

Tutorial: Using Message-Driven Beans

This tutorial describes how the Financial Brokerage Service (FBS 10g) sample application uses message-driven beans (MDB) to process trade orders, emails, and alerts.

This tutorial assumes that you are familiar with the FBS 10g and have installed and configured the required software as described in [About the Financial Brokerage Service](#).

Contents

1. [Concepts](#)
2. [Design](#)
3. [Required Software](#)
4. [Setup](#)
5. [Implementation](#)
6. [Resources](#)
7. [Feedback](#)





Concepts

In general, applications and software components communicate via three mechanisms: method calls, events, and messages. A method call is a direct invocation of a specific operation performed by a specific component. For example, the method call `database.deleteRecord(key)` tells a specific database object to delete the record identified by the parameter `key`.

Communication via events is less direct, because there is at least one intermediate layer between the initiator of an operation and the component that performs the operation. The most familiar events are those generated by a user interacting with objects (buttons, menus, etc.) in an application's user interface. In such cases, the user action (clicking a button, choosing a menu item, etc.) is not an end in itself—the action is interpreted by an event listener which in turn dispatches it to the appropriate destination. For example, a user interface could include a Record menu that provides Insert and Delete options and is implemented to listen for `menuChoice` events. When a user chooses an item from this menu, the system generates a `menuChoice` event and sends it to the Record menu component, which parses the event and invokes a method, for example, `database.insertRecord` or `database.deleteRecord`, based on event parameter values. Note that in this typical scenario, the event listener needs to know which object and which method to invoke. To implement an event listener, developers must decide up front which events to handle and what actions to take when an event is received.

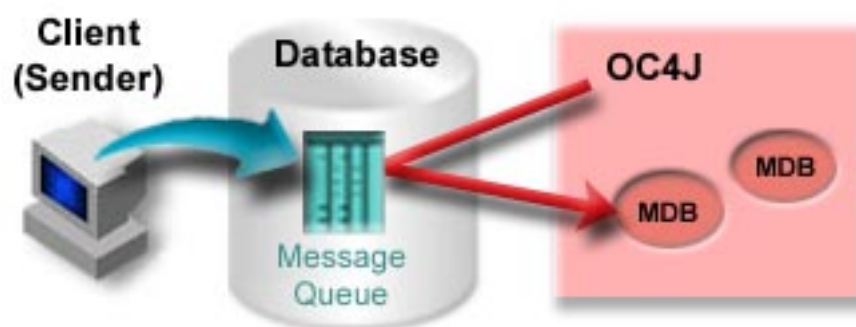
Messaging is the least direct mechanism of the three. The sender must know how to format the message and where to send it, and the recipient must know the message format and where to get it, but neither component needs to know anything about the other. J2EE components and applications perform messaging tasks using the Java Messaging Service (JMS) API. JMS provides two programming models:

- Point-to-point (queue), where each message is sent to one recipient.
- Publish and subscribe (topics), where messages are broadcast to one or more registered listeners.

JMS queues and topics are bound to the JNDI environment and made available to J2EE applications.

The EJB specification uses the point-to-point JMS model as the basis for message-driven beans (MDBs). In essence, an MDB is a JMS listener with additional functionality provided by the EJB container. Senders place messages in a queue, and the container processes the queue by dispatching each message to the appropriate MDB (the deployment descriptor for each MDB declares the queue it will listen to). Processing is asynchronous, meaning that the container does not have to wait for any given MDB to respond before performing the next operation.

Oracle Containers for J2EE (OC4J) provides support for MDB in conjunction with Oracle JMS (Java Messaging Service), which is installed and configured on an Oracle database. Within this database, messages are managed using Oracle Advanced Queuing (AQ). The following figure gives an overview of the process.



Oracle AQ is implemented as an integral part of the Oracle database management system. Advanced Queues are relational database tables, enhanced to support queuing operations like enqueue and dequeue. Messages are stored as rows in a table, and you can use standard SQL to:

- Access a message's payload, control information, and history.
- Optimize access using technology such as indexes.

The [Design](#) section describes how OTN developers decided where and how to use MDBs in the FBS 10g. Coding details are discussed in the [Implementation](#) section.





Design

By using message-driven beans, you get the features of a JMS listener plus functions performed by the container. Compared to method calls and events, MDB is a good choice for an application that:

- Requires components to communicate without knowing about each other.
- Must function even if all components are not running at the same time.
- Must be able to send messages and continue processing without waiting for a response.

In the FBS 10g, alerts can be generated by numerous application modules, and OTN developers used MDBs to centralize the alert mechanism. By removing the tight coupling between the execution of a trade transaction and sending back the user response, the application can serve multiple users without compromising response times. MDBs allow asynchronous processing and thereby reduce the wait time for the customers.

All trade requests (both buy and sell) are sent as messages. When a message is posted to the queue, the application invokes an MDB which extracts the trade details and sends them to the exchange for processing. Once the processing is done, the result is mailed to the customer. Since posting a message is not a costly operation, the customers get split-second response times.





Required Software

This tutorial presents several code examples. If you want to study them in context, download and install the [FBS 10g source code](#).

If you also want to build and run the FBS 10g, you will need the software listed in the [Required Software](#) section of *About the Financial Brokerage Service*.





Setup



To configure your system to build and run the FBS 10g sample application, see the [Setup](#) section of *About the Financial Brokerage Service*.





Implementation

This section describes how OTN developers used message-driven beans (MDBs) to implement an alert system in the FBS 10g. The FBS 10g also uses MDBs to manage trade notifications, but because the implementation is essentially the same, it is not covered here. The [Design](#) section describes how OTN developers decided where to use MDBs.

There's relatively little Java code to write. The key is to get the right information into the right configuration files.

The Java class for an MDB must implement the `MessageDrivenBean` and `MessageListener` interfaces. The key method is `onMessage`, which is invoked by the container. The following listing shows the implementation of `oracle.otnsample.ibfbs.admin.ejb.AlertQueue`.

```
public void onMessage(javax.jms.Message message) {
    try {
        // Initialize context for lookup
        Context ctx = new InitialContext();
        // Lookup for the mailservice bean in the JNDI tree
        MailServiceHome mailServiceHome =
            (MailServiceHome) ctx.lookup("MailService");
        // Get an instance of mailservice
        MailService mailService = mailServiceHome.create();
        TextMessage txtMessage = (TextMessage) message;
        // Based on the Alert Mode, send alert mail to either Email id or
        // Mobile mail box
        if (txtMessage.getStringProperty("alertMode").equals("M")) {
            mailService.sendMail(txtMessage.getStringProperty("mobile"),
                "alerts@fbs.com",
                "Stock Price Alert",
                getTextMsg(
                    txtMessage.getStringProperty("accountNumber"),
                    txtMessage.getStringProperty("name"),
                    txtMessage.getStringProperty("symbol"),
                    txtMessage.getStringProperty("minlimit"),
```

```

        txtMessage.getStringProperty("maxlimit"),
        txtMessage.getFloatProperty("rate")));
    } else {
        mailService.sendMail(txtMessage.getStringProperty("email"),
            "alerts@fbs.com",
            "Stock Price Alert",
            getHTMLMsg(
                txtMessage.getStringProperty("accountNumber"),
                txtMessage.getStringProperty("name"),
                txtMessage.getStringProperty("symbol"),
                txtMessage.getStringProperty("minlimit"),
                txtMessage.getStringProperty("maxlimit"),
                txtMessage.getFloatProperty("rate")));
    }
} catch (Throwable ex) { // Trap errors
    ex.printStackTrace();
}
}

```

The following listings come from the configuration file <fbs_home>\etc\ejb-jar.xml. The first listing below names the MDB, identifies the class that implements it, and specifies a destination class for associated messages.

```

<message-driven>
  <ejb-name>AlertMessageBean</ejb-name>
  <ejb-class>oracle.otnsamples.ibfbs.admin.ejb.AlertQueue</ejb-class>
  <transaction-type>Container</transaction-type>
  <message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
  </message-driven-destination>
</message-driven>

```

The listing below specifies an entity named Alerts, which is referred to later in the code.

```

<entity>
  <ejb-name>Alerts</ejb-name>
  <local-
home>oracle.otnsamples.ibfbs.usermanagement.ejb.AlertsHomeLocal</local-home>
  <local>oracle.otnsamples.ibfbs.usermanagement.ejb.AlertsLocal</local>

```

```

<ejb-class>oracle.otnsamples.ibfbs.usermanagement.ejb.AlertsBean</ejb-class>
<persistence-type>Container</persistence-type>
<prim-key-class>java.lang.Integer</prim-key-class>
<reentrant>False</reentrant>
<cmp-version>2.x</cmp-version>
<abstract-schema-name>Alerts</abstract-schema-name>
<cmp-field><field-name>id</field-name></cmp-field>
<cmp-field><field-name>accountNumber</field-name></cmp-field>
<cmp-field><field-name>symbol</field-name></cmp-field>
<cmp-field><field-name>minLimit</field-name></cmp-field>
<cmp-field><field-name>maxLimit</field-name></cmp-field>
<primkey-field>id</primkey-field>
</entity>

```

The following listing uses the Alert entity defined above.

```

<container-transaction>
  <method>
    <ejb-name>UserManagementSessionBean</ejb-name>
    <method-name>*</method-name>
  </method>
  <method>
    <ejb-name>UserAccount</ejb-name>
    <method-name>*</method-name>
  </method>
  <method>
    <ejb-name>Alerts</ejb-name>
    <method-name>*</method-name>
  </method>
  <method>
    <ejb-name>Preferences</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>

```

The following listing from `<fbs_home>\etc\orion-ejb-jar.xml` specifies MDB deployment information.

```

<message-driven-deployment
    name="AlertMessageBean"
    destination-location="java:comp/resource/ojms/Queues/IBFBS
10g.ALERTQUEUE"
    connection-factory-
location="java:comp/resource/ojms/QueueConnectionFactories/AlertQueueConnectionFactory"
/>

```

The lines below come from `Connection.properties`. They tell the container where to find the `AlertQueue` and `AlertQueueConnFactory` classes.

```

# Lookup name of AlertQueue
ALERTQUEUEUENAME=java:comp/resource/ojms/Queues/IBFBS 10g.ALERTQUEUE
# Lookup name of AlertQueueConnectionFactory
ALERTQUEUECONNFACORYNAME=java:comp/resource/ojms/QueueConnectionFactories/AlertQueueConnectionFactory

```

The following code from `<fbs_home>\etc\orion-application.xml` provides context data for the message queue.

```

<resource-provider class="oracle.jms.OjmsContext" name="ojms">
    <description> OJMS/AQ </description>
    <property name="url"
        value="jdbc:oracle:thin:IBFBS 10g/IBFBS
10g@www.foo.com:1522:ora9idb">
    </property>
</resource-provider>

```

The SQL code below comes from `<fbs_home>\sql\ibfbs.sql`. It sets up Advanced Queuing (AQ) users, tables, and other resources for FBS 10g alerts.

```

CREATE USER IBFBS IDENTIFIED BY IBFBS
/
ALTER USER IBFBS DEFAULT TABLESPACE USERS
/
GRANT CONNECT, RESOURCE,AQ_ADMINISTRATOR_ROLE TO IBFBS
/
GRANT EXECUTE ON DBMS_AQ TO IBFBS
/

```

```
GRANT EXECUTE ON DBMS_AQADM TO IBFBS
/
EXECUTE DBMS_AQADM.Grant_System_Privilege('ENQUEUE_ANY','IBFBS',FALSE)
/
EXECUTE DBMS_AQADM.Grant_System_Privilege('DEQUEUE_ANY','IBFBS',FALSE)
/
EXECUTE DBMS_AQADM.Grant_System_Privilege('MANAGE_ANY','IBFBS',TRUE)
/
EXECUTE DBMS_AQADM.Grant_Type_Access('IBFBS')
```

...

PROMPT Creating Alert Queue

```
BEGIN
  dbms_aqadm.create_queue_table
  (
    queue_table=>'AlertQueueTable',
    queue_payload_type=>'SYS.AQ$_JMS_TEXT_MESSAGE',
    multiple_consumers => false
  );
```

END;

/

```
BEGIN
  dbms_aqadm.create_queue (
    queue_name    => 'AlertQueue',
    queue_table   => 'AlertQueueTable');
```

END;

/

PROMPT Starting Alert Queue ...

```
BEGIN
  DBMS_AQADM.START_QUEUE(queue_name => 'AlertQueue');
```

END;

/





Resources

This tutorial is part of a series, [EJB 2.1 Techniques](#), based on the Financial Brokerage Service (FBS 10g) sample application. Following are links to resources that can help you understand and apply the concepts and techniques presented in this tutorial.

Resource	URL
Message-Driven Beans	http://otn.oracle.com/tech/java/oc4j/doc_library/902/ejb/mdb.htm
Java Messaging Service	http://otn.oracle.com/docs/products/ias/doc_library/90200doc_otn/web.902/a95879/lms.htm
EJB Overview	http://otn.oracle.com/tech/java/oc4j/doc_library/902/ejb/overview.htm
EJB Downloads and Specifications	http://java.sun.com/products/ejb/docs.html
Oracle Products	http://otn.oracle.com/products/





Feedback

If you have questions or comments about this tutorial, you can:

- Post a message in the [OTN Sample Code discussion forum](#).
OTN developers and other experts monitor the forum.
- Send email to the author. <mailto:Robert.Hall@oracle.com>

If you have suggestions or ideas for future tutorials, please send email to:

- <mailto:Raghavan.Sarathy@oracle.com>

