

Code generation with XDoclet

“It’s so beautifully arranged on the plate—you know someone’s fingers have been all over it.”

—*Julia Child*

Developing EJBs today entails the creation of a bean class, multiple interfaces, possible helper data access classes, primary key classes, and even lookup utility classes. In addition, each new bean requires changes and additions to the XML descriptor. With each new bean developed, the possibility for out-of-synch files increases. For example, a change to an EJB local interface would require similar changes to the remote interface, the implementation class, utility classes (data access object, value objects, etc.), a facade class, and the XML deployment descriptor.

Now, take that single change and multiply it across all your EJBs. The final result is that a development team must manage the synchronization of multiple files across multiple beans in order to keep the most basic tasks, such as compilation and deployment, working successfully. Without a doubt, experienced developers can ultimately handle this file management problem, but you must also consider the development time consumed by the trivial task of repeating code from interface files to implementation files. Time is a valuable commodity, and most projects struggle to have enough in all phases.

Increasingly, developers are turning to tools that automate much of bean development. For instance, more and more tools provide support for descriptor generation and manipulation. Rather than cover the multitude of IDE tools, we've chosen to cover XDoclet, an open-source tool that is rapidly gaining acceptance. Simple and easy, XDoclet saves you time and energy while generating excellent code.

In this chapter, we present the most common uses of XDoclet, including the following tasks:

- Generating EJB interfaces
- Adding JNDI names to your home interfaces
- Maintaining the XML descriptor
- Creating value objects for entity beans
- Creating primary key classes
- Customizing XDoclet tags with Ant properties
- Generating a utility object
- Adding security roles to the bean source
- Creating method permission XML
- Generating finder methods
- Creating the XML for `ejbSelect` methods
- Adding home methods to home interfaces

- Generating entity relation XML
- Generating XML descriptors for message-driven EJBs

An XDoclet appetizer

XDoclet requires the use of Ant, a build tool from Apache, which you can find at <http://ant.apache.org>. This chapter assumes that you have a working knowledge of Ant, including writing build.xml files for compiling and packaging your EJB files. If you have not used Ant for a build system, you can find specific recipes for those tasks in chapter 9.

Specifically, XDoclet relies on the Ant task `<ejbdoclet/>`. Once inserted into the build.xml, the `<ejbdoclet/>` task allows you to specify subtasks for file generation, method construction, and more. Tasks execute a section of code within Ant. Ant contains many predefined tasks for such jobs as generating documentation and compiling, but it lets you build your own tasks as well. In fact, the `<ejbdoclet/>` task is a custom task that executes certain code in the XDoclet library.

For this book, we used XDoclet beta version 1.2. Table 2.1 lists the JAR file dependencies needed by this version of XDoclet, as well as the URL for their download. The JAR files listed in table 2.1 must be in the classpath of the `<ejbdoclet/>` task added to your build.xml file before you execute the `<ejbdoclet/>` Ant task. (Some of the JAR files will not be needed if you don't use certain features of the 1.2 version of XDoclet.)

Table 2.1 The JAR file dependencies for the 1.2 version of XDoclet. These jars should be placed in the `<ejbdoclet/>` Ant task classpath in order for you to use XDoclet 1.2.

Framework/application	Needed JAR files	URL
Ant 1.5	ant.jar	http://jakarta.apache.org/ant/
Log4j 1.13	log4j-1.1.3-jar	http://jakarta.apache.org/log4j/
Commons logging	commons-logging-1.0.jar	http://jakarta.apache.org/log4j/
XML APIs	xml-apis-2.0.2.jar	http://xml.apache.org/xerces2-j/
Velocity	velocity-1.4-dev.jar	http://jakarta.apache.org/velocity/index.html
JUnit	junit-3.7.jar	http://junit.org

After your environment is set up, you need to perform only three steps to generate code:

- 1 Add the necessary XDoclet tags to your source files (similar to JavaDoc comment tags).
- 2 Modify the `<ejbdoclet/>` task in the build.xml file to generate the desired files.
- 3 Execute the Ant task using a command similar to `ant.ejbdoclet`.

With three steps, XDoclet can make your EJB development more efficient and streamlined.

Before moving on to the actual recipes, let's quickly examine the pieces of the `<ejbdoclet/>` task contained in the build.xml file. The task basically contains three sections: setup, source file selection, and subtask declaration. The following XML is only a portion of a larger build.xml file and shows an example `<ejbdoclet/>` task definition:

```

<target name="ejbdoclet" depends="init">
  <taskdef name="ejbdoclet" classname="xdoclet.modules.ejb.EjbDocletTask">
    <classpath>
      <fileset dir="${xdoclet.lib.dir}" includes="*.jar"/>
    </classpath>
  </taskdef>
  <ejbdoclet destdir="${src}" ejbspec="2.0" >
    <fileset dir="${src}">
      <include name="**/*Bean.java" />
    </fileset>
    <remoteinterface/>
    <homeinterface/>
    <localhomeinterface/>
    <homeinterface/>
    <deploymentdescriptor destdir="${build}/ejb/META-INF" />
  </ejbdoclet>
</target>

```

Defines the new `<ejbdoclet/>` task ❶

❷ Sets up the task

Adds subtasks to generate code ❸

- ❶ The first section of the target declares the task and provides the name of the implementing class that will perform the functions required of the task. The task definition is also responsible for setting up the classpath for the task. In this case, we have the necessary XDoclet JAR files (see table 2.1) in the XDoclet lib directory. The property `xdoclet.lib.dir` should be defined earlier in the build.xml file containing this target.

- ② Next, the `<ejbdoclet/>` tag is started by specifying the source directory, the generated files destination directory, and the EJB specification version that the build.xml file should use. After starting the task, the next section defines the set of source files the build.xml file should examine for possible source-generation tags. Using the `<fileset/>` tag, not only can you specify which files to include, but you can also exclude files. For instance, the sample shows a file set of Java source files that end with `*Bean.java`.
- ③ Before closing the `<ejbdoclet/>` task, you can specify subtasks that actually perform the source examination and generation of code. The declaration in this sample generates the remote, home, local home, and local interfaces for bean classes with the appropriate XDoclet tags.
XDoclet provides many more features than shown in this simple example. This chapter contains recipes that examine the most useful or commonly used features of XDoclet. We highlight the XDoclet JavaDoc tags in bold to distinguish them from the remaining code. In addition, we show generated code where appropriate.

We hope this chapter will encourage you to look further into XDoclet for your EJB development. Refer to the XDoclet website at <http://XDoclet.sourceforge.net> for downloads, examples, and more documentation. In addition, you can refer to the Ant website (<http://jakarta.apache.org>) to learn more about using Ant or creating build.xml files. For more information about creating an Ant task, or using existing tasks, check the Ant documentation at <http://jakarta.apache.org/ant/manual/index.html>, or check out the excellent book *Java Development with Ant*, from Manning Publications by Erik Hatcher and Steve Loughran.

2.1 Generating home, remote, local, and local home interfaces

◆ **Problem**

You want to generate the EJB home and remote interfaces.

◆ **Background**

While developing bean classes and interfaces, you must spend too much time keeping your interface file in synch with the implementation class. After developing the bean class, you would like to generate all the necessary interfaces for deployment. This includes the remote, home, local, and local home interfaces.

Likewise, after any modifications to the bean class, you want the interfaces to be updated similarly, and in the correct way for the specific interface.

◆ **Recipe**

To generate the interfaces (home, local home, remote, and local), you must do two things. You need to add the appropriate XDoclet tags to the bean implementation class, and then add the correct subtasks to the `<ejbdoclet/>` task in your `build.xml` file. This recipe covers adding create and business methods to the correct interfaces. Other methods, such as finder methods, are covered in later recipes.

A session bean example

The session bean example in listing 2.1 illustrates how to document a bean class in order to generate both remote and local interfaces. The XDoclet tags are shown in bold. Notice that assigned values for the tags always use double quotes.

Listing 2.1 UserBean.java

```
package ch2;

import javax.ejb.*;

/**
 * @ejb.bean type="Stateful" | Declares the
 * view-type="both" | bean attributes
 */
public class UserBean implements SessionBean
{
    private String name = null;

    public UserBean() {}
    public void setSessionContext( SessionContext context) {}

    /**
     * @ejb.create-method ← Indicates
     * | which type of
     * | methods to
     * | generate
     */
    public void create() {}
    public void ejbCreate() {}
    public void ejbRemove() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}

    /**
     * @ejb.interface-method ←
     */
    public void setName( String value )
    {
        this.name = value;
    }
}
```

```

    }
    /**
     * @ejb.interface-method
     */
    public String getName()
    {
        return name;
    }
}

```

An entity bean example

The entity bean example in listing 2.2 illustrates the same tags as the previous session bean example.

Listing 2.2 DataBean.java

```

package ch2;

import javax.ejb.*;
/**
 * @ejb.bean type="CMP"          | Declares the bean
 *           view-type="both"   | attributes
 */
public abstract class DataBean implements EntityBean
{
    public void setEntityContext( EntityContext context) {}
    public void unsetEntityContext( ) {}

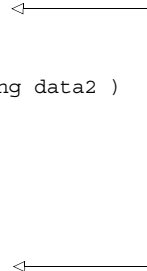
    public void ejbRemove() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}

    /**
     * @ejb.create-method
     *
     */
    public void ejbCreateData( String data1, String data2 )
    {
        setBigData( data1 );
        setSmallData( data2 );
    }

    /**
     * @ejb.interface-method
     *
     */
    public void getAllData()
    {

```

Indicates
which type of
methods to
generate



```

        return getBigData() + " " + getSmallData();
    }

    /**
     * @ejb.interface-method
     */
    public abstract String getBigData();

    /**
     * @ejb.interface-method
     */
    public abstract String getSmallData();

    /**
     * @ejb.interface-method
     */
    public abstract void setBigData( String data );

    /**
     * @ejb.interface-method
     */
    public abstract void setSmallData( String data );
}

```

Modifying the build.xml

As shown in listing 2.3, you add the `<ejbdoclet/>` Ant task in your `build.xml` file with the appropriate subtasks that will actually perform the code generation.

Listing 2.3 Sample Build.xml

```

<target name="ejbdoclet" depends="init">
  <taskdef name="ejbdoclet" classname="xdoclet.modules.ejb.EjbDocletTask">
    <classpath>
      <fileset dir="${xdoclet.lib.dir}" includes="*.jar"/>
    </classpath>
  </taskdef>

  <ejbdoclet destdir="${src}" ejbspec="2.0" >
    <fileset dir="${src}">
      <include name="**/*Bean.java" />
    </fileset>

    <remoteinterface pattern="{0}Remote"/>
    <homeinterface/>
    <localinterface/>
    <localhomeinterface/>

  </ejbdoclet>
</target>

```

**Add subtasks for
code generation**

Notice that this target is only part of a larger build.xml file. In your build.xml, you need to configure the `src` and `xdoclet.lib.dir` properties for the task to work.

◆ **Discussion**

Examining each bean source, you should first notice the class-level JavaDoc comments. The class-level JavaDoc contains the XDoclet tags that describe the bean and specifies the interfaces that should be generated. The `@ejb.bean` tag describes the EJB defined in the source file. It contains two properties—`type` and `view-type`—that are involved in the interface generation. The `type` property describes the type of this EJB; it can be `Stateful`, `Stateless`, `CMP`, or `BMP`. The code generator needs this information in order to properly provide super interfaces for the generated interfaces. The second property, `view-type`, indicates which interfaces should be generated. Its possible values are `local`, `remote`, or `both`. By specifying `both`, you ensure that all four interfaces will be produced.

However, these two properties only help XDoclet to generate the interface declaration; you still must describe the methods that go into each interface. To do this, you need to make use of the `@ejb.interface-method` and `@ejb.create-method` XDoclet tags. As shown in the source, these tags are used to mark bean methods for declaration in the appropriate interfaces. Create methods are routed to the home interfaces, and interface methods are declared in the remote and local interfaces. Table 2.2 summarizes the tags that generate methods into the interfaces.

Table 2.2 Other tags used to specify methods for EJB interfaces

Tag	Description
<code>@ejb.interface-method</code>	Declares a method to be a business method
<code>@ejb.create-method</code>	Declares a method to be an <code>ejbCreate</code> method
<code>@ejb.home-method</code>	Declares a method to be a home method
<code>@ejb.finder</code>	Used to define a finder method for the home and local home interfaces
<code>@ejb.select</code>	Declares a method to be an <code>ejbSelect</code> method
<code>@ejb.pk-field</code>	When used properly, creates a <code>findByPrimaryKey</code> method in the home interface (see recipe 2.5)

Two method types noticeably absent from this discussion are `finder` and `select` methods for entity beans. We show these two method types in greater detail in later recipes in this chapter.

Finally, the additional subtasks must be specified in the `<ejbdoclet/>` task itself. As you can see in the recipe, we add tasks to generate all four interfaces for the beans. Indeed, all four will be generated because we also specified the `view-type` as `both`.

In addition, by default XDoclet will add a component name and JNDI name for both the local home and home interfaces as a `public final static` member variable. You can use the variable to make your client code more maintainable. By default, the names correspond to the fully qualified name of the bean class (using `/` instead of `.`).

Rather than show all four generated interfaces for each bean, we just show the local interfaces for each. For the session bean, the `getName()` and `setName()` methods will be in the local and remote interfaces. The session bean's home and local home interfaces will contain a `create()` method. Listing 2.4 contains the session bean's generated entire remote interface (comments and all).

Listing 2.4 Generated by XDoclet, User.java

```
/*
 * Generated by XDoclet - Do not edit!
 */
package ch2;

/**
 * Remote interface for ch2.User.
 */
public interface User
    extends javax.ejb.EJBObject
{
    public java.lang.String getName( )
        throws java.rmi.RemoteException;

    public void setName( java.lang.String value )
        throws java.rmi.RemoteException;
}
```

The entity bean's home and local home interfaces will contain a `findByPrimaryKey()` method. Its remote and local interface will contain `getFirstName()`, `setFirstName()`, `getLastName()`, `setLastName()`, and `getName()`. Listing 2.5 contains the entity bean's generated remote interface

Listing 2.5 Generated by XDoclet, Data.java

```
/*
 * Generated by XDoclet - Do not edit!
 */
package ch2;

/**
 * Remote interface for ch2.Data.
 */
public interface Data
    extends javax.ejb.EJBObject
{
    public void getAllData( )
        throws java.rmi.RemoteException;

    public java.lang.String getBigData( )
        throws java.rmi.RemoteException;

    public java.lang.String getSmallData( )
        throws java.rmi.RemoteException;

    public void setBigData( java.lang.String data )
        throws java.rmi.RemoteException;

    public void setSmallData( java.lang.String data )
        throws java.rmi.RemoteException;
}
```

◆ **See also**

- 2.2—Adding and customizing the JNDI name for the home interface
- 2.5—Generating a primary key class
- 2.11—Generating finder methods for entity home interfaces

2.2 Adding and customizing the JNDI name for the home interface

◆ **Problem**

You want a good way to store the JNDI name of a bean for easy retrieval to aid in bean lookup.

◆ **Background**

You can use XDoclet to add a `public static final` member variable to the home interfaces that it generates to store the JNDI name of the bean. Without customization, it provides a default value for this name. By specifying the JNDI name in the home interface, you can modify it without changing your bean lookup code.

◆ **Recipe**

Use the recipe shown in recipe 2.1 (listing 2.4) to generate the home interface. However, change the class-level JavaDoc to look like the following and specify the JNDI name (the changes are shown in bold):

```
/**
 * @ejb.bean type="Stateful"
 *      jndi-name="ejb/UserBean"
 *      local-jndi-name="ejb/UserBeanLocal"
 *      view-type="both"
 */
public class UserBean implements SessionBean
{
```

No changes need to be made to the `build.xml` file from the target shown in recipe 2.1.

◆ **Discussion**

By including the JNDI lookup name as a `public static final` member variable in the home interface, you give your code a permanent, safe way of discovering the JNDI name for EJB lookup. Using this method, you don't have to hardcode a name in the lookup implementation. The resulting home interface has the following lines added to it:

```
public static final String
    COMP_NAME="comp/env/ejb/ch2/User";
public static final String JNDI_NAME="ejb/UserBean";
```

The resulting local home interface contains a different name (as specified in the bean source):

```
public static final String
    COMP_NAME="java:comp/env/ejb/ch2/UserLocal";
public static final String JNDI_NAME="ejb/UserBeanLocal";
```

Without customization, XDoclet will enter names using the package name of the bean class. For instance, the `UserBean` JNDI name would have been `ch2/UserBean`.

When looking up an EJB home interface via JNDI, you normally would use code similar to the following:

```
InitialContext ctx = new InitialContext();
UserHome home = (UserHome) ctx.lookup( "ejb/UserBean" );
```

By adding the JNDI name to the home interface, your code can change to something like this:

```
InitialContext ctx = new InitialContext();
UserHome home = (UserHome) ctx.lookup( UserHome.JNDI_NAME );
```

◆ **See also**

2.1—Generating home, remote, local, and local home interfaces

2.3 Keeping your EJB deployment descriptor current

◆ **Problem**

You want to generate the EJB deployment descriptor and update it as the EJB source files change.

◆ **Background**

When developing EJBs, you have a multitude of changes to the bean class that affect the final deployment descriptor of the bean. Even if you generate the deployment descriptor once, you may have to change it each time you alter a bean class, interface, or persistent feature. In addition, changes to security roles, method permissions, and EJB relationships require you to modify the XML descriptor. Generating the deployment XML is only part of an important task. XDoclet will help you maintain this file by updating it as your beans change and develop.

◆ **Recipe**

To have XDoclet generate your deployment descriptor, add the `<deploymentdescriptor/>` subtask to your `<ejbdoclet/>` task in the `build.xml` file. (See the section “An XDoclet appetizer” at the beginning of this chapter for information about XDoclet setup and the `build.xml` file.) The `<ejbdoclet/>` task shown in listing 2.6 uses the descriptor subtask.

Listing 2.6 Sample Build.xml

```

<target name="ejbdoclet" depends="init">
  <taskdef name="ejbdoclet"
    classname="xdoclet.modules.ejb.EjbDocletTask">
    <classpath>
      <fileset dir="${xdoclet.lib.dir}" includes="*.jar"/>
    </classpath>
  </taskdef>

  <ejbdoclet destdir="${src}" ejbspec="2.0" >

    <fileset dir="${src}">
      <include name="**/*Bean.java" />
    </fileset>

    <deploymentdescriptor destdir="${build}/ejb/META-INF" />

  </ejbdoclet>
</target>

```

Adds the subtask for
XML generation

Let's examine the class declaration for a session bean (used from recipes 2.1 and 2.2). However, this time we also include the bean name (shown in bold):

```

/**
 * @ejb.bean type="Stateful"
 *   name="UserBean"
 *   jndi-name="ejb/UserBean"
 *   local-jndi-name="ejb/UserBeanLocal"
 *   view-type="both"
 */
public class UserBean implements SessionBean

```

XDoclet uses this information to build the basic deployment descriptor for each bean. The XML section shown in listing 2.7 is what XDoclet generated for this bean (we have shown only the portion of the XML that contains the UserBean).

Listing 2.7 Deployment descriptor generated by XDoclet

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
  JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar >

  <description><![CDATA[No Description.]]></description>
  <display-name>Generated by XDoclet</display-name>

  <enterprise-beans>

    <!-- Session Beans -->
    <session >

```

```
<description><![CDATA[]]></description>
<ejb-name>UserBean</ejb-name>
<home>ch2.UserBeanHome</home>
<remote>ch2.UserBean</remote>
<local-home>ch2.UserBeanLocalHome</local-home>
<local>ch2.UserBeanLocal</local>
<ejb-class>ch2.UserBean</ejb-class>
<session-type>Stateful</session-type>
<transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
```

◆ **Discussion**

The `<deploymentdescriptor/>` subtask tells XDoclet to generate the deployment descriptor for the beans it has examined from the file set described in the `<fileset/>` tag. XDoclet will also take care of including any other additions in the descriptor along with the actual bean description. As long as you keep this subtask in your `<ejbdoclet/>` task, XDoclet will generate or regenerate the XML deployment descriptor for each modified bean class in the file set. As you can tell, all you need to provide is the destination directory for the XML file.

XDoclet can also generate the numerous other pieces of the `ejb-jar.xml` file for your beans. This includes security roles, method permission, and related EJBs. As you add more XDoclet JavaDoc comments to your bean source files, more generated XML will appear. Many of the additional tags are covered in other recipes.

◆ **See also**

2.8—Generating vendor-specific deployment descriptors

2.4 **Creating value objects for your entity beans**

◆ **Problem**

You want to generate a value object for your entity beans.

◆ **Background**

An accepted practice for improving EJB application performance and for separating client, business, and data layers is to make use of value objects for entity beans.

Value objects create a decoupled view of entity beans and also shield clients from back-end code changes. This class can represent the bean in every way and be passed back to the client for a read-only snapshot of the entity data.

Creating value objects for entity beans adds one more file to the list of multiple files that developers must create for each bean. As with other generated files, XDoclet will help you maintain this file with changes as your beans change.

◆ **Recipe**

Use the `@ejb.value-object` tag in the class-level JavaDoc for entity beans needing a value object. For example, the section of the entity bean `ItemBean` source shown in listing 2.8 uses this tag. The new tag is shown in bold; reference the previous recipes for information about the others. Don't worry about the tags `@ejb.pk-field` and `@ejb.persistence` for now; we cover those in the next recipe.

Listing 2.8 `ItemBean.java`

```
package ch2;

import javax.ejb.*;
/**
 * @ejb.bean type="CMP"
 *          name="ItemBean"
 *          jndi-name="ejb/ItemBean"
 *          view-type="both"
 *
 * @ejb.value-object
 *
 */
public abstract class ItemBean implements EntityBean
{
    public void setEntityContext( EntityContext context) {}
    public void unsetEntityContext( ) {}

    public void ejbRemove() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}

    /**
     * @ejb.create-method
     */
    public void ejbCreate( String id )
    {
        setID( id );
    }
}
/**
```



```
    * @ejb.interface-method
    * @ejb.persistence
    * @ejb.pk-field
    */
    public abstract String getID();

    /**
    * @ejb.interface-method
    */
    public abstract String getType();

    /**
    * @ejb.interface-method
    */
    public abstract String setType();
}
```

In addition, you need to add the subtask `<valueobject/>` to your `build.xml` file in order for XDoclet to know it should generate the new class.

◆ **Discussion**

Listing 2.9 contains the generated value object class (including comments), reformatted for this chapter.

Listing 2.9 ItemBeanValue.java

```
/*
 * Generated file - Do not edit!
 */
package ch2;

import java.util.*;

/**
 * Value object for ItemBean.
 *
 */
public class ItemBeanValue extends java.lang.Object
    implements java.io.Serializable
{
    private java.lang.String id;
    private boolean idHasBeenSet = false;
    private java.lang.String type;
    private boolean typeHasBeenSet = false;

    private ch2.ItemBeanPK pk;

    public ItemBeanValue()
    {
        pk = new ch2.ItemBeanPK();
    }
}
```

```
    }
    public ItemBeanValue( java.lang.String iD,
                          java.lang.String type )
    {
        this.iD = iD;
        iDHasBeenSet = true;
        this.type = type;
        typeHasBeenSet = true;
        pk = new ch2.ItemBeanPK(this.getID());
    }

    //TODO Cloneable is better than this !
    public ItemBeanValue( ItemBeanValue otherValue )
    {
        this.iD = otherValue.iD;
        iDHasBeenSet = true;
        this.type = otherValue.type;
        typeHasBeenSet = true;

        pk = new ch2.ItemBeanPK(this.getID());
    }

    public ch2.ItemBeanPK getPrimaryKey()
    {
        return pk;
    }

    public void setPrimaryKey( ch2.ItemBeanPK pk )
    {
        // it's also nice to update PK object - just in case
        // somebody would ask for it later...
        this.pk = pk;
    }

    public java.lang.String getID()
    {
        return this.iD;
    }

    public java.lang.String getType()
    {
        return this.type;
    }

    public String toString()
    {
        StringBuffer str = new StringBuffer("{}");


        str.append("iD=" + getID() + " " + "type=" + getType());
        str.append('}');

        return(str.toString());
    }
}
```

**Initializes the
value object
with data**

**Provides
read-only
access**

```
/**
 * A Value object has an identity if its
 * attributes making its Primary Key
 * have all been set. One object without identity
 * is never equal to any
 * other object.
 *
 * @return true if this instance have an identity.
 */
protected boolean hasIdentity()
{
    boolean ret = true;
    ret = ret && idHasBeenSet;
    return ret;
}

public boolean equals(Object other)  Implements  
equality testing
{
    if ( ! hasIdentity() ) return false;
    if (other instanceof ItemBeanValue)
    {
        ItemBeanValue that = (ItemBeanValue) other;
        if ( ! that.hasIdentity() ) return false;
        boolean lEquals = true;
        if( this.id == null )
        {
            lEquals = lEquals && ( that.id == null );
        }
        else
        {
            lEquals = lEquals && this.id.equals( that.id );
        }

        lEquals = lEquals && isIdentical(that);
        return lEquals;
    }
    else
    {
        return false;
    }
}

public boolean isIdentical(Object other)
{
    if (other instanceof ItemBeanValue)
    {
        ItemBeanValue that = (ItemBeanValue) other;
        boolean lEquals = true;
        if( this.type == null )
        {
            lEquals = lEquals && ( that.type == null );
        }
    }
}
```

```
        }
        else
        {
            lEquals = lEquals && this.type.equals( that.type );
        }

        return lEquals;
    }
    else
    {
        return false;
    }
}

public int hashCode(){
    int result = 17;
    result = 37*result +
        ((this.id != null) ? this.id.hashCode() : 0);

    result = 37*result +
        ((this.type != null) ? this.type.hashCode() : 0);
    return result;
}
}
```

The combination of the XDoclet tags and the `<ejbdoclet/>` subtask will cause XDoclet to generate the value object for the entity bean. The generated class will contain getters for all the data fields of the bean, as well as the primary key. When used by a bean, the generated value object is a read-only snapshot of the entity bean. It can therefore be passed to clients as a lightweight representation of the bean. Value objects can also be used to separate a client's view to the data persistence model being used.

◆ **See also**

- 2.1—Generating home, remote, local, and local home interfaces
- 2.2—Adding and customizing the JNDI name for the home interface
- 2.5—Generating a primary key class

2.5 Generating a primary key class

◆ **Problem**

You want to generate a primary key class for your entity beans during development.

◆ **Background**

As you develop more and more entity beans, you find yourself also having to create the primary key class. As we emphasized in this chapter, having to code one more class just adds to the time it takes to develop a bean and increases your chances for having source files out of synch.

◆ **Recipe**

To have XDoclet generate a primary key class, use the `@ejb.pk` tag in your bean source file, use the `@ejb.pk-field` tag to denote an accessor for the primary key, and modify your `<ejbdoclet/>` task to include the `<entitypk/>` subtask. For instance, examine the `ItemBean` class shown in listing 2.10. The XDoclet tags applicable to this recipe are shown in bold; the others can be found in earlier recipes.

Listing 2.10 `ItemBean.java`

```
package ch2;

import javax.ejb.*;
/**
 * @ejb.bean type="CMP"
 *         name="ItemBean"
 *         jndi-name="ejb/ItemBean"
 *         view-type="both"
 *         primkey-field="ID"; ← Identifies the
 *                               primary key field
 * @ejb.pk ← Adds the primary key tag
 */
public abstract class ItemBean implements EntityBean
{
    public void setEntityContext( EntityContext context) {}
    public void unsetEntityContext( ) {}

    public void ejbRemove() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}

    /**
     * @ejb.create-method
     */
}
```

```

    public void ejbCreate( String id )
    {
        setID( id );
    }

    /**
     * @ejb.interface-method
     * @ejb.persistence
     * @ejb.pk-field
     */
    public abstract String getID();
}

```

Identifies the primary
key getter method

Notice in addition to the placement of the specified tags that we included the `primkey-field` attribute in the `@ejb.bean` tag at the class declaration level. Note that you must also use the `@ejb.persistence` tag in combination with the `@ejb.pk-field` tag.

◆ Discussion

The result of using these tags is a source file named `ItemBeanPK.java` (containing the `ItemBeanPK` class). Listing 2.11 shows the generated code for this class.

Listing 2.11 `ItemBeanPK.java`

```

/*
 * Generated by XDoclet - Do not edit!
 */
package ch2;

/**
 * Primary key for ItemBean.
 */
public class ItemBeanPK extends java.lang.Object
    implements java.io.Serializable
{
    private int _hashCode = Integer.MIN_VALUE;
    private StringBuffer _toStringValue = null;

    public java.lang.String iD;

    public ItemBeanPK()
    {
    }

    public ItemBeanPK( java.lang.String id )
    {
        this.iD = id;
    }
}

```

```
public java.lang.String getID()
{
    return iD;
}

public void setID(java.lang.String iD)
{
    this.iD = iD;
    _hashCode = Integer.MIN_VALUE;
}

public int hashCode()
{
    if( _hashCode == Integer.MIN_VALUE )
    {
        if (this.iD != null) _hashCode += this.iD.hashCode();
    }

    return _hashCode;
}

public boolean equals(Object obj)
{
    if( !(obj instanceof ch2.ItemBeanPK) )
        return false;

    ch2.ItemBeanPK pk = (ch2.ItemBeanPK)obj;
    boolean eq = true;

    if( obj == null )
    {
        eq = false;
    }
    else
    {
        if( this.iD == null &&
            ((ch2.ItemBeanPK)obj).getID() == null )
        {
            eq = true;
        }
        else
        {
            if( this.iD == null ||
                ((ch2.ItemBeanPK)obj).getID() == null )
            {
                eq = false;
            }
            else
            {
                eq = eq && this.iD.equals( pk.iD );
            }
        }
    }
}
```

```
        return eq;
    }

    /** @return String representation of
     *   this pk in the form of [.field1.field2.field3]. */
    public String toString()
    {
        if( _toStringValue == null )
        {
            _toStringValue = new StringBuffer("[" + ".");
            _toStringValue.append(this.id) .append(' ');
            _toStringValue.append(']');
        }

        return _toStringValue.toString();
    }
}
```

The generated primary key class contains a default constructor, an initialization constructor that accepts a `String` ID parameter, a getter method, a setter method, `hashCode()` and `equals()` methods, and a `toString()` method.

If you use the `@ejb.pk` tag without using the `@ejb.pk-field` tag, you generate a primary key file without the getter, setter, and initialization constructor.

◆ **See also**

2.1—Generating home, remote, local, and local home interfaces

2.6 Avoiding hardcoded XDoclet tag values

◆ **Problem**

You would like to centralize values in one place and not have to modify source files in order to update the values.

◆ **Background**

XDoclet is a great tool for generating necessary EJB files. In addition, it lets you specify values for the XML deployment descriptor and JNDI names for your beans. Using XDoclet with your development lets you automate and generate almost everything you need. However, as you add more XDoclet JavaDoc tags to your source files, you are specifying more values in code for things like JNDI names and bean names. Now you have many values spread out across many bean source files.

◆ **Recipe**

Use Ant properties in your XDoclet tags. Examine listing 2.12, which contains a subsection from a build.xml file. This subsection defines a property and the <ejbdoclet/> task.

Listing 2.12 Sample Build.xml

```

<property name="user.bean.jndi"
          value="ejb/session/UserBean"/>

```

Creates an Ant property

```

<target name="ejbdoclet" depends="init">
  <taskdef name="ejbdoclet"
          classname="xdoclet.modules.ejb.EjbDocletTask">
    <classpath>
      <fileset dir="${xdoclet.lib.dir}" includes="*.jar"/>
    </classpath>
  </taskdef>

  <ejbdoclet destdir="${src}" ejbspec="2.0" >
    <fileset dir="${src}">
      <include name="**/*Bean.java" />
    </fileset>

    <remoteinterface pattern="{0}Remote"/>
    <homeinterface/>
    <localhomeinterface/>
    <homeinterface/>

    <entitypk/>

    <deploymentdescriptor destdir="${build}/ejb/META-INF" />
  </ejbdoclet>
</target>

```

Completes the remaining task

Notice the property `user.bean.jndi` at the top of the file. Now examine the class declaration for the `UserBean`; it uses the Ant property in the JNDI attribute of the `@ejb.bean` tag:

```

/**
 * @ejb.bean type="Stateful"
 *          view-type="both"
 *          jndi-name="${user.bean.jndi}"
 *
 */
public class UserBean implements SessionBean{

```

◆ Discussion

When XDoclet attempts to generate the home interface for this bean, it will see that for the JNDI name it should use the value specified in the Ant property `user.bean.jndi`. Ant replaces the named property in the source file with the value contained in the `build.xml` file. Using this system, you can replace every hard-coded value in your source XDoclet JavaDoc tags with Ant property names. The advantage of this system is that it centralizes all of your property values into your `build.xml` file, and you no longer have to alter source code to change a value.

XDoclet allows you to specify everything about a bean in its source file. Not everything is included in this chapter, but the list includes security roles, EJB relationships, method permission, transactions, and more. By moving all the values of these various elements into Ant properties in the `build.xml` file, you create a centralized control of the various values that can be changed at build time in a single file.

◆ See also

- 2.1—Generating home, remote, local, and local home interfaces
- 2.2—Adding and customizing the JNDI name for the home interface
- 2.3—Keeping your EJB deployment descriptor current

2.7 Facilitating bean lookup with a utility object

◆ Problem

You want to generate a utility object to help with looking up the home interface of an EJB.

◆ Background

Two often-repeated tasks in an EJB application are the lookup of a bean's home interface and the subsequent creation of the bean. Developers sometimes handle these tasks by creating a static method that contains the lookup code for a particular bean. However, it is possible that this code also must change as a bean changes. The generated class will encapsulate all the code necessary for looking up the home interface of its parent EJB.

◆ Recipe

To generate a utility object, use the `@ejb.util` tag in the class-level JavaDoc of your bean and modify your `<ejbdoclet/>` task to include the `<utilityobject/>` subtask. This works for both entity and session beans. For example, examine the class declaration of the `UserBean`:

```
package ch2;

import javax.ejb.*;

/**
 * @ejb.bean type="Stateful"
 *         view-type="both"
 *
 * @ejb.util
 *
 */
public class UserBean implements SessionBean{
```

Listing 2.13 contains the `build.xml` used to generate the utility object.

Listing 2.13 Sample Build.xml

```
<target name="ejbdoclet" depends="init">
  <taskdef name="ejbdoclet" classname="xdoclet.modules.ejb.EjbDocletTask" >
    <classpath>
      <fileset dir="${xdoclet.lib.dir}" includes="*.jar"/>
    </classpath>
  </taskdef>

  <ejbdoclet destdir="${src}" ejbspec="2.0" >
    <fileset dir="${src}">
      <include name="**/*Bean.java" />
    </fileset>

    <utilobject cacheHomes="true" /> ① Adds the utility
                                     object subtask
  </ejbdoclet>
</target>
```

- ① The `<utilobject/>` subtask tells XDoclet to search for source files containing the `@ejb.util` class-level JavaDoc tag and generate a utility object. Notice the subtask specifies an attribute `cacheHomes` equal to `true`. This attribute tells XDoclet to generate a utility object that caches the home object after the first lookup in order to improve performance. Listing 2.14 shows the generated utility class for this example (reformatted for this chapter).

Listing 2.14 UserUtil.java, generated by XDoclet

```
/*
 * Generated by XDoclet - Do not edit!
 */
package ch2;

import javax.rmi.PortableRemoteObject;
import javax.naming.NamingException;
import javax.naming.InitialContext;

import java.util.Hashtable;

/**
 * Utility class for ch2.User.
 */
public class UserUtil
{
    /** Cached remote home (EJBHome). Uses lazy loading to obtain
     * its value (loaded by getHome() methods). */
    private static ch2.UserHome cachedRemoteHome = null;

    /** Cached local home (EJBLocalHome). Uses lazy loading to obtain
     * its value (loaded by getLocalHome() methods). */
    private static ch2.UserLocalHome cachedLocalHome = null;

    // Home interface lookup methods

    /**
     * Obtain remote home interface from default initial context
     * @return Home interface for ch2.User. Lookup using COMP_NAME
     */
    public static ch2.UserHome getHome() throws NamingException
    {
        if (cachedRemoteHome == null) {
            // Obtain initial context
            InitialContext initialContext = new InitialContext();
            try {
                java.lang.Object objRef =
                    initialContext.lookup(ch2.UserHome.COMP_NAME);
                cachedRemoteHome = (ch2.UserHome)
                    PortableRemoteObject.narrow(objRef,
                        ch2.UserHome.class);
            } finally {
                initialContext.close();
            }
        }
        return cachedRemoteHome;
    }

    /**
     * Obtain remote home interface from parameterised initial context
     * @param environment Parameters to use for creating initial context
     * @return Home interface for ch2.User. Lookup using COMP_NAME
     */
}
```

```

*/
public static ch2.UserHome getHome( Hashtable environment )
                                throws NamingException
{
    // Obtain initial context
    InitialContext initialContext =
        new InitialContext( environment );
    try {
        java.lang.Object objRef =
            initialContext.lookup( ch2.UserHome.COMP_NAME );
        return (ch2.UserHome)
            PortableRemoteObject.narrow( objRef, ch2.UserHome.class );
    } finally {
        initialContext.close();
    }
}

/**
 * Obtain local home interface from default initial context
 * @return Local home interface for ch2.User. Lookup using COMP_NAME
 */
public static ch2.UserLocalHome getLocalHome()
                                throws NamingException
{
    // Local homes shouldn't be narrowed,
    // as there is no RMI involved.
    if (cachedLocalHome == null) {
        // Obtain initial context
        InitialContext initialContext = new InitialContext();
        try {
            cachedLocalHome = (ch2.UserLocalHome)
                initialContext.lookup( ch2.UserLocalHome.COMP_NAME );
        } finally {
            initialContext.close();
        }
    }
    return cachedLocalHome;
}
}

```

Looks up and stores the home interface

Looks up and stores the local home interface

◆ Discussion

In addition to the `cacheHomes` attribute, you could add the `generate` attribute to the `@ejb.util` tag to specify whether the generated utility class should use the JNDI name or the component name from the home interface to perform a lookup (see recipe 2.2). The default behavior for the utility object is to use the JNDI name, but the possible values are `false`, `logical`, or `physical`. Keep in mind that

XDoclet uses every piece of information it has on a bean to generate applicable files. The generated utility object uses the names declared in the `public static final` member variables of the home and local home interfaces to perform look-ups, making your code more stable.

◆ **See also**

- 2.1—Generating home, remote, local, and local home interfaces
- 2.2—Adding and customizing the JNDI name for the home interface

2.8 Generating vendor-specific deployment descriptors

◆ **Problem**

You would like to generate a vendor-specific XML file along with the standard XML descriptor.

◆ **Background**

One of the great reasons to use J2EE is that its API is a published standard. This means that EJB applications should be portable across different vendors' application servers. Vendors maintain the specified functionality from the J2EE specification, but usually ask that developers deploy EJBs with an additional deployment XML file that is specific to the application server. This vendor-specific XML file allows the application server to correctly map EJB functionality to its EJB container.

◆ **Recipe**

Use the appropriate subtask in the `<ejbdoclet/>` task of your `build.xml` file. Table 2.3 lists the subtasks that XDoclet uses to generate the vendor-specific XML descriptors.

Table 2.3 These subtasks can be added to your `<ejbdoclet/>` task to generate the vendor-specific deployment XML for your EJBs. Along with each subtask are associated JavaDoc comments in order to help XDoclet completely generate the XML. Refer to the XDoclet documentation for more information about each of these tasks.

Application Server	Subtask	Comments
Weblogic	<code><weblogic/></code>	Generates descriptors for versions 6.0 and 6.1
JBoss	<code><jboss/></code>	Generates the <code>jboss.xml</code> and <code>jaws.xml</code> files

(continued on next page)

Table 2.3 These subtasks can be added to your `<ejbdoclet/>` task to generate the vendor-specific deployment XML for your EJBs. Along with each subtask are associated JavaDoc comments in order to help XDoclet completely generate the XML. Refer to the XDoclet documentation for more information about each of these tasks. (continued)

Application Server	Subtask	Comments
JonAS	<code><jonas/></code>	
JRun	<code><jrun/></code>	
Orion	<code><orion/></code>	Generates the orion-ejb-jar.xml
Websphere	<code><websphere/></code>	
Pramati	<code><pramati/></code>	
Resin	<code><resin-ejb-xml/></code>	Generates the resin-ejb xml
HPAS	<code><hpas/></code>	
EAServer	<code><easerver/></code>	Generates XML for EAServer 4.1

◆ **Discussion**

These subtasks have many common attributes, but also contain a set of subtask-specific attributes. Consult the XDoclet documentation for the details specific to your application server. In addition, many of the subtasks have XDoclet tags that you can include in your bean source file to make the XML generation more complete.

◆ **See also**

2.3—Keeping your EJB deployment descriptor current

2.9 Specifying security roles in the bean source

◆ **Problem**

You want to generate security roles directly into the EJB deployment descriptor. You do not want to edit the XML file manually.

◆ **Background**

Rather than updating the XML deployment descriptor for a bean with security information after development, you would like it generated along with the other XML parts of the descriptor. Creating security roles in the XML can be tedious and error prone when you edit by hand.

◆ Recipe

Listing 2.15 contains the `UserBean` from recipe 2.6 with additional JavaDoc comments (shown in bold) to create security constraints in the generated XML deployment descriptor.

Listing 2.15 Declaring security roles in the source code

```
/**
 * @ejb.bean type="Stateful"
 *          view-type="both"
 *          jndi-name="{user.bean.jndi}"
 *
 * @ejb.security-role-ref
 *          role-name="ADMIN"
 *          role-link="administrator"
 *
 */
public class UserBean implements SessionBean{
```

You must also be sure that your `<ejbdoclet/>` task in the `build.xml` file includes the correct subtask to generate the deployment XML file (see recipe 2.3).

◆ Discussion

As you can see, there is nothing too complicated about specifying security roles. In addition, you can use the `@ejb.security-identity` tag to declare the bean to assume a role when it acts as a client to another bean. This tag has the attributes `user-caller-identity` and `run-as`, which correspond to the XML elements you should recognize.

◆ See also

- 2.3—Keeping your EJB deployment descriptor current
- 2.8—Generating vendor-specific deployment descriptors
- 2.10—Generating and maintaining method permissions

2.10 Generating and maintaining method permissions

◆ Problem

You would like to automate the permission-creation method in the deployment XML. You also want the XML to change as your EJB methods and security roles change.

◆ **Background**

In addition to needing security roles in the EJB deployment XML, EJB applications usually need method permissions based on those roles in order to provide access control to various EJB methods. As EJBs change, and as new EJBs are created, the method permissions created in the deployment descriptor must also change. In addition, as you create new methods (or new security roles), you will have to add method permissions in the XML.

◆ **Recipe**

Use the `@ejb.permission` tag in the method-level JavaDoc comments to specify method permissions for specific methods. This tag must be used in combination with `@ejb.create-method` or `@ejb.interface-method`. Refer to recipe 2.1 for more information on those tags. The `UserBean` source subsection in listing 2.16 shows a single method declaring method permissions (highlighted in bold).

Listing 2.16 `UserBean.java`

```
package ch2;

import javax.ejb.*;

/**
 * @ejb.bean type="Stateful"
 *         view-type="both"
 *         jndi-name="{user.bean.jndi}"
 *
 */
public class UserBean implements SessionBean{

    private String name = null;

    /**
     * @ejb.interface-method
     * @ejb.permission
     * unchecked="true";
     */
    public void setName( String name )
    {
        this.name = name;
    }
}
```

◆ **Discussion**

When using the `@ejb.permission` tag, you can use the `role-name` attribute to specify a specific role for the method permission or the `unchecked` attribute to indicate

universal access. The `role-name` attribute can have a single role name value, or it can be a comma-separated list of role names that can access the method. The use of the `@ejb.permission` tag, along with others in this chapter, helps you to more completely generate your `ejb-jar.xml` for deploying your EJBs. This tag must be used with `@ejb.create-method` or `@ejb.interface-method` so that XDoclet knows with which method the permission is associated. To that end, you must include the subtask `<deploymentdescriptor/>` in your `build.xml` file in order to generate any new XML.

The generated XML will differ depending on which EJB interfaces you are generating. If you generate both, you should see XML for the method permission generated for both view types.

◆ **See also**

- 2.1—Generating home, remote, local, and local home interfaces
- 2.3—Keeping your EJB deployment descriptor current
- 2.8—Generating vendor-specific deployment descriptors
- 2.9—Specifying security roles in the bean source

2.11 **Generating finder methods for entity home interfaces**

◆ **Problem**

You want to generate the finder method declaration as part of the home interface generation process.

◆ **Background**

Recipe 2.1 shows how to generate home (and other) interfaces for session and entity beans. In that recipe, we add creation methods to the home interface. In the case of entity beans, home interfaces often need to include finder methods. Adding these finder methods requires time-consuming changes to the interface and may cause file synchronization problems, as described in recipe 2.1.

◆ **Recipe**

To generate the finder method declaration, use the `@ejb.finder` tag in the class-level JavaDoc of your bean source. For example, the following class section of

code from the `ItemBean` generates a finder method for the bean's home and local home interface:

```
package ch2;

import javax.ejb.*;
/**
 * @ejb.bean type="CMP"
 *           name="ItemBean"
 *           jndi-name="ejb/ItemBean"
 *           view-type="both"
 *
 * @ejb.finder signature="java.util.Collection findAll()"
 */
public abstract class ItemBean implements EntityBean
{
```

◆ **Discussion**

The result of this tag is the declaration of the finder method into the home and local home interface of the EJB. As long as you are generating the home or local home interface, you don't need to make any changes to the `build.xml` file.

◆ **See also**

2.1—Generating home, remote, local, and local home interfaces

2.12 Generating the `ejbSelect` method XML

◆ **Problem**

You want to use XDoclet to generate the XML for the select methods of a bean.

◆ **Background**

Entity beans often must declare specific select methods allowing you to select collections or specific entities from the persistent store. Select methods must be described in the deployment XML for a bean. As with all manual tasks, editing the XML descriptor is error prone and tedious.

◆ **Recipe**

To generate the XML for select methods, declare the abstract select methods in your bean class and identify them with the `@ejb.select` tag in their JavaDoc comments. Use the tag attribute `query` to specify the EJB-QL statement for the method. For instance, examine the `ItemBean` in listing 2.17.

Listing 2.17 ItemBean.java

```
package ch2;

import javax.ejb.*;
/**
 * @ejb.bean type="CMP"
 *          name="ItemBean"
 *          jndi-name="ejb/ItemBean"
 *          view-type="both"
 */
public abstract class ItemBean implements EntityBean
{
    //various bean methods...

    //ejbSelect methods

    /**
     * @ejb.select query="SELECT OBJECT( i ) FROM Item AS i"
     */
    public abstract java.util.Collection ejbSelectAll();
}
```

Also, you must specify the `<deploymentdescriptor/>` subtask in your build.xml file.

◆ Discussion

Select methods are not generated into a particular interface—the only result you should see is in the XML deployment descriptor. The descriptor will contain the EJB-QL and proper declarations for the method. Keep in mind that `ejbSelect` methods run under the transaction context of the invoker.

◆ See also

2.3—Keeping your EJB deployment descriptor current

2.13 Adding a home method to generated home interfaces

◆ Problem

You want to add home methods to your generated home or local home interface.

◆ Background

Occasionally you need to compute a value that encompasses all bean instances, such as the sum of all account balances over all `Account` entity beans. Since these methods are independent of any particular bean instance, they need to be defined on the home interface. As long as XDoclet is generating your home interface (see recipe 2.1), you should add any home methods to that generation. Please read recipe 2.1 before following this recipe.

◆ Recipe

To add home methods to either/both of your home and local home interfaces, you simply need to add a method-level JavaDoc tag to the method in the bean source. For example, the following method from an entity bean illustrates the necessary JavaDoc:

```
/**
 * @ejb.home-method
 *     view-type="both"
 */
public void addDataToAll()
{
    //method implementation here
}
```

◆ Discussion

Adding a home method to your home interfaces (home and local home) is no different than adding a regular business interface method—except that the JavaDoc tag routes the method to the home interface. The `@ejb.home-method` JavaDoc tag has an optional attribute, `view-type`, which you can use to specify the home interfaces you want to add this method. The possible values are `remote`, `local`, and `both`. This recipe once again illustrates how XDoclet provides the easiest way to keep your interface synchronized with your EJB source. If you later add methods, such as a home method, to your bean source, another trip through the Ant build process will entirely regenerate your interfaces and keep them up to date.

◆ See also

2.1—Generating home, remote, local, and local home interfaces

2.14 Adding entity relation XML to the deployment descriptor

◆ **Problem**

You want to generate the deployment XML for an entity bean relationship.

◆ **Background**

A new feature for EJB 2.0 applications is the ability to relate entity beans using *relationships*. This is similar to what you would find in any relational database. With EJB 2.0, you can create one-to-one, one-to-many, and many-to-many data relationships. The only drawback is that creating relationships requires large additions to the `ejb-jar.xml` file. Please read recipes 2.1 and 2.3 before using this recipe.

◆ **Recipe**

The following source shows a method that indicates a relationship between two entity beans. This method comes from the `OwnerBean` entity bean. Each `OwnerBean` entity bean is related unidirectly to a `DataBean` entity bean.

```
/**
 * @ejb.interface-method
 * @ejb.relation
 *     name="OwnerToData"
 *     relation-role="Owner"
 *     target-ejb="ch2.DataBean"
 */
public abstract Data getData();
```

◆ **Discussion**

Using the method-level `@ejb-relation` tag shown in the recipe generates the following XML in the assembly descriptor section of the `ejb-jar.xml` file:

```
<relationships >
  <ejb-relation >
    <ejb-relation-name>OwnerToData</ejb-relation-name>
    <ejb-relationship-role >
      <multiplicity>One</multiplicity>
      <relationship-role-source >
        <ejb-name>ch2.Owner</ejb-name>
      </relationship-role-source>
      <cmr-field >
        <cmr-field-name>data</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role >
  </ejb-relation >
</relationships >
```

```
</ejb-relationship-role>
<ejb-relationship-role >
  <multiplicity>One</multiplicity>
  <relationship-role-source >
    <ejb-name>ch2.DataBean</ejb-name>
  </relationship-role-source>
</ejb-relationship-role>
</ejb-relation>
</relationships>
```

The `JavaDoc` tag is used to specify a data accessor that indicates the entity data relationship. In this case, the `OwnerBean` entity data is related to the `DataBean` entity bean. The three attributes shown with the tag are the mandatory properties that must be set when using this tag.

◆ **See also**

- 2.3—Keeping your EJB deployment descriptor current
- 3.7— Modeling one-to-one entity data relationships

2.15 Adding the destination type to a message-driven bean deployment descriptor

◆ **Problem**

You want to generate the XML for the JMS message destination type while generating the deployment descriptor for a message-driven bean.

◆ **Background**

Message-driven beans must declare their destination type in their deployment descriptor from which they will be receiving JMS messages. Recipe 2.3 showed how to use `XDoclet` to generate the deployment descriptor for EJBs. Additionally, you can specify the destination type for a message-driven bean in its class source and add it to the generated XML. Please read recipe 2.3 before using this one.

◆ **Recipe**

To generate the XML for the message destination type, add the `destination-type` attribute to the class-level `@ejb.bean` `XDoclet` tag for your message-driven bean class. The following code does this for the `MessageBean` class:

```

/**
 * @ejb.bean
 *     name="MessageBean"
 *     type="MDB"
 *     destination-type="javax.jms.Queue"
 */
public class MessageBean
    implements MessageDrivenBean, MessageListener {

```

Notice also the change in the type attribute for this example. Instead of `session` or `entity`, its value is `MDB`, indicating that this class is a message-driven EJB.

◆ Discussion

Using the `destination-type` attribute with the `@ejb.bean` tag generates the additional XML (shown in bold):

```

<ejb-jar>
<enterprise-beans>

  <message-driven>
    <ejb-name>MDB</ejb-name>
    <ejb-class>MessageBean</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-driven-destination>
      <destination-type>javax.jms.Topic</destination-type>
    </message-driven-destination>
  </message-driven>

</enterprise-beans>
</ejb-jar>

```

The other possible value would be `javax.jms.Topic`, which would add a `Topic` destination instead of a `Queue`. If you are using a `Topic`, then you can optionally specify whether the topic should be `Durable` or `NonDurable` by using an additional attribute, `subscription-durability`.

◆ See also

- 2.3—Keeping your EJB deployment descriptor current
- 2.16—Adding message selectors to a message-driven bean deployment descriptor
- Chapter 6, “Messaging”

2.16 Adding message selectors to a message-driven bean deployment descriptor

◆ **Problem**

You want to generate the XML for a message selector while generating the deployment descriptor for a message-driven bean.

◆ **Background**

Message-driven beans have the ability to filter incoming messages by using message selectors. Each message selector for a message-driven bean must be specified in its deployment XML. Recipe 2.3 showed how to use XDoclet to generate the deployment descriptor for EJBs. You can also use XDoclet to add a message selector to generated deployment XML for a message-driven bean. Please read recipe 2.3 before using this one.

◆ **Recipe**

To generate the XML for a message selector, add the `message-selector` attribute to the class-level `@ejb.bean` XDoclet tag for your message-driven bean class. The following code does this for the `MessageBean` class:

```
/**
 * @ejb.bean
 *   name="MessageBean"
 *   type="MDB"
 *   message-selector="<![CDATA messageType = 'buyerRequest']]"
 */
public class MessageBean
    implements MessageDrivenBean, MessageListener {
```

Notice also the change in the `type` attribute for this example. Instead of `session` or `entity`, its value is `MDB`, indicating that this class is a message-driven EJB.

◆ **Discussion**

Using the `message-selector` attribute with the `@ejb.bean` tag generates the following XML:

```
<ejb-jar>
  <enterprise-beans>
```

```
<message-driven>
  <ejb-name>MDB</ejb-name>
  <ejb-class>MessageBean</ejb-class>
  <transaction-type>Container</transaction-type>
  <message-selector>
    <![CDATA[ messageType = 'buyerRequest' ]]>
  </message-selector>
</message-driven>

</enterprise-beans>
<ejb-jar>
```

Notice the use of the CDATA brackets when specifying the message selector value. Because message selectors can use special characters like > and <, you must use the CDATA brackets so that the XML file can be correctly parsed.

◆ **See also**

2.3—Keeping your EJB deployment descriptor current

2.15—Adding the destination type to a message-driven bean deployment descriptor

Chapter 6, “Messaging”

Messaging

“An army marches on its stomach.”

—*Napoleon Bonaparte*

With the introduction of the message-driven bean in the EJB 2.0 specification, Enterprise JavaBean applications can now easily be integrated with messaging systems. Java 2 Platform Enterprise Edition (J2EE)-compliant application servers are required to provide messaging capabilities. Before the message-driven bean, EJB applications could still send Java Message Service (JMS) messages and listen for those messages by including a JMS listener object; however, messages had to be processed in a synchronous manner. Message-driven beans are now the ideal way to expose business logic to messaging applications.

This chapter primarily focuses on problems associated with message-driven bean development. In this chapter, you will find recipes dealing with these topics:

- Sending JMS messages
- Creating a message-driven EJB
- Processing messages first in, first out
- Putting business logic in message-driven beans
- Streaming data with JMS
- Triggering multiple message-driven beans
- Speeding up message delivery
- Using message selectors
- Handling errors in a message-driven bean
- Sending email asynchronously
- Handling rollbacks in a message-driven bean

6.1 Sending a *publish/subscribe* JMS message

◆ **Problem**

You want to send a JMS message to a message topic (known as *publish/subscribe* messaging).

◆ **Background**

Enterprise applications can now use the JMS to communicate to outside applications or other application servers. EJBs can use JMS to decouple communication with these other systems in an asynchronous manner using a *publish/subscribe* pattern. JMS message topics create a one (the sender) to many (the receiver) relationship between messaging partners. In addition, *publish/subscribe* topics can

be used to store messages even when no entity is ready to retrieve them (referred to as *durable subscriptions*).

◆ **Recipe**

The code in listing 6.1 shows a private method, `publish()`, that can be used in any object that wishes to send a JMS message to a *publish/subscribe* topic.

Listing 6.1 The `publish()` method

```
private void publish(String subject, String content) {
    TopicConnection      topicConnection = null;
    TopicSession         topicSession = null;
    TopicPublisher       topicPublisher = null;
    Topic                topic = null;
    TopicConnectionFactory topicFactory = null;
    try{
        Properties props = new Properties();
        props.put (Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        props.put (Context.PROVIDER_URL, url);
        InitialContext context = new InitialContext( props );

        topicFactory =
            ( TopicConnectionFactory )
            context.lookup("TopicFactory");

        topicConnection =
            topicFactory.createTopicConnection();

        topicSession =
            topicConnection.createTopicSession( false,
                Session.AUTO_ACKNOWLEDGE );

        topic = ( Topic ) context.lookup( "ProcessorJMSTopic" );
        topicPublisher = topicSession.createPublisher(topic);

        MapMessage message = topicSession.createMapMessage();
        message.setString("Subject", subject );
        message.setString("Content", content);
        topicPublisher.publish(message);
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

Creates an InitialContext for the Weblogic application server

Looks up the topic factory

Creates a topic connection and session

Finds the topic and builds a publisher

Builds and sends the message

◆ **Discussion**

Publish/subscribe messaging allows you to send a single message to many message listeners. In fact, you can create message Topic destinations as durable, allowing message listeners to retrieve messages that were sent before the listener subscribed to the topic.

To send a message to a JMS topic, you first need to create a Java Naming and Directory Interface (JNDI) context and retrieve the JMS connection factory for topics in the JMS environment. Next, you must create a topic connection in order to establish a topic session. Once you have a session, you can find the actual topic to which you want to send a message, and build a publisher object for transmission of your message. Finally, simply construct your message and send it using the publisher. For more about the JMS API, visit <http://java.sun.com>

◆ **See also**

- 6.2—Sending a point-to-point JMS message
- 6.3—Creating a message-driven Enterprise JavaBean
- 7.8—Securing a message-driven bean

6.2 ***Sending a point-to-point JMS message***

◆ **Problem**

You want to send a point-to-point message.

◆ **Background**

Like the publish/subscribe messaging model shown in recipe 6.1, the point-to-point model allows applications to send messages asynchronously to remote message listeners. Point-to-point messaging differs in that it creates a one-to-one relationship between sender and receiver—that is, a single receiver consumes a single message. No message will be duplicated across multiple consumers.

◆ **Recipe**

The code in listing 6.2 shows a private method, `send()`, that can be used in any object that wishes to send a JMS point-to-point message.

Listing 6.2 The send() method

```

private void send(String subject, String content) {
    QueueConnection    queueConnection = null;
    QueueSession       queueSession=null;
    QueueSender        queueSender=null;
    Queue              queue=null;
    QueueConnectionFactory queueFactory = null;

    try{
        Properties props = new Properties();
        props.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        props.put(Context.PROVIDER_URL, url);
        InitialContext context = new InitialContext( props );

        queueFactory =
            (QueueConnectionFactory) context.lookup("QueueFactory");

        queueConnection = queueFactory.createQueueConnection();
        queueSession = queueConnection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);

        queue = (Queue)context.lookup("BookJMSQueue");
        queueSender = queueSession.createSender(queue);

        MapMessage message =
            queueSession.createMapMessage();
        message.setString("Symbol", symbol);
        message.setString("Description", description);
        queueSender.send(message);
    }
    catch(Exception e){
        log("Error Publishing Message");
        e.printStackTrace();
    }
}

```

Creates an InitialContext for the Weblogic application server

Looks up the topic factory

Creates a topic connection and session

Finds the topic and builds a sender

Builds and sends the message

◆ Discussion

To send a point-to-point message, you must send a message to a JMS message queue. To send the message, you first have to create a JNDI context and retrieve the JMS connection factory for a message queue in the JMS environment. Next, you must create a queue connection in order to establish a queue session. Once you have a session, you can find the actual queue to which you want to send a message, and build a sender object for transmission of your message. Finally, you simply construct your message and send it using the sender.

Message queues guarantee that messages are consumed by only one receiver and are never duplicated across multiple listeners (unlike a JMS topic). Message queues are ideal for messages that should be processed concurrently but only once. Many receivers can be pulling messages from a queue for processing at the same time, but each message will be sent to only one consumer.

◆ **See also**

- 6.1—Sending a publish/subscribe JMS message
- 6.3—Creating a message-driven Enterprise JavaBean
- 7.8—Securing a message-driven bean

6.3 **Creating a message-driven Enterprise JavaBean**

◆ **Problem**

You want to create a message-driven bean to contain business logic that will be triggered by a JMS message.

◆ **Background**

Message-driven beans (added to the EJB 2.0 specification) are assigned to receive messages from a particular JMS message destination. These EJBs are ideal for executing business logic asynchronously and for exposing EJB applications to enterprise messaging systems. Message-driven beans use the same transaction models (see chapter 5) and declarative security (see chapter 7) as do session and entity beans. Another advantage of message-driven beans is that they can be used to process messages concurrently. EJB containers can create a pool of identical message-driven beans that are able to process messages at the same time, generating a great deal of processing power.

◆ **Recipe**

This recipe illustrates how to build a simple message-driven bean and create its XML descriptor. The class in listing 6.3 defines a message-driven bean. It implements the required `MessageDrivenBean` interface and the necessary `MessageListener` interface that allows the bean to receive JMS messages.

Listing 6.3 SampleMDB.java

```

public class SampleMDB implements MessageDrivenBean, MessageListener
{
    private MessageDrivenContext ctx;

    public void ejbRemove() { }

    public void ejbPassivate() { }

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }

    public void ejbCreate () throws CreateException { }

    public void ejbActivate() { }

    public void onMessage( Message msg ) {
        MapMessage map = ( MapMessage ) msg;
        try {
            processMessage( map );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    private void processMessage( MapMessage map ) throws Exception
    {
        //implementation not shown
    }
}

```

← **Implements the MessageDrivenBean and MessageListener interfaces**

← **Handles incoming messages**

Listing 6.4 contains the partial deployment XML file for the bean; notice how it indicates the source type of messages for the bean (either point-to-point or publish/subscribe).

Listing 6.4 Deployment descriptor

```

<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>SampleMDB</ejb-name>
      <ejb-class>SampleMDB</ejb-class>
      <transaction-type>Container</transaction-type>
    
```

```

        <message-driven-destination>
            <destination-type>javax.jms.Topic</destination-type>
        </message-driven-destination>
    </message-driven>

</enterprise-beans>

<assembly-descriptor>
</assembly-descriptor>

</ejb-jar>

```

Describes the messaging type for this bean

Finally, you must perform the vendor-specific steps to assign the bean to an actual JMS message destination. The deployment XML describes only the type of messaging used by the message-driven bean, not the actual name of a topic or queue. Consult your application server documentation for more information. For example, the following XML could be used for the Weblogic application server:

```

<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <ejb-name>SampleMDB</ejb-name>
        <message-driven-descriptor>
            <destination-jndi-name>BookJMSTopic</destination-jndi-name>
        </message-driven-descriptor>
        <jndi-name>ejb/SampleMDB</jndi-name>
    </weblogic-enterprise-bean>
</weblogic-ejb-jar>

```

◆ Discussion

As with all other types of EJBs, security and transaction control is implemented in the usual way. In some cases, transactions and security do have special considerations that you must take into account when dealing with message-driven beans. For example, you need a good way to prevent unwanted clients from sending messages to your message-driven EJBs and triggering business logic, and you also need to know how to handle rollbacks in the `onMessage()` method. In addition, you should keep in mind that message-driven beans are stateless, and you should therefore not attempt to keep any state information stored at a class level in-between `onMessage()` invocations.

The `MessageDrivenBean` interface must be implemented in order to provide the bean with the appropriate bean methods, such as `ejbRemove()` and `ejbCreate()`. The `Context` object set in the bean is an instance of the `MessageDrivenContext`, which provides many of the methods found in the session and entity bean context classes. However, due to the nature of the message-driven bean, many of the

context methods will throw an exception if used. Since a message-driven bean has no real EJB client (only the container that delivers the message), the `getCallerPrincipal()` and `isCallerInRole()` methods throw a runtime exception. In addition, message-driven beans have no home interfaces (and therefore have no home objects), so `getEJBHome()` and `getEJBLocalHome()` also throw runtime exceptions if used. Finally, since no EJB clients exist for a message-driven bean, the transaction context for the start of the `onMessage()` method is started by the container in the case of container-managed transactions, or by the bean itself in the case of bean-managed transactions.

◆ **See also**

- 6.1—Sending a publish/subscribe JMS message
- 6.2—Sending a point-to-point JMS message
- 7.8—Securing a message-driven bean

6.4 Processing messages in a FIFO manner from a message queue

◆ **Problem**

You want to ensure that a message in a queue is processed only after any previous message has finished processing.

◆ **Background**

While some business logic operated by a message-driven bean can process messages in any order, other business functions might need messages supplied in a specific order. For instance, you might want to process incoming JMS messages according to the order in which they were received to preserve a specific data-driven workflow. Each message can be a step in a workflow, and the next step cannot begin without the previous one completing. Refer to recipe 6.2 for a discussion of using message queues.

◆ **Recipe**

The client shown in listing 6.5 publishes messages onto a message queue for a message-driven bean to pick up.

Listing 6.5 Client.java

```
public class Client
{
    private QueueConnection    queueConnection = null;
    private QueueSession      queueSession = null;
    private QueueSender        queueSender = null;
    private Queue              queue = null;
    private QueueConnectionFactory queueFactory = null;
    private String              url = getURL();

    public Client() throws JMSEException, NamingException {
        Context context = getInitialContext();

        queueFactory = (QueueConnectionFactory)
            context.lookup("BookJMSFactory");
        queueConnection = queueFactory.createQueueConnection();
        queueSession = queueConnection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);
        queue = (Queue) context.lookup("BookJMSQueue");
        queueSender = queueSession.createSender(queue);
    }

    public void send() throws JMSEException {
        MapMessage message = null;

        for(int i=0;i<10;i++){
            message = queueSession.createMapMessage();
            message.setInt("Index",i);
            queueSender.send(message);
        }
    }

    public void close() throws JMSEException {
        queueConnection.close();
    }

    public static void main(String[] args) {
        Client sender = null;

        try{
            sender = new Client();
            sender.send();
            sender.close();
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Notice that the client sends a counter value in the messages. The message-driven bean will use that value to show that the messages are received and processed one at a time. The message-driven bean shown in listing 6.6 picks up messages from the message queue and prints out the counter value that each message contains.

Listing 6.6 MessageBean.java

```
public class MessageBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;

    public void onMessage( Message msg ) {
        MapMessage map = (MapMessage) msg;

        try {
            int index = map.getInt("Index");
            System.out.println( "Processing message: " + index );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    //other bean methods not shown
}
```

Since we made use of a message queue, we are guaranteed that messages will be removed from the queue in the order in which they were placed. To ensure that one message is completely processed before the next message begins, you should deploy only a single message-driven bean to remove messages from the queue.

Listing 6.7 contains the deployment XML for the bean; notice how it indicates the source type of messages for the bean.

Listing 6.7 Deployment descriptor

```
<ejb-jar>
  <enterprise-beans>

    <message-driven>
      <ejb-name>fifoMDB</ejb-name>
      <ejb-class>fifo.MessageBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
        <destination-type>javax.jms.Queue</destination-type>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

```

</enterprise-beans>
<assembly-descriptor>
</assembly-descriptor>
</ejb-jar>

```

To ensure that the second message is not consumed before the first message processing has completed, you must have only one bean listening to the queue. This is set up in the vendor-specific deployment file. For example, you can use XML like that shown in listing 6.8 for the Weblogic application server.

Listing 6.8 Weblogic deployment descriptor

```

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>fifoMDB</ejb-name>
    <message-driven-descriptor>
      <pool>
        <max-beans-in-free-pool>1</max-beans-in-free-pool>
        <initial-beans-in-free-pool>1</initial-beans-in-free-pool>
      </pool>
      <destination-jndi-name>BookJMSQueue</destination-jndi-name>
    </message-driven-descriptor>
    <jndi-name>fifo.MBD</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>

```

**Creates the
message-
driven bean
pool of size 1**

◆ **Discussion**

Message queues guarantee that only one consumer will process each single message. By limiting the number of consumers assigned to a queue to a single message-driven bean, you ensure that all the messages will be processed in the order in which they were received. In addition, using one consumer guarantees that each message will completely process before the next one begins processing. Otherwise, you can create a pool of message-driven beans (by increasing the pool size in a vendor-specific manner) to pull messages faster from the queue. Messages will still only be delivered to a single message-driven bean instance, but using many message-driven bean instances with the same queue results in faster message processing.

◆ **See also**

6.2—Sending a point-to-point JMS message

6.5 Insulating message-driven beans from business logic changes

◆ **Problem**

You want to prevent changing your message-driven EJB classes when the business logic they invoke changes.

◆ **Background**

Message-driven EJBs are ideal for executing business logic via JMS messages. However, when developing an enterprise application in a changing environment (or a shorter development cycle), you might find that you spend too much time upgrading the business logic contained in your message-driven beans. It would be ideal to encapsulate your business logic and insulate your message-driven beans from unnecessary changes.

◆ **Recipe**

To shield your message-driven beans from business logic changes, encapsulate the business logic with an intermediary object. The message-driven EJB shown in listing 6.9 uses an instance of the `BusinessLogicBean` class.

Listing 6.9 `MessageBean.java`

```
public class MessageBean implements MessageDrivenBean, MessageListener
{
    private MessageDrivenContext ctx;

    public void onMessage(Message msg) {
        MapMessage map= (MapMessage)msg;

        try {
            String symbol = map.getString("Symbol");
            String description = map.getString("Description");

            BusinessLogicBean bean =
                new BusinessLogicBean( symbol, description );
            bean.executeBusinessLogic();
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Invokes the
encapsulated
business logic

◆ **Discussion**

The `BusinessLogicBean` class has a single purpose: to encapsulate business logic. This class is a simple object that allows the message-driven bean to execute business logic by passing in parameters and invoking a method. Using a class like this allows the EJB to shield itself from changes to the business logic. In addition, it is good practice to build business logic into reusable classes. An alternative to using a simple object is to invoke a session EJB that already encapsulates some business logic. By encapsulating all business logic with session beans, you ensure that the logic is available to both message-driven beans and any other EJB clients.

◆ **See also**

6.3—Creating a message-driven Enterprise JavaBean

6.6 Streaming data to a message-driven EJB

◆ **Problem**

You want to send a stream of data to a message-driven EJB.

◆ **Background**

Message-driven beans can receive all types of JMS messages. Because of that capability, you can use the most appropriate JMS message type for the data you want to send. For instance, when you want to send a large amount of binary data (like an image), you should stream the data to conserve bandwidth and memory. Refer to recipe 6.1 for more information on using message topics.

◆ **Recipe**

This solution demonstrates a client and a message-driven bean using streamed data. Listing 6.10 shows a client that streams a message containing data from a binary file to a message-driven EJB. It uses a message topic as a message destination.

Listing 6.10 Client.java

```
public class Client
{
    private TopicConnection    topicConnection = null;
    private TopicSession      topicSession = null;
    private TopicPublisher     topicPublisher = null;
    private Topic              topic = null;
    private TopicConnectionFactory topicFactory = null;
```



```

private String          url= "http://myjndihost";

public Client( String factoryJNDI, String topicJNDI )
    throws JMSEException, NamingException {

    // Get the initial context, implementation not shown
    Context context = getInitialContext();

    // Get the connection factory
    topicFactory = ( TopicConnectionFactory )
        context.lookup( factoryJNDI );

    // Create the connection
    topicConnection = topicFactory.createTopicConnection();

    // Create the session
    topicSession=topicConnection.createTopicSession( false,
        Session.AUTO_ACKNOWLEDGE );

    // Look up the destination
    topic = ( Topic ) context.lookup( topicJNDI );

    // Create a publisher
    topicPublisher = topicSession.createPublisher( topic );
}

public void sendToMDB( String filename ) throws JMSEException
{
    byte[]          bytes = new byte[1024];
    FileInputStream istream = null;
    int             bytesRead = 0;

    try{
        BytesMessage message = topicSession.createBytesMessage();
        Istream = new FileInputStream(filename);
        while( (bytesRead = istream.read( bytes,0,bytes.length ) ) > 0 )
        {
            message.writeBytes( bytes, 0, bytesRead);
        }
        istream.close();
        topicPublisher.publish(message);
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

public void close() throws JMSEException {
    topicSession.close();
    topicConnection.close();
}

public static void main(String[] args) {
    Client publisher = null;
    String filename = null;

    try{

```

**Creates the
BytesMessage
instance**

**Writes the data
to the message**

```

        publisher = new Client( "BookJMSFactory", "BookJMSTopic" );
        System.out.println("Publishing message:");

        if( args.length > 0){
            filename = args[0];
            publisher.sendToMDB(filename);
            publisher.close();
        }
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}

```

Listing 6.11 shows the sample message-driven bean that receives data from a streamed message. This bean simply prints out the data it receives.

Listing 6.11 MessageBean.java

```

public class MessageBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;
    public void ejbRemove() { }
    public void ejbPassivate() { }
    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }
    public void ejbCreate () throws CreateException { }
    public void ejbActivate() { }
    public void onMessage( Message msg )
    {
        BytesMessage message = ( BytesMessage ) msg;
        int bytesRead = 0;
        byte[] bytes = new byte[1024];
        try {
            while( (bytesRead = message.readBytes(bytes, 1024) ) > 0 ){
                System.out.println( new String( bytes, 0 , bytesRead ) );
            }
        }catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

**Reads the data
off the message**

◆ **Discussion**

Streaming large amounts of data helps you to avoid building a single large message. In addition, message streams are ideal for sending binary file data. By using message streams, you can more easily build messaging systems that can restart message transmission from the point of failure, rather than retransmit data. The client uses the `BytesMessage` message class. This message type is used specifically for sending large amounts of data to a message listener. The message-driven bean uses its `onMessage()` method to receive the message, as it would any other message type. The message-driven bean in this recipe only printed out the data it received from the streamed message, but it could instead store it in a database or create a new file containing the data.

◆ **See also**

- 6.1—Sending a publish/subscribe JMS message
- 6.3—Creating a message-driven Enterprise JavaBean

6.7 **Triggering two or more message-driven beans with a single JMS message**

◆ **Problem**

You want to start two or more business methods concurrently with a single JMS message.

◆ **Background**

Message-driven beans give other parts of an enterprise application the ability to execute business logic asynchronously. However, sending multiple JMS messages to execute multiple pieces of business logic can be time-consuming and redundant. To improve the efficiency of code, you should send a single message that triggers multiple message-driven beans.

◆ **Recipe**

To execute two pieces of business logic with a single message, you need only have two different message-driven beans listen for the same message. To do this, you must use a JMS message topic. Topics create a one-to-many relationship between sender and receiver(s). For this example, we will use two simple message-driven

beans (listings 6.12 and 6.13). The `onMessage()` method simply prints out a statement indicating it has received a message.

Listing 6.12 MessageBean.java

```
public class MessageBean implements MessageDrivenBean, MessageListener {
    public void onMessage(Message msg) {
        MapMessage map = ( MapMessage ) msg;
        try {
            String symbol = map.getString("Symbol");
            String description = map.getString("Description");
            System.out.println("MDB 1 received Symbol : " + symbol
                + " " + description );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
        //other bean methods not shown
    }
}
```

Listing 6.13 MessageBean2.java

```
public class MessageBean2 implements MessageDrivenBean, MessageListener {
    public void onMessage(Message msg) {
        MapMessage map=(MapMessage)msg;
        try {
            String symbol = map.getString("Symbol");
            String description = map.getString("Description");
            System.out.println("MDB 2 received Symbol : " + symbol
                + " " + description );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Listing 6.14 contains the XML descriptor for these beans. As you can see, the descriptor indicates the JMS destination type.

Listing 6.14 Deployment descriptor

```
<enterprise-beans>
  <message-driven>
    <ejb-name>MDB</ejb-name>
    <ejb-class>multiSubscriber.MessageBean</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-driven-destination>
      <destination-type>javax.jms.Topic</destination-type>
    </message-driven-destination>
  </message-driven>
  <message-driven>
    <ejb-name>MDB2</ejb-name>
    <ejb-class>multiSubscriber.MessageBean2</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-driven-destination>
      <destination-type>javax.jms.Topic</destination-type>
    </message-driven-destination>
  </message-driven>
</enterprise-beans>
```

Assigns the message-driven bean to a JMS topic

The actual topic used by the message-driven beans is specified in a vendor-specific manner. For example, listing 6.15 shows the XML used by the Weblogic application server to specify the JMS topic for each bean.

Listing 6.15 Weblogic deployment descriptor

```
<weblogic-ear>
  <weblogic-enterprise-bean>
    <ejb-name>MDB</ejb-name>
    <message-driven-descriptor>
      <destination-jndi-name>BookJMSTopic</destination-jndi-name>
    </message-driven-descriptor>
    <jndi-name>multiSubscriber.MDB</jndi-name>
  </weblogic-enterprise-bean>
  <weblogic-enterprise-bean>
    <ejb-name>MDB2</ejb-name>
    <message-driven-descriptor>
      <destination-jndi-name>BookJMSTopic</destination-jndi-name> #1
    </message-driven-descriptor>
    <jndi-name>multiSubscriber2.MDB</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ear>
```

Assigns the message-driven bean to the BookJMSTopic topic

◆ **Discussion**

Recipe 6.1 provides more information on JMS topics. Since they allow multiple message-driven beans (even message-driven beans of different Java classes) to receive the same incoming message, you can use them to create concurrent processing for sections of business logic. Sending a single message, you can trigger two completely unrelated business functions to start processing at the same time.

In this recipe, each message-driven bean simply prints out a statement indicating it has received a message. However, in a practical application the two message-driven beans should each contain an important business function. To ensure that both message-driven beans receive the same message, they both need to subscribe to a JMS topic. For both beans to be triggered by a single message, both EJBs need to use the same topic.

◆ **See also**

- 6.1—Sending a publish/subscribe JMS message
- 6.3—Creating a message-driven Enterprise JavaBean
- 6.9—Filtering messages for a message-driven EJB

6.8 **Speeding up message delivery to a message-driven bean**

◆ **Problem**

You want to reduce the time it takes for a message to start processing in a message-driven bean.

◆ **Background**

In most enterprise situations, you want your asynchronous business functions to complete as quickly as possible. Since a message-driven bean processes a single message at a time, the waiting time for a single message increases as the number of messages delivered before it increases. In other words, if a single message takes a long period of time to complete, other messages experience a delay before processing. In critical applications, these messages should be processed as quickly as possible.

◆ **Recipe**

To speed up the consumption of messages, use a pool of message-driven beans. Each EJB is an instance of the single EJB class. With a pool of message-driven beans, you can consume more messages in a shorter time. Listing 6.16 shows a simple message-driven bean used to consume messages.

Listing 6.16 MessageBean.java

```
public class MessageBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;

    public void ejbRemove() {
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }

    public void ejbCreate () throws CreateException {
    }

    public void onMessage(Message msg) {
        MapMessage map= (MapMessage) msg;

        try {
            String symbol = map.getString("Symbol");
            String description = map.getString("Description");

            System.out.println("MDB received Symbol : " + symbol
                + " " + description );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

To ensure a single message is not duplicated across instances in the message-driven bean pool, the message-driven bean instances should use a message queue as the destination type. Listing 6.17 contains the deployment descriptor for the bean.

Listing 6.17 Deployment descriptor

```

<enterprise-beans>
  <message-driven>
    <ejb-name>concurrentMDB</ejb-name>
    <ejb-class>concurrent.MessageBean</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-driven-destination>
      <destination-type>javax.jms.Queue</destination-type>
    </message-driven-destination>
  </message-driven>
</enterprise-beans>

```

Message-driven bean instance pools are created in a vendor-specific manner. Listing 6.18 shows how this is accomplished using the Weblogic application server. Notice the vendor XML creates a pool maximum size of five beans, with an initial size of two beans.

Listing 6.18 Weblogic deployment descriptor

```

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>concurrentMDB</ejb-name>
    <message-driven-descriptor>
      <pool>
        <max-beans-in-free-pool>5</max-beans-in-free-pool>
        <initial-beans-in-free-pool>2</initial-beans-in-free-pool>
      </pool>
      <destination-jndi-name>BookJMSQueue</destination-jndi-name>
    </message-driven-descriptor>
    <jndi-name>concurrent.MBD</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>

```

Sets up the message-driven bean pool

◆ Discussion

Using a bean pool is a quick and dirty way to achieve concurrent processing of messages. The pool, combined with a message queue, provides a way to process many messages at once without duplicating messages across instances. Creating an environment like this allows messages to start processing instead of waiting for previous messages to complete. You should use this type of processing only when

concurrent processing of messages will not cause problems in your business logic or invalid states in your data.

◆ **See also**

- 6.1—Sending a publish/subscribe JMS message
- 6.3—Creating a message-driven Enterprise JavaBean

6.9 Filtering messages for a message-driven EJB

◆ **Problem**

You want your message-driven beans to receive only the messages that they are intended to process.

◆ **Background**

Message-driven beans that subscribe to a topic or receive messages from a queue should be able to handle messages of the wrong type (which should not invoke the message-driven business logic). Beans should just politely discard these messages when they are encountered. This is especially true for message-driven beans that exist in an environment with many different beans that watch a single source for incoming messages. However, it would be more efficient to avoid the execution time used for discarding messages and instead avoid receiving unwanted messages.

◆ **Recipe**

To selectively deliver messages to a message-driven bean, the bean should be deployed with a *message selector*. The bean source needs no changes in order to use the message selector. Listing 6.19 shows a simple message-driven bean that only wants messages that contain an attribute `UserRole` set to `"BuyerRole"`. The bean prints out the role of the incoming message for verification.

Listing 6.19 MessageBean.java

```
public class MessageBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;

    public void onMessage(Message msg) {
        MapMessage map= (MapMessage)msg;

        try {
            String role = map.getString( "UserRole" );
```

```

        System.out.println("Received Message for Role: " + role);
        ProcessTheMessage( message );
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}
}
}

```

In the XML descriptor for the bean, you describe the message selector that filters undesired messages for the message-driven bean. Listing 6.20 shows the partial XML descriptor that describes the simple EJB and its message selector.

Listing 6.20 Deployment descriptor

```

<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>MDB</ejb-name>
      <ejb-class>messageSelector.MessageBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-selector>
        <![CDATA[ UserRole = 'BuyerRole' ]]>
      </message-selector>
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>
</ejb-jar>

```

**Specifies a
message
selector**

Here is a simple publish method that appropriately creates messages for the message-driven bean message selector:

```

public void publish(String role) throws JMSEException {
    MapMessage message = topicSession.createMapMessage();
    message.setString("UserRole", role);
    message.setStringProperty("UserRole", role);

    System.out.println( "Publishing message to Role:" + role );
    topicPublisher.publish(message);
}

```

◆ Discussion

When sending particular messages, we must assign a value to the property `User-Role`. The message selector will pick out the messages that meet its criteria and deliver them to the message-driven bean. Message selectors operate using the property values that are set in JMS messages. Any property that is set in the message can be examined by a message selector for filtering purposes.

Message selection strings can be any length and any combination of message property comparisons. The following is an example of a more complex message selector:

```
"DollarAmount < 100.00 OR (ShareCount < 100 AND ( CreditAmount  
- DollarAmount > 0 ) ) AND Role in ('Buyer', 'ADMIN' )"
```

You can make other familiar comparisons using the following operators as well: `=`, `BETWEEN`, and `LIKE` (using a `%` as a wildcard). As mentioned in the recipe, message selectors operate upon messages by examining the properties set in the message using its `setStringProperty()` method. If a property is not present in a message, the selector considers that a nonmatching message. To specify the message selector in the deployment XML, you must use the `CDATA` tag to avoid XML parsing errors due to the use of special characters like `<` or `>`.

◆ See also

6.3—Creating a message-driven Enterprise JavaBean

6.10 Encapsulating error-handling code in a message-driven EJB

◆ Problem

Rather than handle errors in a message-driven bean, you want your beans to off-load errors to an error-handling system.

◆ Background

Handling errors across all your message-driven beans should be consistent and exact. By keeping the error-handling code in your message-driven beans, you open your beans to tedious changes if your error-handling strategy changes. If you must change the error-handling code in one bean, you might have to change it in all your message-driven beans. Passing exceptions to an error-handling object or

session bean allows you to avoid rollbacks and gracefully handle errors in a consistent manner.

◆ **Recipe**

Instead of acting upon any errors, the message-driven bean catches any exceptions and forwards them on to an error-handling session bean. The message-driven bean should be implemented as usual; the only new addition is the error-handling mechanism (see listing 6.21).

Listing 6.21 MessageBean.java

```
public class MessageBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;

    public void onMessage(Message msg) {
        MapMessage map = (MapMessage) msg;
        String symbol = null;
        String description = null;
        ErrorHome errorHome = null;

        try {
            symbol = map.getString("Symbol");
            description = map.getString("Description");

            System.out.println("Received Symbol : " + symbol);
            System.out.println("Received Description : " + description);

            processEquityMessage( symbol, description );
        }
        catch(Exception e){
            e.printStackTrace();
            System.out.println("Error Creating Equity with Symbol:"+symbol);
            System.out.println("Consuming error and "
                + "passing on to error Handler");

            try{
                handleError( e, msg );
            }
            catch(Exception errorExc){}
        }
    }

    private void handleError( Exception e, Message msg )
    {
        ErrorHandler handler = lookupErrorEJB();
        handler.handleMessageDrivenError( e.getMessage(), msg );
    }
}
```

Looks up and
uses the error-
handling
session EJB

The `handleError()` method looks up a session EJB that handles specific errors. For example, the following remote interface could expose error-handling functionality to an entire EJB application:

```
public interface ErrorHandler extends javax.ejb.EJBObject
{
    public void handleMessageDrivenError( String message, Message msg );
    public void handleSessionError( Object errorMessage );
    public void handleEntityError( Object errorMessage );
}
```

◆ **Discussion**

The message-driven EJB shown in the recipe processes messages containing equity information. The actual message-processing logic is not shown, so instead let's examine the `handleError()` method invoked only when an exception occurs during message processing. The session EJB interface shown in the recipe declares methods for handling different types of errors. For example, the session bean has a specific way it can handle session bean errors, entity bean errors, and message-driven bean errors. Using an error-handling system like this does not have to take the place of a normal transactional system. Instead, it acts as a way to store information on errors occurring in your application—acting as a logger of errors, and possibly offloading them to a management system.

◆ **See also**

6.12—Handling rollbacks in a message-driven bean

6.11 *Sending an email message asynchronously*

◆ **Problem**

You want to provide your EJBs with the ability to send email in an asynchronous manner.

◆ **Background**

The ability to send email is an important part of many enterprise applications. Email can be used to send notifications, alerts, and general information, such as price quotes or contract information. When sending an email from an EJB, you should be able to continue processing without waiting for an email to be sent. Sending email using the Java mail package is a simple process.

◆ **Recipe**

Combining email-sending code with a message-driven bean provides the asynchronous behavior that is ideal for enterprise applications. Listing 6.22 contains a message-driven bean that sends email using property values passed to it via a JMS message.

Listing 6.22 EmailBean.java

```
import javax.jms.*;

public class EmailBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;

    public void onMessage( Message msg ) {
        MapMessage map = (MapMessage) msg;

        try {
            sendEmail( map.getProperty( "Recipient" ),
                      map.getProperty("message" ) );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    private void sendEmail(String recipient, String text)
        throws Exception {

        Session mailSession = null;
        javax.mail.Message msg = null;

        try{
            System.out.println( "Sending Email to: " + rcpt );

            mailSession = (Session) ctx.lookup("BookMailSession");

            msg = new MimeMessage(mailSession);
            msg.setFrom();
            msg.setRecipients(Message.RecipientType.TO,
                             InetAddress.parse( recipient , false));
            msg.setSubject("Important Message");
            msg.setText(text);
            Transport.send(msg);
            System.out.println("Sent Email to: "+rcpt);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Retrieves the email address and text from the JMS message

Sends the email message

◆ Discussion

When using a message-driven bean to send an email message, you need to be sure to send a JMS message with all the values that you need for the email. For instance, the solution in the recipe only retrieved the email address and text from the JMS message and populated the subject of the email with a hardcoded value.

Another improvement you can make to your message-driven email beans is to only send JMS messages that contain the email recipient address and the type of email to send. For instance, a message-driven bean can be initialized with standard email message texts to use for your various email needs in your enterprise application (purchase confirmation, error, contract status, etc.). This would include the subject and message. All your application needs to do is supply a valid email address and the type of email to send. This way, you won't have to transmit the body of an email message to the EJB. In addition, you could pass parameters to the EJB for formatting an already loaded email body.

◆ See also

4.5—Sending an email from an EJB

6.12—Handling rollbacks in a message-driven bean

6.12 Handling rollbacks in a message-driven bean

◆ Problem

When a transaction in a message-driven bean rolls back, the application server can be configured to resend the JMS message that started the transaction. If the error that caused the rollback keeps occurring, you could potentially cause an endless loop.

Background

Rollbacks in message-driven beans occur in the same way that they can happen in other beans—an error occurs in executing logic. However, in the case of a message-driven bean using a durable subscription, the application server will most likely attempt to redeliver the message that caused the rollback in the bean. If the error is not corrected, the rollback will continue on, consuming more processing time and memory. You need your message-driven beans to be insulated from rollback loops and able to handle error-causing messages without a rollback every time.

◆ **Recipe**

To handle rollbacks in a message-driven bean, keep track of the number of times a particular message has been delivered to the bean. Once a certain retry limit is reached, you want to discard the message from that point on. Listing 6.23 shows the `onMessage()` method from a message-driven bean that tracks and checks message redelivery attempts.

Listing 6.23 The `onMessage()` method

```
private HashMap previousMessages;
private int count = 0;

public void onMessage( Message incomingMsg )
{
    // get the unique message Id
    String msgId = incomingMsg.getJMSMessageID();
    if ( previousMessages.containsKey(msgId) ) ← Checks for previous
        count = ( (Integer) msgMap.get(msgId) ).intValue();      attempts
    else
        count = 0;

    // if msg has been retried couple of times, discard it.
    // and remove the stored id.
    if ( count < _MAX_REDLIVERY_CONST_ ) ← Checks the number of attempts
    {
        logMessage(incomingMsg);
        previousMessages.remove( msgId );
        return;
    }

    //perform business logic for message
    boolean error = doBusinessFunction();

    if ( error )
    {
        mdbContext.setRollBackOnly(); ← Checks for
        previousMessages.put( msgId, new Integer(++count) );      necessary
    }
    else
    {
        if( previousMessages.containsKey( msgId ) )
            previousMessages.remove( msgId );
    }
}
```


◆ **Discussion**

Some application servers and some JMS vendors allow you to specify the redelivery count of a rolled-back message delivery to a message-driven bean. However, to ensure your message-driven EJBs are the most secure and portable, you can implement a simple message tracker like the one shown in the recipe. In this code, the EJB maintains a Map of message IDs and the number of times they have been delivered. If the delivered count for a particular message reaches a predefined constant value, the bean simply logs the message and returns. By returning successfully, the EJB ensures that the EJB container commits the transaction and the message will not be delivered again.

If the message makes it past the count check, the bean will attempt to perform its business function. After attempting the business logic, the EJB will check to see if it is necessary to mark the current transaction for rollback. If so, the EJB uses its `MessageDrivenContext` instance to mark the transaction and returns. The container will roll back the transaction and will attempt to redeliver the message. The `previousMessages` Hashtable will store only those message IDs that caused errors. If the message succeeds, no ID is stored (and any previously stored ID is removed).

◆ **See also**

Chapter 5, “Transactions”

