16. JMS Message-driven beans

The Enterprise JavaBeans (EJB) 3.2 specification states[6] that a message-driven bean that implements the `javax.jms.MessageListener` interface is a JMS message-driven bean and defines[7] a set of standard activation properties for such beans.

This chapter extends the EJB specification to define an additional type of JMS message-driven bean that does not implement the `javax.jms.MessageListener` interface but instead implements the javax.jms.MessageDrivenBean interface.

16.1. Classic JMS MDBs

A message-driven bean class that implements the `javax.jms.MessageListener` interface is referred to in this specification as a "classic" JMS MDB. This interface defines a single callback method, `onMessage`.

A classic JMS MDB is configured, like MDBs in general, using activation properties. These may be specified either using the `ActivationConfigProperty` annotation or by the `<activation-config-property>` deployment descriptor element.

A set of standard activation properties for configuring classic JMS MDBs is defined in the EJB 3.2 specification, section 5.4.17 "JMS Message-Driven Beans"

Note to reviewers: It is hoped to move those parts of the EJB 3.2 specification that define "classic" JMS MDBs (mainly section 5.4.17) into this chapter so that both types of JMS MDB are defined in the same place.

EDR1 RC1

[6] Enterprise JavaBeans 3.2 specification, section 5.4.2 "The Required Message Listener Interface".

[7] Enterprise JavaBeans 3.2 specification, section 5.4.17 "JMS Message-Driven Beans".

Here is an example of a classic JMS MDB:

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(
    propertyName = "destinationLookup",
    propertyValue = "java:global/requestQueue"),
  @ActivationConfigProperty(
    propertyName = "destinationType",
    propertyValue = "javax.jms.Queue")
 })
public class MyMessageBean implements MessageListener {
  public void onMessage(Message message){
   ...
  }
}
```

16.2. Flexible JMS MDBs

A message-driven bean class that implements the `javax.jms.JMSMessageDrivenBean` interface is referred to in this specification as a "flexible" JMS MDB. This interface defines no methods. The bean class may have any number of callback methods, each of which must be specified using one of the three method annotations `JMSQueueListener`, `JMSNonDurableTopicListener` or `JMSDurableTopicListener`.

A flexible JMS MDB does not need to be configured using activation properties. Instead it can be configured using attributes of the `JMSQueueListener`, `JMSNonDurableTopicListener` or `JMSDurableTopicListener` method annotations or by using any number of `JMSListenerProperty` method annotations.

Here is an example of a flexible MDB with a single callback method:

```
@MessageDriven
public class MyMessageBean implements JMSMessageDrivenBean {
  @JMSQueueListener(destinationLookup="java:global/requestQueue")
  public void myMessageCallback(Message message) {
    ...
  }
}
```

Note to reviewers: The need to implement a no-methods marker interface (`JMSMessageDrivenBean`) is a requirement of EJB 3.2 It is hoped to remove this requirement before this specification is released. In that case a flexible JMS MDB would not need to implement any interface. All that would be needed would be to identify the callback methods.

Note to reviewers: Although the text above states that the bean class may have any number of callback methods this is still open for review as it may cause unnecessary complexity without any clear benefit.

16.2.1. Specifying the callback methods

A flexible JMS MDB may have any number of callback methods. Each callback method will be treated as representing a separate consumer, and so

EDR1C1

may specify a different queue or topic, connection factory, subscription name, message selector etc.

Each callback method must be specified using one of the three annotations `@JMSQueueListener`, `@JMSNonDurableTopicListener` or `@JMSDurableTopicListener`.

16.2.1.1. JMSQueueListener

The `@JMSQueueListener` annotation is used to specify that the callback method should be used to deliver messages from a queue.The `@JMSQueueListener` annotation has the following elements:

- The `destinationLookup` element may be used to specify the lookup name of the `Queue` from which messages will be received. It corresponds to the classic JMS MDB activation property `destinationLookup`.
- The `connectionFactoryLookup` element may be used to specify the lookup name of the `ConnectionFactory` that will be used to connect to the JMS provider. It corresponds to the classic JMS MDB activation property `connectionFactoryLookup`.
- The `messageSelector` element may be used to specify the message selector that will be used. It corresponds to the classic JMS MDB activation property `messageSelector`.
- The `acknowledge` element may be used to specify the acknowledgement mode that will be used if the MDB is not configured to use container-managed transactions. It may be set to either `@JMSQueueListener.Mode.AUTO_ACKNOWLEDGE` or `@JMSQueueListener.Mode.DUPS_OK_ACKNOWLEDGE`. It corresponds to the classic JMS MDB activation property `acknowledgeMode`.

Here is an example of a flexible MDB which defines one callback method that is annotated with `JMSQueueListener` and which sets all four elements:

```
@MessageDriven
public class MyMessageBean implements JMSMessageDrivenBean {
  @JMSQueueListener(
    destinationLookup="java:global/requestQueue",
    connectionFactoryLookup="java:global/connectionFactory",
    messageSelector="JMSType = 'car' AND colour = 'pink'",
    acknowledge=JMSQueueListener.Mode.DUPS_OK_ACKNOWLEDGE
  )
  public void myMessageCallback(Message message) {
    ...
  }
}
```

16.2.1.2. JMSNonDurableTopicListener

The `@JMSNonDurableTopicListener` annotation is used to specify that the callback method should be used to deliver messages from a non-durable subscription on a topic.The `@JMSNonDurableTopicListener` annotation has the following elements:

- The `destinationLookup` element may be used to specify the lookup name of the `Topic` from which messages will be received. It

EDR RC1

corresponds to the classic JMS MDB activation property `destinationLookup`.

- The `connectionFactoryLookup` element may be used to specify the lookup name of the `ConnectionFactory` that will be used to connect to the JMS provider. It corresponds to the classic JMS MDB activation property `connectionFactoryLookup`.
- The `messageSelector` element may be used to specify the message selector that will be used. It corresponds to the classic JMS MDB activation property `messageSelector`.
- The `acknowledge` element may be used to specify the acknowledgement mode that will be used if the MDB is not configured to use container-managed transactions. It may be set to either `JMSNonDurableTopicListener.Mode.AUTO_ACKNOWLEDGE` or `JMSNonDurableTopicListener.Mode.DUPS_OK_ACKNOWLEDGE`. It corresponds to the classic JMS MDB activation property `acknowledgeMode`.

Here is an example of a flexible MDB which defines one callback method that is annotated with `@JMSNonDurableListener` and which sets all four elements:

```
@MessageDriven
public class MyMessageBean implements JMSMessageDrivenBean {
  @JMSNonDurableTopicListener(
    destinationLookup="java:global/requestQueue",
    connectionFactoryLookup="java:global/connectionFactory",
    messageSelector="JMSType = 'car' AND colour = 'pink'",
    acknowledge=
      JMSNonDurableTopicListener.Mode.DUPS_OK_ACKNOWLEDGE
  )
  public void myMessageCallback(Message message) {
    ...
  }
}
```

16.2.1.3. JMSDurableTopicListener

The `@JMSDurableTopicListener` annotation is used to specify that the callback method should be used to deliver messages from a non-durable subscription on a topic.The `@JMSDurableTopicListener` annotation has the following elements:

- The `destinationLookup` element may be used to specify the lookup name of the `Topic` from which messages will be received. It corresponds to the classic JMS MDB activation property `destinationLookup`.
- The `connectionFactoryLookup` element may be used to specify the lookup name of the `ConnectionFactory` that will be used to connect to the JMS provider. It corresponds to the classic JMS MDB activation property `connectionFactoryLookup`.
- The `subscriptionName` element may be used to specify the name of the durable subscription that will be used. It corresponds to the classic JMS MDB activation property `subscriptionName`.

EDR1 RC1

- The `clientId` element may be used to specify the JMS client identifier that will be used to connect to the JMS provider. It corresponds to the classic JMS MDB activation property `clientId`.
- The `messageSelector` element may be used to specify the message selector that will be used. It corresponds to the classic JMS MDB activation property `messageSelector`.
- The `acknowledge` element may be used to specify the acknowledgement mode that will be used if the MDB is not configured to use container-managed transactions. It may be set to either `JMSDurableTopicListener.Mode.AUTO_ACKNOWLEDGE` or `JMSDurableTopicListener.Mode.DUPS_OK_ACKNOWLEDGE`. It corresponds to the classic JMS MDB activation property `acknowledgeMode`.

Here is an example of a flexible MDB which defines one callback method that is annotated with `@JMSDurableListener` and which sets all six elements:

```
@MessageDriven
public class MyMessageBean implements JMSMessageDrivenBean {
  @JMSDurableTopicListener(
    destinationLookup="java:global/requestQueue",
    connectionFactoryLookup="java:global/connectionFactory",
    subscriptionName="mySubName",
    clientId="myClientId",
    messageSelector="JMSType = 'car' AND colour = 'pink'",
    acknowledge=JMSQueueListener.Mode.DUPS_OK_ACKNOWLEDGE
  )
  public void myMessageCallback(Message message) {
    ...
  }
}
```

16.2.1.4. JMSListenerProperty

The `@JMSListenerProperty` method annotation may be used to set an arbitrary activation property on a callback method. Multiple `@JMSListenerProperty` annotations may be used to set multiple properties.

Applications that use the `@JMSListenerProperty` annotation to set non-standard activation properties may not be portable.

The following example shows a flexible MDB that uses JMSListenerProperty annotations to set the properties `foo1` and `foo2`:

EDR1 RC1

```
@MessageDriven
public class MyMessageBean implements JMSMessageDrivenBean {

  @JMSQueueListener(
    destinationLookup="java:global/requestQueue")
  @JMSListenerProperty(name="foo1", value="bar1")
  @JMSListenerProperty(name="foo2", value="bar2")
  public void myMessageCallback(Message message) {
    ...
  }
}
```

16.2.2. Callback methods

Each callback method on a flexible MDB may have any number of parameters. Depending on the parameter type and any parameter annotations, each parameter will be set to the message, the message body, a message header or a message property.

Each callback method must return void.

Callback methods may be inherited.

Callback methods may throw checked exceptions and `RuntimeException`s. See section 16.3 "Exceptions thrown by callback methods" below.

16.2.2.1. Message parameters

If the callback method has a parameter of type `Message`, `TextMessage`, `StreamMessage`, `BytesMessage`, `MapMessage` or `ObjectMessage` then the parameter will be set to the message being delivered.

The message must be capable of being assigned to the specified type. If the parameter is of type `TextMessage`, `StreamMessage`, `BytesMessage`, `MapMessage`, `ObjectMessage` and the message is not of the specified type then the callback method will not be called and the message will be handled as described in section 16.2.3 "When a message parameter cannot be set" below

In the following example, all the messages being delivered are expected to be of type `BytesMessage`. The callback method has one parameter of type `BytesMessage` which will be set to the message being delivered:

```
@MessageDriven
public class MyMessageBean implements JMSMessageDrivenBean {

  @JMSQueueListener(destinationLookup="java:global/requestQueue")
  public void myMessageCallback(BytesMessage message) {
    ...
  }
}
```

16.2.2.2. Message body parameters

If the callback method has a parameter of any type other than `Message`, `TextMessage`, `StreamMessage`, `BytesMessage`, `MapMessage` or `ObjectMessage`, and the parameter is not annotated with `MessageHeader` or `MessageProperty`, then the application server or resource adapter will set it to the message body.

EDR1 RC1

The application server or resource adapter will obtain the message body by calling the method `getBody(Class<T> c)` on the message object, where `c` will be set to the parameter's type. The API documentation for this method defines which parameter type must be used for a given message type.

The message body must be capable of being assigned to the parameter's type. If the call to `getBody` throws a `MessageFormatException` then the callback method will not be called and the message will be handled as described in section 16.2.3 "When a message parameter cannot be set" below.

In the following example, all the messages being delivered are expected to be of type `TextMessage`. The callback method has one parameter of type `String` which will be set to the message body.

```
@MessageDriven
public class MyMessageBean implements JMSMessageDrivenBean {

  @JMSQueueListener(destinationLookup="java:global/requestQueue")
  public void myMessageCallback(String textBody) {
    ...
  }
}
```

16.2.2.3. Message header parameters

The `@MessageHeader` annotation may be used to specify that a parameter must be set to a specified message header.

The `@MessageHeader` annotation has one element, `value`, which must be set to an enumerated constant of type `MessageHeader.Header` that specifies which header value is required.

The parameter must have a type appropriate to the specified header.

Table 16.1 below lists the available headers and the parameter type that must be used for each.

If the parameter type is not appropriate for the specified header then deployment must fail.

EDR1 RC1

Table 16.1 Message header parameter annotations types

Annotation	Parameter type
`@MessageHeader(Header.JMSCorrelationID)`	`String`
`@MessageHeader(Header.JMSCorrelationIDAsBytes)`	`Byte[]`
`@MessageHeader(Header.JMSDeliveryMode)`	`Integer` or `int`
`@MessageHeader(Header.JMSDeliveryTime)`	`Long` or `long`
`@MessageHeader(Header.JMSDestination)`	`Destination`
`@MessageHeader(Header.JMSExpiration)`	`Long` or `long`
`@MessageHeader(Header.JMSMessageID)`	`String`
`@MessageHeader(Header.JMSPriority)`	`Integer` or `int`
`@MessageHeader(Header.JMSRedelivered)`	`Boolean` or `boolean`
`@MessageHeader(Header.JMSReplyTo)`	`Destination`
`@MessageHeader(Header.JMSTimestamp)`	`Long` or `long`
`@MessageHeader(Header.JMSType)`	`String`

In the following example, the callback method has one parameter of type `Message` which will be set to the message itself, and one parameter of type `boolean` which will be set to the value of the `JMSRedelivered` header:

```
@MessageDriven
public class MyMessageBean implements JMSMessageDrivenBean {

  @JMSQueueListener(destinationLookup="java:global/requestQueue")
  public void myMessageCallback(Message message,
    @MessageHeader(MessageHeader.Header.JMSRedelivered)
      boolean redeliveredFlag) {
    ...
  }
}
```

16.2.2.4. Message property parameters

The `@MessageProperty` annotation may be used to specify that a parameter must be set to a specified message property.

The `@MessageProperty` annotation has one element, `value`, which must be set to the property name.

The parameter must have a type appropriate to the specified property.

EDR1 RC1

The method that will be used by the application server or resource adapter to obtain the property value will depend on the parameter type. Table 16.2 below lists the methods that will be used.

If the method parameter is not one of the types listed then deployment must fail.

If the method parameter is one of the types listed but the message property cannot be converted to the specified type using the conversion rules defined in the table, then the callback method will not be called and the message will be handled as described in section 16.2.3 "When a message parameter cannot be set" below.

Table 16.2 How parameters annotated with `@MessageProperty("foo")` will be set

Parameter type	**Set to**
`boolean`	`message.getBooleanProperty("foo")`
`Boolean`	`(Boolean)message.getObjectProperty("foo")`
`byte`	`message.getByteProperty("foo")`
`Byte`	`(Byte)message.getObjectProperty("foo")`
`short`	`message.getShortProperty("foo")`
`Short`	`(Short)message.getObjectProperty("foo")`
`integer`	`message.getIntProperty("foo")`
`Integer`	`(Integer)message.getObjectProperty("foo")`
`long`	`message.getLongProperty("foo")`
`Long`	`(Long)message.getObjectProperty("foo")`
`float`	`message.getFloatProperty("foo")`
`Float`	`(Float)message.getObjectProperty("foo")`
`double`	`message.getFloatProperty("foo")`
`Double`	`(Double)message.getObjectProperty("foo")`
`String`	`message.getStringProperty("foo")`

Note that only `getObjectProperty` and `getStringObject` can return a null value. This means that if the specified property is not set then the callback method will only be called if the parameter is an object type (`Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` or `String`), in which case the parameter will be set to `null`. If the parameter has a primitive type (`boolean`, `byte`, `short`, `integer`, `long`, `float` or `double`) then this will cause a conversion error and be handled as described above.

In the following example, the callback method has one parameter of type `Message` which will be set to the message itself, and one parameter of type `String` which will be set to the value of the message property `foo`, which must also be of type `String`.

EDR1RC1

```
@MessageDriven
public class MyMessageBean implements JMSMessageDrivenBean {

  @JMSQueueListener(destinationLookup="java:global/requestQueue")
  public void myMessageCallback(Message message,
    @MessageProperty("foo") String fooValue) {
    ...
  }
}
```

16.2.3. When a message parameter cannot be set

There are a number of cases in which a message parameter cannot be set because it has an inappropriate type for the particular message being delivered:

- Message parameters (see section 16.2.2.1): The parameter is of type `TextMessage`, `StreamMessage`, `BytesMessage`, `MapMessage`, `ObjectMessage` but the message is of a different type.
- Message body parameters (see section 16.2.2.2): The parameter is of any type other than `Message`, `TextMessage`, `StreamMessage`, `BytesMessage`, `MapMessage` or `ObjectMessage`, and the parameter is not annotated with `MessageHeader` or `MessageProperty`, but using `getBody` to assign the message body to the parameter's type has thrown an exception.
- Message property parameters (see section 16.2.2.4): The parameter has been annotated with `@MessageProperty` and is one of the valid primitive types for holding message properties, but using the appropriate method to obtain a property of the parameter's type has thrown an exception.
- Message property parameters (see section 16.2.2.4): The parameter has been annotated with `@MessageProperty` and is one of the valid object types for holding message properties, but the message property cannot be cast to the parameter's type.

When such an error occurs any container-managed transaction must not be started and the callback method must not be called.

The message will be redelivered in the same way as if bean-managed transaction demarcation was being used and the callback method had thrown an exception.

Note to reviewers: Essentially this section is saying that a conversion error should be handled in the same way as a RuntimeException is now. JMS 2.0 does not define behaviour such as redeliver delay, redelivery delay and dead message queues and this section deliberately avoids doing so.

However it is hoped to extend this section to define redelivery delay, redelivery counts and dead message queues before final release.

16.2.4. Use of flexible JMS MDB annotations on classic JMS MDBs

The `JMSQueueListener`, `JMSNonDurableTopicListener`, `JMSDurableTopicListener`, `JMSListenerProperty` or `JMSListenerProperties` annotations must not be used on a classic JMS MDB. If any of these annotations are used on a MDB that implements `javax.jms.MessageListener` then deployment must fail.

EDR1 RC1

16.3. Exceptions thrown by callback methods

The `onMessage` method of a classic JMS MDB (one that implements `MessageListener`) may throw `RuntimeException`s.

A callback method of a flexible MDB may throw checked exceptions or `RuntimeException`s.

All exceptions thrown by message callback methods must be handled by the container as defined in the EJB 3.2 specification section 9.3.4 "Exceptions thrown from Message-Driven Bean Message Listener methods". This defines whether or not any container-managed transaction is committed or rolled back by the container. It also defines whether or not the MDB instance is discarded, whether or not the exception is required to be logged, and what exception is re-thrown to the resource adapter (if a resource adapter is being used).

If a resource adapter is being used it must catch any checked exceptions or `RuntimeException`s thrown by the callback method.

If a message is being delivered to the callback method of a MDB using container-managed transaction demarcation, and the resource adapter had called the `beforeDelivery` method on the `javax.resource.spi.endpoint.MessageEndpoint` prior to invoking the callback method, then it must call the `afterDelivery` method afterwards even if the callback method threw a checked exception or `RuntimeException`. This ensures that the container-managed transaction is rolled back or committed by the container as required by the EJB specification.

If a message is being delivered to the callback method of a MDB, and auto-acknowledge or dups-ok-acknowledge mode is being used, and the callback method throws a checked exception or a `RuntimeException`, then the message will be automatically redelivered.

The number of times a JMS provider will redeliver the same message before giving up is provider-dependent. The `JMSRedelivered` message header field will be set, and the `JMSXDeliveryCount` message property incremented, for a message redelivered under these circumstances.

> ***Note to reviewers:*** Essentially this section is saying that exceptions should be handled in the same way as they are now. JMS 2.0 does not define behaviour such as redeliver delay, redelivery delay and dead message queues and this section deliberately avoids doing so.
>
> However it is hoped to extend this section to define redelivery delay, redelivery counts and dead message queues before final reelase.

EDR1 PFD