote app server
```

```
Waiting for 0 message(s) from local app server
Waiting for 0 message(s) from remote app server
Finished
Closing connection 1
Closing connection 2
```

On the local system, where the message-driven bean receives the odd-numbered messages, the output in the server log looks like this (wrapped in logging information):

```
ReplyMsgBean: Received message: text: id=1 to local app server
ReplyMsgBean: Received message: text: id=3 to local app server
ReplyMsgBean: Received message: text: id=5 to local app server
ReplyMsgBean: Received message: text: id=7 to local app server
ReplyMsgBean: Received message: text: id=9 to local app server
```

On the remote system, where the bean receives the even-numbered messages, the output in the server log looks like this (wrapped in logging information):

```
ReplyMsgBean: Received message: text: id=2 to remote app server
ReplyMsgBean: Received message: text: id=4 to remote app server
ReplyMsgBean: Received message: text: id=6 to remote app server
ReplyMsgBean: Received message: text: id=8 to remote app server
ReplyMsgBean: Received message: text: id=10 to remote app server
```

▼ To Run the `sendremote` Example Using Ant

1 To package the modules, follow these steps:

a. Go to the following directory:

tut-install`/examples/jms/sendremote/multiclient/`

b. Type the following command:

```
ant
```

This command creates a JAR file that contains the client class file and a manifest file.

c. Change to the directory `replybean`:

```
cd ../replybean
```

d. Type the following command:

```
ant
```

This command creates a JAR file that contains the bean class file.

2 To deploy the `replybean` module on the local and remote servers, follow these steps:

a. Verify that you are still in the directory `replybean`.

b. Type the following command:

```
ant deploy
```

Ignore the message that states that the application is deployed at a URL.

c. Type the following command:

```
ant deploy-remote -Dsys=remote-system-name
```

Replace *remote-system-name* with the actual name of the remote system.

3 To deploy the client, follow these steps:

a. Change to the directory `multiclient`:

```
cd ../multiclient
```

b. Type the following command:

```
ant getclient
```

4 To run the client, type the following command:

```
ant run
```

On the local system, the output looks something like this:

```
running application client container.
...
Sent message: text: id=1 to local app server
Sent message: text: id=2 to remote app server
ReplyListener: Received message: id=1, text=ReplyMsgBean processed message:
text: id=1 to local app server
Sent message: text: id=3 to local app server
ReplyListener: Received message: id=3, text=ReplyMsgBean processed message:
text: id=3 to local app server
ReplyListener: Received message: id=2, text=ReplyMsgBean processed message:
text: id=2 to remote app server
Sent message: text: id=4 to remote app server
ReplyListener: Received message: id=4, text=ReplyMsgBean processed message:
text: id=4 to remote app server
Sent message: text: id=5 to local app server
ReplyListener: Received message: id=5, text=ReplyMsgBean processed message:
text: id=5 to local app server
Sent message: text: id=6 to remote app server
ReplyListener: Received message: id=6, text=ReplyMsgBean processed message:
text: id=6 to remote app server
Sent message: text: id=7 to local app server
ReplyListener: Received message: id=7, text=ReplyMsgBean processed message:
text: id=7 to local app server
Sent message: text: id=8 to remote app server
ReplyListener: Received message: id=8, text=ReplyMsgBean processed message:
text: id=8 to remote app server
Sent message: text: id=9 to local app server
ReplyListener: Received message: id=9, text=ReplyMsgBean processed message:
text: id=9 to local app server
Sent message: text: id=10 to remote app server
```

```
ReplyListener: Received message: id=10, text=ReplyMsgBean processed message:
text: id=10 to remote app server
Waiting for 0 message(s) from local app server
Waiting for 0 message(s) from remote app server
Finished
Closing connection 1
Closing connection 2
```

On the local system, where the message-driven bean receives the odd-numbered messages, the output in the server log looks like this (wrapped in logging information):

```
ReplyMsgBean: Received message: text: id=1 to local app server
ReplyMsgBean: Received message: text: id=3 to local app server
ReplyMsgBean: Received message: text: id=5 to local app server
ReplyMsgBean: Received message: text: id=7 to local app server
ReplyMsgBean: Received message: text: id=9 to local app server
```

On the remote system, where the bean receives the even-numbered messages, the output in the server log looks like this (wrapped in logging information):

```
ReplyMsgBean: Received message: text: id=2 to remote app server
ReplyMsgBean: Received message: text: id=4 to remote app server
ReplyMsgBean: Received message: text: id=6 to remote app server
ReplyMsgBean: Received message: text: id=8 to remote app server
ReplyMsgBean: Received message: text: id=10 to remote app server
```

▼ To Disable Deployment on the Remote System

After running this example and undeploying the components, you should disable secure administration on the remote system (`jupiter`). In addition, you will probably want to return the GlassFish Server on `jupiter` to its previous state of not requiring a user name and password for administration, to make it easier to run subsequent examples there.

1. **On the remote system (`jupiter`), start the Administration Console by opening a browser at `http://localhost:4848/`, if it is not already running.**

   You will need to log in.

2. **In the navigation tree, click the Server (Admin Server) node.**

3. **On the General Information page, click Secure Administration.**

4. **Click Disable Secure Admin.**

5. **The server will stop and restart automatically. Log in to the Administration Console again.**

6. **In the navigation tree, expand the Configurations node, then expand the server-config node.**

7. **Expand the Security node.**

8. **Expand the Realms node.**
9. **Select the admin-realm node.**
10. **On the Edit Realm page, click Manage Users.**
11. **In the File Users table, click admin in the User ID column.**
12. **On the Edit File Realm Users page, click Save.**
13. **In the dialog that asks you to confirm that you are setting an empty password for the specified user, click OK.**

    The next time you start the Administration Console or issue an `asadmin` command, you will not need to provide login credentials.

Next Steps On `earth`, if you used NetBeans IDE to add the remote server, you may also want to remove the server.

◆ ◆ ◆ CHAPTER 49

Bean Validation: Advanced Topics

This chapter describes how to create custom constraints, custom validator messages, and constraint groups using the Java API for JavaBeans Validation (Bean Validation).

The following topics are addressed here:

- "Creating Custom Constraints" on page 929
- "Customizing Validator Messages" on page 930
- "Grouping Constraints" on page 931

Creating Custom Constraints

Bean Validation defines annotations, interfaces, and classes to allow developers to create custom constraints.

Using the Built-In Constraints to Make a New Constraint

Bean Validation includes several built-in constraints that can be combined to create new, reusable constraints. This can simplify constraint definition by allowing developers to define a custom constraint made up of several built-in constraints that may then be applied to component attributes with a single annotation.

```
@Pattern.List({
  @Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
    +"[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*"
    +"@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?")
})
@Constraint(validatedBy = {})
@Documented
@Target({ElementType.METHOD,
```

```
    ElementType.FIELD,
    ElementType.ANNOTATION_TYPE,
    ElementType.CONSTRUCTOR,
    ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Email {

    String message() default "{invalid.email}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ElementType.METHOD,
        ElementType.FIELD,
        ElementType.ANNOTATION_TYPE,
        ElementType.CONSTRUCTOR,
        ElementType.PARAMETER})
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    @interface List {
        Email[] value();
    }
}
```

This custom constraint can then be applied to an attribute.

```
...
@Email
protected String email;
...
```

Customizing Validator Messages

Bean Validation includes a resource bundle of default messages for the built-in constraints. These messages can be customized, and localized for non-English speaking locales.

The `ValidationMessages` Resource Bundle

The `Validationmessages` resource bundle and the locale variants of this resource bundle contain strings that override the default validation messages. The `ValidationMessages` resource bundle is typically a properties file, `ValidationMessages.properties`, in the default package of an application.

Localizing Validation Messages

Locale variants of `ValidationMessages.properties` are added by appending an underscore and the locale prefix to the base name of the file. For example, the Spanish locale variant resource bundle would be `ValidationMessages_es.properties`.

Grouping Constraints

Constraints may be added to one or more groups. Constraint groups are used to create subsets of constraints so only certain constraints will be validated for a particular object. By default, all constraints are included in the `Default` constraint group.

Constraint groups are represented by interfaces.

```
public interface Employee {}

public interface Contractor {}
```

Constraint groups can inherit from other groups.

```
public interface Manager extends Employee {}
```

When a constraint is added to an element, the constraint declares the groups to which that constraint belongs by specifying the class name of the group interface name in the `groups` element of the constraint.

```
@NotNull(groups=Employee.class)
Phone workPhone;
```

Multiple groups can be declared by surrounding the groups with angle brackets ({ and }) and separating the groups' class names with commas.

```
@NotNull(groups={ Employee.class, Contractor.class })
Phone workPhone;
```

If a group inherits from another group, validating that group results in validating all constraints declared as part of the supergroup. For example, validating the `Manager` group results in the `workPhone` field being validated, because `Employee` is a superinterface of `Manager`.

Customizing Group Validation Order

By default, constraint groups are validated in no particular order. There are cases where some groups should be validated before others. For example, in a particular class, basic data should be validated before more advanced data.

To set the validation order for a group, add a `javax.validation.GroupSequence` annotation to the interface definition, listing the order in which the validation should occur.

```
@GroupSequence({Default.class, ExpensiveValidationGroup.class})
public interface FullValidationGroup {}
```

When validating `FullValidationGroup`, first the `Default` group is validated. If all the data passes validation, then the `ExpensiveValidationGroup` group is validated. If a constraint is

part of both the `Default` and the `ExpensiveValidationGroup` groups, the constraint is validated as part of the `Default` group, and will not be validated on the subsequent `ExpensiveValidationGroup` pass.

◆ ◆ ◆ CHAPTER 50

Using Java EE Interceptors

This chapter discusses how to create interceptor classes and methods that interpose on method invocations or lifecycle events on a target class.

The following topics are addressed here:

- "Overview of Interceptors" on page 933
- "Using Interceptors" on page 935
- "The `interceptor` Example Application" on page 939

Overview of Interceptors

Interceptors are used in conjunction with Java EE managed classes to allow developers to invoke interceptor methods on an associated *target class*, in conjunction with method invocations or lifecycle events. Common uses of interceptors are logging, auditing, and profiling.

The Interceptors 1.1 specification is part of the final release of JSR 318, Enterprise JavaBeans 3.1, available from `http://jcp.org/en/jsr/detail?id=318`.

An interceptor can be defined within a target class as an *interceptor method*, or in an associated class called an *interceptor class*. Interceptor classes contain methods that are invoked in conjunction with the methods or lifecycle events of the target class.

Interceptor classes and methods are defined using metadata annotations, or in the deployment descriptor of the application containing the interceptors and target classes.

Note – Applications that use the deployment descriptor to define interceptors are not portable across Java EE servers.

Interceptor methods within the target class or in an interceptor class are annotated with one of the metadata annotations defined in Table 50–1.

TABLE 50–1 Interceptor Metadata Annotations

Interceptor Metadata Annotation	Description
`javax.interceptor.AroundInvoke`	Designates the method as an interceptor method.
`javax.interceptor.AroundTimeout`	Designates the method as a timeout interceptor, for interposing on timeout methods for enterprise bean timers.
`javax.annotation.PostConstruct`	Designates the method as an interceptor method for post-construct lifecycle events.
`javax.annotation.PreDestroy`	Designates the method as an interceptor method for pre-destroy lifecycle events.

Interceptor Classes

Interceptor classes may be designated with the optional `javax.interceptor.Interceptor` annotation, but interceptor classes aren't required to be so annotated. An interceptor class *must* have a public, no-argument constructor.

The target class can have any number of interceptor classes associated with it. The order in which the interceptor classes are invoked is determined by the order in which the interceptor classes are defined in the `javax.interceptor.Interceptors` annotation. However, this order can be overridden in the deployment descriptor.

Interceptor classes may be targets of dependency injection. Dependency injection occurs when the interceptor class instance is created, using the naming context of the associated target class, and before any `@PostConstruct` callbacks are invoked.

Interceptor Lifecycle

Interceptor classes have the same lifecycle as their associated target class. When a target class instance is created, an interceptor class instance is also created for each declared interceptor class in the target class. That is, if the target class declares multiple interceptor classes, an instance of each class is created when the target class instance is created. The target class instance and all interceptor class instances are fully instantiated before any `@PostConstruct` callbacks are invoked, and any `@PreDestroy` callbacks are invoked before the target class and interceptor class instances are destroyed.

Interceptors and CDI

Contexts and Dependency Injection for the Java EE Platform (CDI) builds on the basic functionality of Java EE interceptors. For information on CDI interceptors, including a discussion of interceptor binding types, see "Using Interceptors in CDI Applications" on page 547.

Using Interceptors

An interceptor is defined using one of the interceptor metadata annotations listed in Table 50–1 within the target class, or in a separate interceptor class. The following code declares an `@AroundTimeout` interceptor method within a target class.

```
@Stateless
public class TimerBean {
...
    @Schedule(minute="*/1", hour="*")
    public void automaticTimerMethod() { ... }

    @AroundTimeout
    public void timeoutInterceptorMethod(InvocationContext ctx) { ... }
...
}
```

If interceptor classes are used, use the `javax.interceptor.Interceptors` annotation to declare one or more interceptors at the class or method level of the target class. The following code declares interceptors at the class level.

```
@Stateless
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
public class OrderBean { ... }
```

The following code declares a method-level interceptor class.

```
@Stateless
public class OrderBean {
...
    @Interceptors(OrderInterceptor.class)
    public void placeOrder(Order order) { ... }
...
}
```

Intercepting Method Invocations

The `@AroundInvoke` annotation is used to designate interceptor methods for managed object methods. Only one around-invoke interceptor method per class is allowed. Around-invoke interceptor methods have the following form:

```
@AroundInvoke
visibility Object method-name(InvocationContext) throws Exception { ... }
```

For example:

```
@AroundInvoke
public void interceptOrder(InvocationContext ctx) { ... }
```

Around-invoke interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

An around-invoke interceptor can call any component or resource callable by the target method on which it interposes, have the same security and transaction context as the target method, and run in the same Java virtual machine call-stack as the target method.

Around-invoke interceptors can throw any exception allowed by the throws clause of the target method. They may catch and suppress exceptions, and then recover by calling the `InvocationContext.proceed` method.

Using Multiple Method Interceptors

Use the `@Interceptors` annotation to declare multiple interceptors for a target method or class.

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
        LastInterceptor.class})
public void updateInfo(String info) { ... }
```

The order of the interceptors in the `@Interceptors` annotation is the order in which the interceptors are invoked.

Multiple interceptors may also be defined in the deployment descriptor. The order of the interceptors in the deployment descriptor is the order in which the interceptors will be invoked.

```
...
<interceptor-binding>
    <target-name>myapp.OrderBean</target-name>
    <interceptor-class>myapp.PrimaryInterceptor.class</interceptor-class>
    <interceptor-class>myapp.SecondaryInterceptor.class</interceptor-class>
    <interceptor-class>myapp.LastInterceptor.class</interceptor-class>
    <method-name>updateInfo</method-name>
</interceptor-binding>
...
```

To explicitly pass control to the next interceptor in the chain, call the `InvocationContext.proceed` method.

Data can be shared across interceptors:

- The same `InvocationContext` instance is passed as an input parameter to each interceptor method in the interceptor chain for a particular target method. The `InvocationContext` instance's `contextData` property is used to pass data across interceptor methods. The `contextData` property is a `java.util.Map<String, Object>` object. Data stored in `contextData` is accessible to interceptor methods further down the interceptor chain.
- The data stored in `contextData` is not sharable across separate target class method invocations. That is, a different `InvocationContext` object is created for each invocation of the method in the target class.

Accessing Target Method Parameters From an Interceptor Class

The `InvocationContext` instance passed to each around-invoke method may be used to access and modify the parameters of the target method. The `parameters` property of `InvocationContext` is an array of `Object` instances that corresponds to the parameter order of the target method. For example, for the following target method, the `parameters` property, in the `InvocationContext` instance passed to the around-invoke interceptor method in `PrimaryInterceptor`, is an `Object` array containing two `String` objects (`firstName` and `lastName`) and a `Date` object (`date`):

```
@Interceptors(PrimaryInterceptor.class)
public void updateInfo(String firstName, String lastName, Date date) { ... }
```

The parameters can be accessed and modified using the `InvocationContext.getParameters` and `InvocationContext.setParameters` methods, respectively.

Intercepting Lifecycle Callback Events

Interceptors for lifecycle callback events (post-create and pre-destroy) may be defined in the target class or in interceptor classes. The `@PostCreate` annotation is used to designate a method as a post-create lifecycle event interceptor. The `@PreDestroy` annotation is used to designate a method as a pre-destroy lifecycle event interceptor.

Lifecycle event interceptors defined within the target class have the following form:

```
void method-name() { ... }
```

For example:

```
@PostCreate
void initialize() { ... }
```

Lifecycle event interceptors defined in an interceptor class have the following form:

```
void <method-name>(InvocationContext) { ... }
```

For example:

```
@PreDestroy
void cleanup(InvocationContext ctx) { ... }
```

Lifecycle interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

Lifecycle interceptor methods are called in an unspecified security and transaction context. That is, portable Java EE applications should not assume the lifecycle event interceptor method has access to a security or transaction context. Only one interceptor method for each lifecycle event (post-create and pre-destroy) is allowed per class.

Using Multiple Lifecycle Callback Interceptors

Multiple lifecycle interceptors may be defined for a target class by specifying the interceptor classes in the `@Interceptors` annotation:

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
        LastInterceptor.class})
@Stateless
public class OrderBean { ... }
```

The order in which the interceptor classes are listed in the `@Interceptors` annotation defines the order in which the interceptors are invoked.

Data stored in the `contextData` property of `InvocationContext` is not sharable across different lifecycle events.

Intercepting Timeout Events

Interceptors for EJB timer service timeout methods may be defined using the `@AroundTimeout` annotation on methods in the target class or in an interceptor class. Only one `@AroundTimeout` method per class is allowed.

Timeout interceptors have the following form:

```
Object <method-name>(InvocationContext) throws Exception { ... }
```

For example:

```
@AroundTimeout
protected void timeoutInterceptorMethod(InvocationContext ctx) { ... }
```

Timeout interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

Timeout interceptors can call any component or resource callable by the target timeout method, and are invoked in the same transaction and security context as the target method.

Timeout interceptors may access the timer object associated with the target timeout method through the `InvocationContext` instance's `getTimer` method.

Using Multiple Timeout Interceptors

Multiple timeout interceptors may be defined for a given target class by specifying the interceptor classes containing `@AroundTimeout` interceptor methods in an `@Interceptors` annotation at the class level.

If a target class specifies timeout interceptors in an interceptor class, and also has a `@AroundTimeout` interceptor method within the target class itself, the timeout interceptors in the interceptor classes are called first, followed by the timeout interceptors defined in the target class. For example, in the following example, assume that both the `PrimaryInterceptor` and `SecondaryInterceptor` classes have timeout interceptor methods.

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
@Stateful
public class OrderBean {
...
    @AroundTimeout
    private void last(InvocationContext ctx) { ... }
...
}
```

The timeout interceptor in `PrimaryInterceptor` will be called first, followed by the timeout interceptor in `SecondaryInterceptor`, and finally the `last` method defined in the target class.

The `interceptor` Example Application

The `interceptor` example demonstrates how to use an interceptor class, containing an `@AroundInvoke` interceptor method, with a stateless session bean.

The `HelloBean` stateless session bean is a simple enterprise bean with two business methods, `getName` and `setName`, to retrieve and modify a string. The `setName` business method has an `@Interceptors` annotation that specifies an interceptor class, `HelloInterceptor`, for that method.

```
@Interceptors(HelloInterceptor.class)
public void setName(String name) {
    this.name = name;
}
```

The HelloInterceptor class defines an @AroundInvoke interceptor method, modifyGreeting, that converts the string passed to HelloBean.setName to lowercase.

```
@AroundInvoke
public Object modifyGreeting(InvocationContext ctx) throws Exception {
    Object[] parameters = ctx.getParameters();
    String param = (String) parameters[0];
    param = param.toLowerCase();
    parameters[0] = param;
    ctx.setParameters(parameters);
    try {
        return ctx.proceed();
    } catch (Exception e) {
        logger.warning("Error calling ctx.proceed in modifyGreeting()");
        return null;
    }
}
```

The parameters to HelloBean.setName are retrieved and stored in an Object array by calling the InvocationContext.getParameters method. Because setName has only one parameter, it is the first and only element in the array. The string is set to lowercase and stored in the parameters array, then passed to InvocationContext.setParameters. To return control to the session bean, InvocationContext.proceed is called.

The user interface of interceptor is a JavaServer Faces web application that consists of two Facelets views: index.xhtml, which contains a form for entering the name, and response.xhtml, which displays the final name.

Running the interceptor Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the interceptor example.

▼ To Run the interceptor Example Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to** *tut-install*`/examples/ejb/`**.**
3. **Select the** `interceptor` **folder and click Open Project.**
4. **In the Projects tab, right-click the** `interceptor` **project and select Run.**

   This will compile, deploy, and run the interceptor example, opening a web browser page to http://localhost:8080/interceptor/.

5. **Type a name into the form and select Submit.**

   The name will be converted to lowercase by the method interceptor defined in the `HelloInterceptor` class.

▼ To Run the `interceptor` Example Using Ant

1. **Go to the following directory:**

   tut-install`/examples/ejb/interceptor/`

2. **To compile the source files and package the application, use the following command:**

   ```
   ant
   ```

   This command calls the `default` target, which builds and packages the application into a WAR file, `interceptor.war`, located in the `dist` directory.

3. **To deploy and run the application using Ant, use the following command:**

   ```
   ant run
   ```

   This command deploys and runs the `interceptor` example, opening a web browser page to `http://localhost:8080/interceptor/`.

4. **Type a name into the form and select Submit.**

   The name will be converted to lowercase by the method interceptor defined in the `HelloInterceptor` class.