

# Creating Resource Adapters with J2EE Connector Architecture 1.5

---



THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.  
4150 Network Circle  
Santa Clara, CA 95054 USA  
1-800-555-9SUN

Trademark Information: <http://www.sun.com/suntrademarks/>

Java, J2EE, J2SE, J2ME, JavaMail, Java Naming and Directory Interface, and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries.

# Creating Resource Adapters with J2EE Connector Architecture 1.5

---

By Alejandro E. Murillo and Binod P. G.  
August 31, 2004

## Contents:

- *Implementing the MailConnector-RA* on page 2
    - Basic Components of the MailConnector-RA
    - Connector Contracts
  - *Managing the Resource Adapter Life Cycle* on page 5
    - Starting the Resource Adapter
    - Stopping the Resource Adapter
  - *Implementing the Outbound Resource Adapter* on page 6
    - Managing Outbound Resource Adapter Connections
    - Configuring Outbound Messaging
  - *Delivering Messages to Message-driven Beans* on page 13
    - Implementing Message Inflow
    - Going Behind the Scenes
    - Configuring Inbound Message Flow Support
    - Recovering from a Crash
  - *Implementing Administered Objects* on page 21
  - *Appendix A: MailConnector-RA Implementation Details* on page 22
  - *References* on page 23
- 

The Java 2, Enterprise Edition 1.4 (J2EE) platform enhances the *J2EE Connector Architecture 1.5 Specification* with many new features, described below. In this article, we use some of them to create a resource adapter (RA) that provides synchronous and asynchronous access to email servers from client components of a J2EE 1.4 application server.

Among the new Connector Architecture (Connector) features are the mechanisms that support inbound RAs. An inbound RA enables a J2EE server to process asynchronous requests (usually messages) coming from external Enterprise Information Systems (EIS). The RA described here allows message-driven beans (MDBs) to receive email (`javax.mail.Message`) messages.

New in Connector 1.5:

- Extensions to the existing Outbound Connection Management contracts
- Message Inflow and Transaction Inflow contracts
- Support for administered objects
- Life Cycle and Work Management contracts

In addition to the new features in the Connector Architecture, the *Enterprise JavaBeans Specification, version 2.1* (EJB 2.1 specification) also has new features that enable MDBs to accept messages from any messaging system that could be integrated into a J2EE 1.4 application server using an RA.

TABLE 1 defines the major players in this article.

**TABLE 1** Terms Used in This Article

Term	Definition
MailConnector-RA	An implementation of the Connector specification that supports connectivity to a back-end system that provides email ( <code>javax.mail.Message</code> ) messages
ActivationSpecImpl	The MailConnector-RA-specific implementation of the <code>ActivationSpec</code> class required by Connector 1.5
Application server	Any implementation of the J2EE 1.4 platform

---

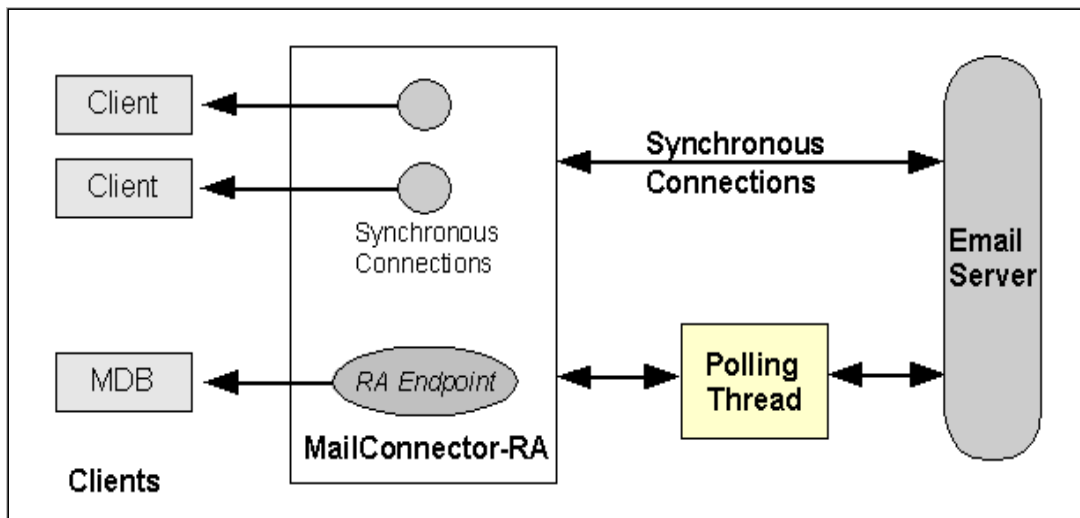
## Implementing the MailConnector-RA

The MailConnector-RA serves as an intermediary between the application server and any email server. The MailConnector-RA exposes custom interfaces related to JavaMail that illustrate how to implement an RA. These interfaces do not represent any major messaging API such as the Java Message Service (JMS) API. The basic features provided by these custom interfaces are

- Support for synchronous query of email servers (outbound RA)
- Support for delivery of JavaMail (`javax.mail.Message`) messages to MDBs (inbound RA)

Synchronous query of email servers is implemented by means of the Connector's Connection Management contracts. In simpler words, clients that want to use the MailConnector-RA to synchronously interact with an email server can look up a connection factory associated with this RA, use it to create a connection, and then use the interface provided by that connection to query the email server for new messages. The outbound APIs of the MailConnector-RA can be accessed from any of the J2EE containers except the application client. They do not provide any transactional support in these containers. See "Implementing the Outbound Resource Adapter" on page 6 for an in-depth description.

Delivering JavaMail messages to MDBs or asynchronous messaging is implemented with the features described in the Message Inflow contract. See "Delivering Messages to Message-driven Beans" on page 13 for the implementation steps. To assist in the implementation of asynchronous messaging, we use the capabilities for scheduling worker threads from an RA (Work Management APIs) to obtain a thread which works as a bridge between the MailConnector-RA and the email servers. FIGURE 1 shows the basic relationships among client components, the MailConnector-RA, the worker (polling) thread, and an email server.



**FIGURE 1** Relationship Between the Client Components and the MailConnector-RA

## Basic Components of the MailConnector-RA

Two basic components of an RA are the main class implementing the `javax.resource.spi.ResourceAdapter` interface and the RA deployment descriptor. The main class of the MailConnector-RA is `ResourceAdapterImpl`. It also implements `java.io.Serializable` because this class operates as a JavaBeans component, as required by the Connector 1.5 model.

The deployment descriptor of the MailConnector-RA, `ra.xml`, is used by the application server to determine which components of the RA implement the required interfaces. This article provides relevant sections of the `ra.xml` file to illustrate how this configuration or mapping was done for the MailConnector-RA. For instance, CODE EXAMPLE 1 shows the portion of `ra.xml` that indicates that the MailConnector-RA is a resource adapter based on Connector 1.5 and that the `ResourceAdapterImpl` class implements `javax.resource.spi.ResourceAdapter`.

**CODE EXAMPLE 1** Snapshot of the MailConnector-RA Deployment Descriptor: `ra.xml`

```
<spec-version>1.5</spec-version>
<resourceadapter>
  <resourceadapter-class>
samples.connectors.mailconnector.ra.inbound.ResourceAdapterImpl
  </resourceadapter-class>
  ...
</resourceadapter>
```

## Connector Contracts

TABLE 2 lists the Connector 1.5 contracts that implement the MailConnector-RA.

**TABLE 2** Use of Connector Contracts

Connector Contract	Connector 1.5 Specification	MailConnector-RA Features
Life Cycle Management contract	Chapter 5, "Lifecycle Management"	Connector core
Connection Management contracts	Chapter 6, "Connection Management," and Chapter 7, "Transaction Management"	Providing messages to clients in a synchronous way
Security contract	Chapter 8, "Security Contract"	Authenticating users
Message Inflow contracts	Chapter 11, "Message Inflow"	Supplying email messages to MDBs
Transaction Inflow contracts	Chapter 11, "Message Inflow"	Providing correct transactional delivery semantics <sup>1</sup>
Administered objects	Section 12.4.2, "Resource Adapter"	Not applicable

<sup>1</sup> The MailConnector-RA does not implement either of the two possible approaches for supporting transactions on message inflow. See Chapter 14 of the Connector 1.5 Specification, "Transaction Inflow," for more details on how to support such transactions.

---

# Managing the Resource Adapter Life Cycle

Chapter 5 of the Connector 1.5 specification describes the life cycle contracts for an RA.

These are the main stages in the Connector 1.5 life cycle:

- Starting the RA
- Stopping the RA
- Recovering from crashes
- Activating endpoints
- Deactivating endpoints

In this section we describe how the MailConnector-RA approaches the first two stages. Endpoint activation, endpoint deactivation, and crash recovery are described in “Delivering Messages to Message-driven Beans” on page 13.

## Starting the Resource Adapter

When the user deploys an RA, the application server instantiates a copy of the RA main class (the one implementing `javax.resource.spi.ResourceAdapter`) and then invokes its `start` method. The same process may also happen when an application server starts up with an already deployed RA. The `start` method is an opportunity for an RA to execute any initialization procedures and, in most cases, to establish communication with the remote EIS. Some RAs might even start the EIS service at this point. The RA can also get a `BootstrapContext`, which gives access to certain application server resources. For instance, the MailConnector-RA obtains the `WorkManager` object from this context and utilizes it to start a worker thread (called `PollingThread`) that will be used to monitor the email folders specified by the MDBs associated with the RA.

**Method:** `ResourceAdapterImpl.start()`

**Implements:** `javax.resource.spi.ResourceAdapter.start()`

### Processing:

1. Save a copy of the `BootstrapContext` as provided (`javax.resource.spi.BootstrapContext`), which can be used to acquire `Timer`, `WorkManager`, or `XATerminator` objects.
2. Start the mail folders polling thread.
3. Get the RA ready to activate MDB endpoints.

If any of these operations fails, an exception is thrown and the MailConnector-RA does not start.

## Stopping the Resource Adapter

The `stop` method of the `ResourceAdapter` class is called by the application server when it is in the process of shutting down or when the RA is being undeployed. When the `stop` method is called, the RA should perform whatever cleanup is required before it is unloaded from the system. Section 5.3.4 of the Connector 1.5 specification describes this process. The RA can assume that the application server has already shut down applications that are accessing the RA's resources, such as `ManagedConnection` (MC) or `EndpointConsumer` objects.

**Method:** `ResourceAdapterImpl.stop()`

**Implements:** `javax.resource.spi.ResourceAdapter.stop()`

**Processing:** When the `stop` method is called on the MailConnector-RA, the MailConnector-RA removes the internal objects used for supporting message inflow and closes any connections that it has opened. If failures occur, the MailConnector-RA still stops. Stopping the MailConnector-RA does not affect the mail servers.

---

## Implementing the Outbound Resource Adapter

The MailConnector-RA implements the Connection Management contract of the Connector Architecture to allow clients to check for new email messages that arrived to a specific mail folder of a given email server. In version 1.5 of the Connector specification, this is called the outbound RA. Sections 6 and 7 of the Connector specification explain all the details and roles of the various components of these contracts. This section describes the specific classes that were implemented to make the MailConnector-RA compliant with the requirements of these contracts.

The connection management contract requires an RA to provide a connection factory and a connection interface. To fulfill this requirement, the MailConnector-RA provides the custom interfaces `JavaMailConnectionFactory` and `JavaMailConnection`.

An outbound RA can choose to provide transactional support by implementing the transaction management contracts of the Connector specifications. Currently, the MailConnector-RA does not provide transactional support; here, however, are some useful tips that will help you add transactional support to your own RA. First, the J2EE Connector Architecture defines three transaction levels that an RA can provide:

- No support at all
- Support only for local transactions
- Support for global transactions (XA) that are controlled and coordinated by an external transaction manager provided by an application server

Second, to be able to provide XA-based transactional support, the RA must implement the `XAResource` interface defined in the Java Transaction API (JTA). Typically, this interface is implemented by the back-end EIS, and the RA just needs a way to obtain it. Finally, an RA is responsible for maintaining a one-to-one relationship between a `ManagedConnection` and each `XAResource` instance. In a typical EIS, transaction support is usually associated with operations at the connection level. For some messaging systems, such as JMS, the transaction support may not be provided at the connection level, so special considerations must be taken into account to match `ManagedConnection` objects with the objects that provide transactional support in the back-end system.

FIGURE 2 shows the relationships among some of the most important classes required by the Connector contracts to implement an outbound RA. For simplicity, we do not include the possible application server connection pooling process in that figure. The following subsections describe how these classes interact.

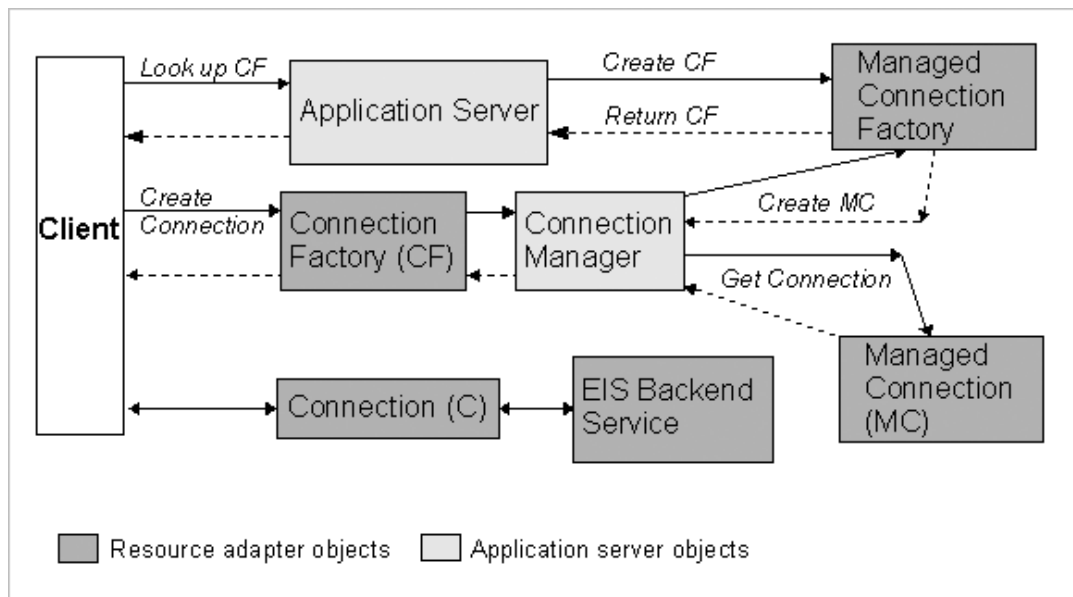


FIGURE 2 Outbound RA: Object Relationships

# Managing Outbound Resource Adapter Connections

The outbound RA must provide the means to create and manage connections. In addition to implementing the interfaces required by the connection management contracts, the MailConnector-RA implements a custom API that allows application components to access a remote email server. TABLE 3 lists all the outbound RA classes and the interfaces of the Connector contract or the custom API that they implement. Classes in bold are discussed in the next section.

**TABLE 3** MailConnector-RA Outbound RA Classes and Interfaces They Implement

Class Name	Interface Implemented
<b>ConnectionManagerImpl</b>	<code>javax.resource.spi.ConnectionManager</code>
<b>ManagedConnectionFactoryImpl</b>	<code>javax.resource.spi.ManagedConnectionFactory</code>
<code>ManagedConnectionMetaDataImpl</code>	<code>javax.resource.spi.ManagedConnectionMetaData</code>
<code>ConnectionRequestInfoImpl</code>	<code>javax.resource.spi.ConnectionRequestInfo</code>
<b>ManagedConnectionImpl</b>	<code>javax.resource.spi.ManagedConnection</code>
<b>JavaMailConnectionFactoryImpl</b>	<code>JavaMailConnectionFactory</code>
<b>JavaMailConnectionImpl</b>	<code>JavaMailConnection</code>
<code>ConnectionMetaDataImpl</code>	<code>javax.resource.spi.ConnectionMetaData</code>

## Outbound Resource Adapter Classes

This section describes in more detail five classes listed in the preceding table, their methods, and their implementation of custom interfaces.

### ■ **ConnectionManagerImpl**

The MailConnector-RA was designed to be used in a managed J2EE environment, so the application server is expected to provide the implementation of `ConnectionManager`. `ConnectionManager` as provided with the MailConnector-RA enables you to use the MailConnector-RA in a nonmanaged environment. `ConnectionManager` as provided with the MailConnector-RA has had limited testing at the time this article was written.

- **ManagedConnectionFactoryImpl**

This class implements the following interface:

```
javax.resource.spi.ManagedConnectionFactory
```

This class allows the application server to create JavaMail connection factories and managed connections associated with the MailConnector-RA. The JavaMail connection factories are created through the `createConnectionFactory` method, and the managed connections are created through the `createManagedConnection` method. The `createConnectionFactory` method of this class returns an object that implements the `JavaMailConnectionFactory` interface, `JavaMailConnectionFactoryImpl`. The `createManagedConnection` method returns a `ManagedConnectionImpl` object.

- **ManagedConnectionImpl**

This class implements the following interface:

```
javax.resource.spi.ManagedConnection
```

This class is used by the application server's connection manager to obtain `JavaMailConnection` objects. When the application server calls the `getConnection` method of a `ManagedConnectionImpl`, a `JavaMailConnectionImpl` object is returned. The Connector contracts allow multiple connections to be associated with the same `ManagedConnection` object. In the MailConnector-RA, an MC is associated with a mail store, and connections are associated with email folders, so connections associated with folders from the same store share an MC.

When an application server calls the `getConnection` method of a `ManagedConnection`, it provides security information through a `ConnectionRequestInfo` object that can be used to authenticate the client.

When `ManagedConnectionImpl` creates a connection (`JavaMailConnectionImpl`), it associates a connection event listener with it. When the application component closes the connection, a `CONNECTION_CLOSED` event is sent back through that event listener to the `ManagedConnection` object so that any pending operation (usually a transaction) is completed.

- **JavaMailConnectionFactoryImpl**

This class implements the following custom interface:

```
samples.connectors.mailconnector.api.JavaMailConnectionFactory
```

A J2EE application component looks up `JavaMailConnectionFactory` to request a connection to an email server. When an application looks up `JavaMailConnectionFactory`, the application server returns a `JavaMailConnectionFactoryImpl` object that was created through the `ManagedConnectionFactoryImpl` object of the MailConnector-RA, as explained above.

## ■ `JavaMailConnectionImpl`

This class implements the following custom interface:

```
samples.connectors.mailconnector.api.JavaMailConnection
```

When the application component requests a connection from `JavaMailConnectionFactory`, it actually receives a `JavaMailConnectionImpl` class. This class is associated with an email folder and is used by the client to check for new email messages in that folder.

## Class Interaction Scenarios

To further illustrate how all the outbound RA classes interact with each other to implement the connection management contracts and the custom interfaces, we present some common scenarios that trigger those interactions.

### *Scenario 1: Registering `JavaMailConnectionFactory` in the JNDI Namespace*

Before (or when) a client application component looks up a `ConnectionFactory` object, that `ConnectionFactory` object must be created and made available in the Java Naming and Directory Interface (JNDI) namespace by means of an administration tool provided by the application server. This tool in turn uses the `ManagedConnectionFactory` object of the RA to create the actual connection factory: `JavaMailConnectionFactory`. A reference to `ConnectionFactory` is bound in the global JNDI namespace.

**Action:** The application server requests a `ConnectionFactory` object from the `MailConnector-RA` and registers a reference to it in the JNDI namespace under the specified name. To achieve this, the application server uses the `createConnectionFactory` method of the `ManagedConnectionFactoryImpl` object of the `MailConnector-RA`. The `createConnectionFactory` method returns a `JavaMailConnectionFactoryImpl` object.

### *Scenario 2: Acquiring a Connection Factory and Creating a Connection*

The application component does a JNDI lookup of a specific `JavaMailConnectionFactory` interface. Then the application component uses that to create a connection (`JavaMailConnection`) to the mail server.

**Actions:** Here's what happens:

1. The application receives a `JavaMailConnectionFactoryImpl` object and invokes the `createConnection` method of that object.

2. The `createConnection` method creates a `ConnectionRequestInfoImpl` object that contains the information required to connect to a given mail server (user name, password, server name, folder name, and so on). This method then delegates the allocation of `Connection` to the application server's `Connection Manager (ConnectionManager)` (CM).
3. The application server's CM uses the appropriate `ManagedConnectionFactory` object of the `MailConnector-RA` to create or find a `Managed Connection (ManagedConnectionImpl)` that will be associated with the connection being created. In a transacted RA, this ensures that the transaction operations involving this connection will be coordinated by the application server through `ManagedConnection`. In the `MailConnector-RA`, `ManagedConnection` is associated with a mail store of the mail server.

In an attempt to reuse `ManagedConnection` objects (connection pooling), the application server issues a `matchManagedConnections` call to the MCF. An MCF can choose to throw `NotSupportedException` if it doesn't support connection pooling. The `MailConnector-RA` supports connection pooling; a match is found when the mail store of the connection being requested matches the one of an existing `ManagedConnection` object.

4. The application server's CM registers a listener with the `ManagedConnection` object so that it can receive notifications of connection events generated by the RA. When closing a connection, the RA generates a `CONNECTION_CLOSED` event.
5. The application server adds the new `ManagedConnection` object to its pool of `ManagedConnection` objects. Then the application server invokes the `getConnection` method of that `ManagedConnection` object to request an application-level connection (`JavaMailConnection`). This method returns a `JavaMailConnectionImpl` object, which eventually will be returned to the application component.
6. The application server obtains an `XAResource` or `LocalTransaction` object from `ManagedConnection`, based on the RA's transaction support. Using one of these objects, the application server manages the transaction. For example, when a transaction starts, the application server executes `XAResource.start()` or `LocalTransaction.begin()` methods. The RA usually delegates this request to the underlying Enterprise Information System (EIS).
7. The client component uses the `JavaMailConnectionImpl` object to query the email server for new messages.

### *Scenario 3: Closing a JavaMailConnection*

The application component completes its interaction with the email server and invokes the `close` method of the `JavaMailConnectionImpl` object.

**Action:** When the `JavaMailConnectionImpl` object receives the `close` message, a `CONNECTION_CLOSED` connection event is created and sent to the associated `ManagedConnection` object.

# Configuring Outbound Messaging

The names of the classes that compose the outbound portion of the MailConnector-RA, as well as their required configuration properties, are specified in the `outbound-resourceadapter` element of the MailConnector-RA deployment descriptor. This element can include several `connection-definition` entries. The MailConnector-RA contains only one `connection-definition` entry, corresponding to the custom connection interface `JavaMailConnection`.

Each `connection-definition` entry contains three main elements:

- The name of the class implementing the `ManagedConnectionFactory` interface, along with any required configuration properties for that class. We have associated five configuration properties with each `ManagedConnectionFactory` object: `folderName`, `serverName`, `userName`, `password`, and `protocol`. The values of these properties are used as the default values to establish a connection when the client requests a connection without providing any parameters.
- The name of the connection factory interface and the name of the class implementing it. The `ConnectionFactory` interface is the custom interface `JavaMailConnectionFactory`, and the class implementing it is `JavaMailConnectionFactoryImpl`.
- The name of the interface that the connection implements, as well as the name of the class implementing it. The connection interface entry is `JavaMailConnection`, and the class implementing it in the MailConnector-RA is `JavaMailConnectionImpl`.

CODE EXAMPLE 2 contains the deployment descriptor portion of the outbound RA.

**CODE EXAMPLE 2** Deployment Descriptor Portion of the Outbound RA (`PACK_PREFIX` is `samples.connectors.mailconnector`)

```
<outbound-resourceadapter>

  <connection-definition>
    <managedconnectionfactory-class>
      PACK_PREFIX.ra.outbound.ManagedConnectionFactoryImpl
    </managedconnectionfactory-class>
    <config-property>
      <config-property-name>
        serverName
      </config-property-name>
      <config-property-type>
        java.lang.String
      </config-property-type>
    </config-property>
    ...
  </connection-definition>
</outbound-resourceadapter>
```

**CODE EXAMPLE 2** Deployment Descriptor Portion of the Outbound RA (`PACK_PREFIX` is `samples.connectors.mailconnector`) (*Continued*)

```
<connectionfactory-interface>
    PACK_PREFIX.api.JavaMailConnectionFactory
</connectionfactory-interface>
<connectionfactory-impl-class>
    PACK_PREFIX.ra.outbound.JavaMailConnectionFactoryImpl
</connectionfactory-impl-class>
<connection-interface>
    PACK_PREFIX.api.JavaMailConnection
</connection-interface>
<connection-impl-class>
    PACK_PREFIX.ra.outbound.JavaMailConnectionImpl
</connection-impl-class>
</connection-definition>
...
</outbound-resourceadapter>
```

---

## Delivering Messages to Message-driven Beans

Chapter 10 of the Connector 1.5 specification describes the Message Inflow Contract. This contract enables the RA to deliver messages to MDBs. The EJB 2.1 specification describes the responsibilities of the container and RA implementer in Chapter 15, “Message-driven Bean Component Contract,” and Chapter 17, “Support for Transactions.” Here we briefly describe some of these semantics, but the EJB and Connector specifications remain the definitive sources of information.

### Implementing Message Inflow

To implement the message inflow mechanism in an RA, you follow these steps:

1. Implement an `ActivationSpec` class:  
`ActivationSpecImpl`.
2. Implement the following method:  
`javax.resource.spi.ResourceAdapter.endpointActivation`

3. Implement the following method:  
`javax.resource.spi.ResourceAdapter.endpointDeactivation`
4. Define the entries corresponding to the inbound connector in the RA deployment descriptor.

Under an XA-capable RA, in addition to the steps above, the RA must be able to provide an XA resource when requested by the application server. Therefore, it is also required to implement the method

`javax.resource.spi.ResourceAdapter.getXAResources.`

TABLE 4 contains the main classes that compose the inbound portion of the MailConnector-RA.

**TABLE 4** Inbound RA Classes

Class Name	Interface
ResourceAdapterImpl	<code>javax.resource.spi.ResourceAdapter</code>
EndpointConsumer	<code>samples.connectors.mailconnector.ra.inbound.EndpointConsumer</code>
ActivationSpecImpl	<code>javax.resource.spi.ActivationSpec</code>

## Going Behind the Scenes

We explain how an RA delivers messages to MDBs by describing what happens when

1. An MDB is deployed;
2. An endpoint is activated in the RA;
3. The RA back end has a message ready for delivery;
4. An endpoint is deactivated in the RA.

## MDB Deployment

According to the EJB 2.1 specification, MDBs can receive messages from virtually any messaging system. Therefore, the deployment descriptor for MDBs now contains a section called `activationConfig`, where the creator of the MDB specifies the configuration information related to message delivery. When an MDB is deployed, the deployer provides this `activationConfig` information to the selected RA so that it can deliver messages to the MDB.

Since every messaging system has its own configuration needs, the `activationConfig` entry of the MDB deployment descriptor is composed of name-value property entries in which each property name matches the name of a

property defined in the `activationSpec` class of the target RA. In this way there is no need to hardwire the names of properties in the MDB deployment descriptor schema.

Because of this, each RA that supports the message inflow contract must implement an `ActivationSpec` (`javax.resource.spi.ActivationSpec`) class. The deployer populates this class by using the MDB deployment descriptor values supplied by the MDB assembler and passes it back to the RA when the MDB is deployed.

An `ActivationSpec` is a JavaBeans component that has the configurable attributes for a specific type of message delivery; different RAs will have different configurable elements:

**Class:** `ActivationSpecImpl`

**Implements:** `javax.resource.spi.ActivationSpec`

Because it is a JavaBeans component, `ActivationSpecImpl` also implements `java.io.Serializable`.

**Processing:** The `ActivationSpecImpl` class is a basic JavaBeans component, with setters and getters that define the properties that must be configured so that the RA implementing the component can deliver messages to MDBs. For the `MailConnector-RA`, all the properties are of type `String`. TABLE 5 contains these properties and the semantics of the values the user is expected to provide.

**TABLE 5** `ActivationSpecImpl` Elements

Element	Description	Default Value
<code>serverName</code>	Name of the machine where the email server is running	None
<code>userName</code>	User name required to access the mail server	None
<code>password</code>	Password for the user name used to access the mail server	None
<code>folderName</code>	Name of the folder to be monitored	Inbox
<code>protocol</code>	Protocol of the mail server (only <code>Imap</code> is supported)	IMAP

The `ActivationSpecImpl` class also implements the `setResourceAdapter` method, which allows the application server to associate the `ActivationSpecImpl` with a specific instance of the RA. The `ActivationSpecImpl` class can implement the optional `ActivationSpec.validate` method. The `validate` method may be called by application server deployment tools to allow additional checking beyond the simple setter methods. For example, the `MailConnector-RA` `validate` method may be implemented to check whether the configuration parameters provided by the MDB assembler correspond to an actual email server. RAs that support the message inflow contract provide an implementation of the `javax.resource.spi.ActivationSpec` for each message listener they support.

This implementation indicates the properties required to configure that listener. If the RA requires no configuration, it does not have to implement the `ActivationSpec` interface.

## Endpoint Activation

When an MDB is deployed, the application server must notify the appropriate RA. This notification happens through the `endpointActivation` method of the main class of the RA. In addition to providing a populated `ActivationSpec` object, as explained in the previous section, in this notification the application server also provides `MessageEndpointFactory` that can be used by the RA to create endpoints. An endpoint is a proxy object that an RA can use to deliver messages to the MDB when they become available.

### Method:

```
ResourceAdapterImpl.endpointActivation(MessageEndpointFactory,  
ActivationSpec)
```

### Implements:

```
javax.resource.spi.ResourceAdapter.endpointActivation(MessageEndpoi  
ntFactory, ActivationSpec)
```

**Processing:** Each time the MailConnector-RA receives an `endpointActivation` call, the MailConnector-RA creates an internal object called `EndpointConsumer`. `EndpointConsumer` serves as a bridge between the MailConnector-RA and the thread that polls the mail servers. `EndpointConsumer` holds the relevant local data needed for message consumption, such as `MessageEndpointFactory`. FIGURE 3 graphically summarizes these steps.

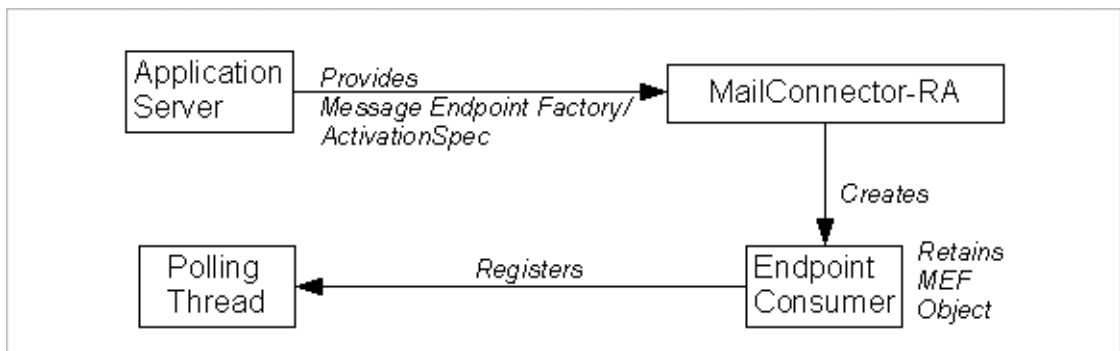


FIGURE 3 Activation of MDB Endpoints

## Message Delivery

When the polling thread detects that new messages have arrived in the mail folder specified by a given MDB, the thread uses the WorkManagement interfaces to spawn a thread (`DeliveryThread`) that will use the corresponding `EndpointConsumer` object to deliver the messages to the MDB with the appropriate endpoint proxy. The burden of message delivery is handled by the `deliverMessage` method of the `EndpointConsumer` object. This method creates an endpoint, using the `MessageEndpointFactory` object provided during endpoint activation, and delivers the message to the endpoint. FIGURE 4 shows the interactions between the components when message delivery occurs. An RA may optionally engage the message delivery in a transaction; this may be important in the requirements for message delivery. See the next section for details about how that can be done.

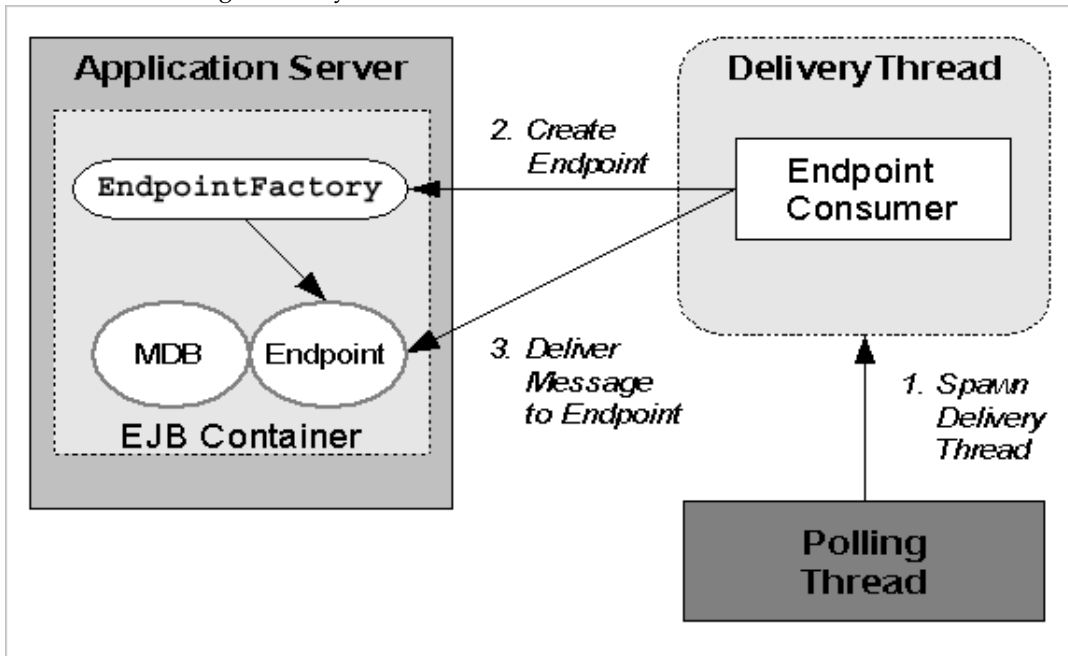


FIGURE 4 Simplified Message Delivery

**Method:** `EndpointConsumer.deliverMessage()`

**Processing:** When this method is called, this is what happens:

1. A `MessageEndpoint` object is created with the `MessageEndpointFactory` object that was supplied during `endpointActivation`.
2. If the RA is transacted, the `MessageEndpoint.beforeDelivery` method is called to allow the application server to enlist the message delivery in a transaction if necessary. The application server is responsible for determining whether transaction enlistment is required.

3. The message is delivered using the appropriate interface.

If the RA is transacted, the `MessageEndpoint.afterDelivery` method is called to allow the application server to commit or roll back any incomplete transactions. CODE EXAMPLE 3 gives a simplified version of the code used in the MailConnector-RA to deliver messages.

**CODE EXAMPLE 3** Simplified Message Delivery (from `EndpointConsumer` Class)

```
XAResource xa = null;

Method onMessage; // method used to deliver message

//Create Endpoint (provide XA resource if transacted)
endpoint = endpointFactory.createEndpoint(xa);

// supply Method object that performs
// message delivery in the bean

//endpoint.beforeDelivery(onMessage);

((samples.connectors.inbound.api.JavaMailMessageListener) endpoint).
onMessage(message);

//endpoint.afterDelivery();
```

### *Notes About Message Delivery, Transactions, and Exceptions*

Connector implementers have two choices for providing transactional support on message inflow:

- They can use the Before/AfterDelivery model.
- They can initiate a transaction in their messaging system and use the Transaction Inflow contracts described in Chapter 17 of the Connector 1.5 specification.

Under either of these alternatives, the `MessageEndpointFactory.isDeliveryTransacted` method can determine if the message delivery associated with the `MessageEndpointFactory` requires transactional support. The MailConnector-RA does not implement either of these models. A given RA implementation will use the model that best matches its needs.

The Before/AfterDelivery model requires that both the `beforeDelivery` and `afterDelivery` methods be called. If the `beforeDelivery` method is called and starts a distributed transaction, but the `afterDelivery` method is never called, the message delivery will be left in an incomplete state.

The EJB 2.1 containers can propagate exceptions on message delivery. This is a change from the EJB 2.0 containers. The `onMessage` method must be capable of handling these exceptions.

## Endpoint Deactivation

When an MDB is undeployed, the application server calls the `endpointDeactivation` method of the main class of the RA to stop the delivery of messages to the MDB.

### Method:

```
ResourceAdapterImpl.endpointDeactivation(MessageEndpointFactory,  
ActivationSpec)
```

### Implements:

```
javax.resource.spi.ResourceAdapter.endpointDeactivation(MessageEndp  
ointFactory, ActivationSpec)
```

**Processing:** When the MailConnector-RA receives a message to deactivate an endpoint, it uses `MessageEndpointFactory` as a key to find the `EndpointConsumer` object that represents this endpoint. The `MessageEndpointFactory` object is unique, while `ActivationSpec` is not.

The MailConnector-RA shuts down message consumption by deregistering `EndpointConsumer` from the polling thread. When this operation completes, the polling thread stops monitoring the email folder associated with `EndpointConsumer` and removes the `EndpointConsumer` object from the list of active consumers.

## Configuring Inbound Message Flow Support

To indicate support for the use of the message inflow contract, the MailConnector-RA must specify the following elements in the MailConnector-RA deployment descriptor:

- The name of the message listener that is available to the MDB  
(`samples.connectors.inbound.api.JavaMailMessageListener`)
- The class that implements the `ActivationSpec` interface  
(`samples.connectors.inbound.ra.ActivationSpecImpl`)
- The elements of the `ActivationSpec` implementation that are required. The recommended required elements for this RA are `serverName`, `userName`, `password`, and `mailFolder`.

CODE EXAMPLE 4 lists the MailConnector-RA deployment descriptor.

**CODE EXAMPLE 4** Setting up MessageListener and ActivationSpec (from ra.xml)

```
<inbound-resourceadapter>
  <messageadapter>
    <messagelistener>
      <messagelistener-type>
samples.connectors.inbound.api.JavaMailMessageListener
      </messagelistener-type>
      <activation-spec>
        <activation-spec-class>
samples.connectors.inbound.api.ActivationSpecImpl
        </activation-spec-class>
        <required-config-property>
          <config-property-name>
serverName
          </config-property-name>
        </required-config-property>
        <required-config-property>
          <config-property-name>
userName
          </config-property-name>
        </required-config-property>
        <required-config-property>
          <config-property-name>
password
          </config-property-name>
        </required-config-property>
        <required-config-property>
          <config-property-name>
mailFolder
          </config-property-name>
        </required-config-property>
      </activation-spec>
    </messagelistener>
  </messageadapter>
</inbound-resourceadapter>
```

## Recovering from a Crash

If an application server fails, it may leave some of its transactions in an unresolved state. Therefore, the application server has a mechanism for querying RAs to get the status of transactions. This query takes place on server restart for RAs that support transactions in the delivery of messages. The application server supplies an array of `ActivationSpec` objects that represent the MDBs that were consuming from the RA when the crash occurred.

The RA then returns an array of `XAResource` objects for any `ActivationSpec` objects that had pending transactions. The application server can then use those `XAResource` objects to determine whether to commit or roll back the transaction. The application server assesses what action to take based on the status of the transactions associated with each `XAResource` object and the status of other transactional resources associated with the corresponding MDB.

**Method:** `ResourceAdapterImpl.getXAResources()`

**Implements:** `javax.resource.spi.ResourceAdapter.getXAResources()`

**Processing:** The MailConnector-RA does not support transacted delivery of messages to MDBs, so it returns a null array of `XAResource` objects.

---

## Implementing Administered Objects

A J2EE 1.4 compatible application server provides administration tools to perform various administrative functions, including creating administered objects. These tools can bind the administered objects into the JNDI namespace, where they can be accessed by the application components. An RA can expose administered objects to these tools. To expose an administered object, an RA must do the following:

1. Implement the administered object as a JavaBeans component.
2. Enter the administered object's interface in the RA's deployment descriptor and provide the name of the class that implements that interface.
3. Specify any configurable elements for this object. These are specified as name/type/value tuples.

For instance, the deployment descriptor of an RA for a JMS implementation might include an entry for an administered object like that shown in CODE EXAMPLE 5. This entry indicates that there is an object that implements the `javax.jms.Queue` interface and that can be placed in the JNDI namespace. The object is implemented by `com.sun.jms.QueueImpl`, and has one configurable property, its `Name`. There is no default value for the `Name`. The `QueueImpl` object, a JavaBeans component, has methods to set and get its name and implement the semantics of a JMS queue object.

```

<adminobject>
  <adminobject-interface>
    javax.jms.Queue
  </adminobject-interface>
  <adminobject-class>
    com.sun.jms.QueueImpl
  </adminobject-class>
  <config-property>
    <config-property-name> Name </config-property-name>
    <config-property-type>
      java.lang.String
    </config-property-type>
  </config-property>
</adminobject>

```

## Appendix A: MailConnector-RA Implementation Details

The MailConnector-RA resource adapter is distributed as a sample application for Sun Java System Application Server Platform Edition 8 (Application Server PE 8). The source code of that application resides at *AppServerPE8\_install\_dir/samples/connectors/apps/mailconnector/*. You can modify, assemble, and deploy the application by using the *asant* command in Application Server PE 8 to invoke the corresponding *ant* target for each of the tasks. The targets are defined in the main *build.xml* file of the sample application. For detailed instructions, see the documentation at *AppServerPE8\_install\_dir/samples/connectors/apps/mailconnector/docs/index.html*.

TABLE 6 shows the main classes that comprise the MailConnector-RA.

TABLE 6 MailConnector-RA Implementation Classes

Class Name	Interface
<i>General</i>	
ResourceAdapterImpl	javax.resource.spi.ResourceAdapter
<i>Synchronous Messaging</i>	
ConnectionManagerImpl	javax.resource.spi.ConnectionManager
ConnectionRequestInfoImpl	javax.resource.spi.ConnectionRequestInfo

**TABLE 6** MailConnector-RA Implementation Classes (Continued)

Class Name	Interface
ConnectionSpecImpl	javax.resource.cci.ConnectionSpec
JavaMailConnectionEventListener	javax.resource.spi.ConnectionEventListener
JavaMailConnectionFactoryImpl	JavaMailConnectionFactory
JavaMailConnectionImpl	JavaMailConnection
ManagedConnectionFactoryImpl	javax.resource.spi.ManagedConnectionFactory
ManagedConnectionImpl	javax.resource.spi.ManagedConnection
<b><i>Inbound Messaging</i></b>	
ActivationSpecImpl	javax.resource.spi.ActivationSpec
EndpointConsumer	None
<b><i>Worker Threads</i></b>	
PollingThread	javax.resource.spi.work.Work
DeliveryThread	javax.resource.spi.work.Work
<b><i>Utility</i></b>	
Util	Support functions, mostly for PasswordCredential

## References

The following specifications are of interest to the Connector developer:

- *Enterprise JavaBeans Specification, version 2.1*  
<http://java.sun.com/products/ejb/docs.html>
- *Java 2 Platform Enterprise Edition Specification, version 1.4*  
<http://java.sun.com/j2ee/1.4/>
- *J2EE Connector Architecture Specification, version 1.5*  
<http://java.sun.com/j2ee/connector/download.html>

See also the J2EE Connector Architecture site at

<http://java.sun.com/j2ee/connector/>.

For details on Application Server PE 8, visit the following pages:

- Main page  
[http://wwws.sun.com/software/products/appsrvr\\_pe/](http://wwws.sun.com/software/products/appsrvr_pe/)
- Developer home  
<http://developers.sun.com/prodtech/appserver/>

---

## About the Authors

Alejandro E. Murillo, a member of the Sun Java Partner Engineering Enterprise team and the Sun Java System Application Server development team, has been with Sun for over five years. Previously, he performed R&D activities on high-performance and distributed computing for numerically intensive applications. Alejandro has a M.Sc. degree in Applied Mathematics and Computer Science.

Binod P. G., who joined Sun four years ago, is the engineer lead for Connectors, JDBC, and JMS on the J2EE and Sun Java Application Server development teams, with a focus on integrations. Before joining Sun, Binod worked for many years on non-J2EE and J2EE platforms, concentrating on distributed computing.