EJB Control Developer's Guide

Table of contents

1. Overview

To access the capabilities of an Enterprise JavaBean (EJB) without an EJB control, several preparatory operations must be performed. You must look up the EJB in the JNDI registry, obtain the EJB's home interface, obtain an EJB instance, and then finally invoke methods on the EJB's remote interface to perform tasks.

The EJB control eliminates all of this preparatory work. Once you have created the EJB control, a web service or page flow can use the control to access the EJB's business methods directly. The EJB control manages communication with the EJB, including all JNDI lookup, interface discovery and EJB instance creation and management.

In short, EJB controls provide an alternative approach that makes it easy to use an existing, deployed EJB from within an application. EJB controls supports interaction with two of the three types of EJBs, that is, session beans and entity beans. The EJB control does not support direct communication with message-driven EJBs.

Note: Requests for messages can be sent indirectly to message-driven EJBs using the JMS control instead. However, unlike the EJB control, the JMS control is not used to locate and reference an existing message-driven EJB. For more information, see JMS Control.

To create an EJB control to represent an EJB, you must know the names of the home and business interfaces. The name for the home interface is typically of the form `com.mycompany.MyBeanNameHome` or `com.mycompany.MyBeanNameLocalHome`, and the business interface is typically of the form `com.mycompany.MyBeanName` or `com.mycompany.MyBeanNameLocal`. The EJB control uses either the EJB's local interfaces or the remote interfaces.

2. EJB Control Annotations

2.1. The EJBHome Annotation

EJBHome is a required class-level annotation used to specify the target EJB's home interface for the EJB control. Either the `jndiName` or `ejbLink` attribute must be specified.

Member Name	Value Type	Value Required	Description
jndiName	String	No	Specifies the JNDI name of the target EJB's home interface (e.g. EJBNameHome). This value may also be an URL using the

			"JNDI:" protocol (e.g. jndi://username:password@host:port/EJB
ejbLink	String	No	Specifies the name of the target EJB using the application relative path to the EJB JAR. This syntax causes the runtime to use an application scoped name when locating the referenced EJB. The naming syntax is BeanName#EJBJAR (e.g. CreditCard#CustomerData.jar).

2.2. The JNDIContextEnv Annotation

JNDIContextEnv specifies the environment properties for the JNDI context that will be used to lookup the target EJB. It is an optional class-level annotation for the EJB Control.

If using a URL with the "JNDI:" protocol or to use a JNDI context with the default envirnoment properties, this annotation is not necessary.

Member Name	Value Type	Value Required	Description
contextFactory	String	No	The fully qualified class name of a JNDI context factory. If not set the default InitialContext will be used and none of the other attribute values of this annotation will be used.
providerURL	String	No	The provider URL. Used only if `contextFactory` attribute has been set.
principal	String	No	Specifies the identity of the principal for authenticating the caller to the service. Used only if the `contextFactory`

			attribute has been set.
credentials	String	No	Specifies the credentials of the principal for authenticating the caller to the service. Used only if the `contextFactory` attribute has been set.

3. EJB Control Methods

The following methods are supported by the EJB Control:

Method	Description
getEJBHomeInstance()	Returns an instance of the home interface associated with the target bean component.
hasEJBHomeInstance()	Returns true if the EJB control currently has a target bean instance upon which bean business interface methods may be invoked. This will be true after a successful `create()` or single select finder method execution, or in cases where implicit creation or find has occurred by the control on the control users behalf. This provides a simple way to procedurally check the status of explicit or implicit bean instance creation or find operations.
getEJBBeanInstance()	Returns the current target instance of the bean business interface used for business interface method invocations. This API is provided for advanced use cases were direct access to the local/remote interfaces outside of the control is required. It will return `null` if no target instance is currently selected.
getEJBException()	Returns the last EJB exception serviced by the EJB control on the developers behalf. This can be used to discover or log additional information, for example when a create or find method is unable to locate a target bean instance.

4. Accessing EJBs on a Different Server

You can access EJBs on a different server with an EJB control, provided the server hosting the EJB control and the server to which the target EJB is deployed are in the same domain. You access EJBs on a different server by using special JNDI syntax in the `EJBHome` annotation's `jndiName` attribute.

For example:

```
@EJBHome(jndiName="jndi://username:password@host:7001/my.resource.jndi.object")
```

You can also use environment properties to specify configuration information, such as:

```
@EJBHome(jndiName="jndi://host:7001/MyEJBHome?SECURITY_PRINCIPAL=me&SECURITY_CREDENTIAL
```

5. Creating an EJB Control

The EJB Control is an extensible control, and you do not use it directly. To create an EJB control for an EJB, you would create a control extending the EJB Control. An extended EJB control can only represent one EJB, so you must create one for each EJB.

The following steps should be observed:

1. Create a Java interface extending the appropriate EJB Control interface. If the EJB is a session bean, extend `org.apache.beehive.controls.system.ejb.SessionEJBControl`, if it is an entity bean, extend `org.apache.beehive.controls.system.ejb.EntityEJBControl`.
2. Annotate the Java interface with @ControlExtension (`org.apache.beehive.controls.api.bean.ControlExtension`), so the Control Annotation Processor will know that the Java interface is a control extension.
3. Have the Java interface also extend the EJB's home and business interfaces. The business interface may either by the EJB's local interface or the remote interface.
4. Specifiy how the EJB control should lookup the EJB. To lookup the EJB by its JNDI name, set the EJB control's @EJBHome.jndiName annotation to the EJB's JNDI name. To lookup the EJB using an EJB link, set the EJB control's @EJBHome.ejbLink annotation to the name of the EJB link.
5. If the EJB control uses JNDI to look up an EJB, optionally specify the JNDI context environment properties using the @JNDIContextEnv annotation.

6. Using an EJB Control

After you have created an EJB Control, you can invoke an target EJB method via the EJB control. Specifically, the EJB control exposes all and only the EJB methods defined in the

EJB interfaces that the control extends. You can invoke these methods simply by invoking the method with the same signature on your EJB control.

The EJB control automatically manages locating and referencing the EJB instance, and directs method invocations to the correct instance of the target EJB. Whether or not you must first create an instance of the target EJB using the EJB's create method depends on whether the EJB control references a session or an entity bean.

Here is an example of the code required to invoke a single method on an exposed EJB using standard J2EE APIs:

```
Trader trader = null;
try {
     InitialContext ic = new InitialContext();
     TraderHome home = (TraderHome)ic.lookup("MyTraderBean");
     trader = home.create();
     TradeResult tradeResult = trader.buy(stock, shares);
     return  tradeResult;
}
catch (NamingException e) {
     ...
}
catch (CreateException e) {
     ...
}
catch (RemoteException e) {
     ...
}
finally {
    if (trader != null)
        trader.remove();
}
```

The code can be reduced to the following using the EJB Control:

```
@Control
TraderControlBean traderControl;

try {
     TradeResult tradeResult = traderControl.buy(stock, shares);
     return  tradeResult;
}
catch (RemoteException re) {
     ...
}
finally {
    if (traderControl != null)
        traderControl.remove();
}
```

7. Selecting Instances for Session EJBs

A session EJB is used to execute business tasks for a client on the application server. The session EJB might execute only a single method for a client, in the case of stateless session beans, or it might execute several methods for that same client, in the case of stateful session beans. A session bean never serves multiple clients at the same time. The lifetime of a stateful session bean is tied to the duration of the conversation with the client. In contrast, a small number of pooled stateless session bean instances is used to serve large number of client requests.

7.1. Creating a Session EJB

If the target EJB is a stateless session bean, you do not need to invoke the create method of the EJB via the EJB control. Instead, the EJB control automatically creates a reference to an appropriate instance of the EJB whenever one of the EJB's business methods is invoked, as is shown in this code fragment:

```
@Control()
private EJBControls.MusicBeanControl library;
...
// create method is not invoked first
allBands = library.getBands();
```

If the target EJB is a stateful session bean you must first invoke (one of) its create method(s) to obtain a reference.

7.2. Caching a Session EJB Reference

After a reference is obtained, it is cached in the EJB control and is used for any subsequent calls to the EJB within the method invocation in which the initial call was made. If for a stateless session bean you explicitly call the create method, or if for a stateful session bean you again call a create method, the EJB control replaces a previously cached reference with the newly created reference.

The lifetime of the cached EJB reference within the EJB control depends on the invoking application and the type of session bean. If a stateful session EJB is invoked by a conversational web service (that is, a web service method that takes part in a conversation), the EJB reference's lifetime is the lifetime of the conversation. If a stateful or stateless session EJB is invoked by a non-conversational web service, the lifetime of the EJB reference in the EJB control is the lifetime of the web service method invocation. For page flows and both stateful and stateless session EJBs, if the session EJB is defined in the controller class, the lifetime of the reference is the lifetime of the page flow.

7.3. Removing a Session EJB

When you call the remove method on an EJB control that represents a session EJB, the currently cached instance of the bean is released. The server might destroy the bean at that time, but the actual behavior is up to the server. Either way, the bean no longer communicates with the EJB control.

8. Selecting Instances for Entity Beans

Instances of entity EJBs are associated with a particular collection of data. Typically this collection of data is a row in a database table.

8.1. Creating an Entity EJB

When you invoke the EJB's create method through the EJB control, you create a new persistent entity, that is, a new record in the underlying database table. In other words, creating a new entity bean with the create method amounts to inserting a new record in a table.

8.2. Referencing an Entity EJB

You can reference an entity EJB instance by calling the findByPrimaryKey method, or another findXxx method provided by the EJB's designer that returns a reference to one entity bean. In other words, the entity bean instance represents an existing record in a database table.

8.3. Caching an Entity EJB Reference

The EJB control caches a reference to the EJB instance being used, that is, the instance returned by the most recent call to the create, `findByPrimaryKey` or `findXxx` method, which returns one data record. When you invoke subsequent methods on the EJB control, it invokes that method on the EJB instance to which the cached reference refers. If there is no EJB reference currently cached, the EJB control attempts to invoke the `findByPrimaryKey` method with the last successful key used in a create or `findByPrimaryKey` call. If there is no previous key, the EJB control throws an exception.

The lifetime of the cached entity EJB reference within the EJB control depends on the invoking application. If the entity EJB is invoked by a conversational web service (that is, a web service method that takes part in a conversation), the EJB reference's lifetime is the lifetime of the conversation. For non-conversational web services, the lifetime of the EJB

reference in the EJB control is the lifetime of the web service method invocation. For page flows, if the entity EJB is defined in the controller class, the lifetime of the reference is the lifetime of the page flow.

8.4. Removing an Entity EJB

When you call the remove method on an EJB control that represents an entity EJB, the record represented by the cached EJB reference is removed from the underlying persistent storage. That is, the row is deleted from the database table.

8.5. Returning Multiple Records

A findXxx method may return a Collection object, holding a set of references to entity beans. The EJB control does not cache this object. If you wish to cache the return value of a findXxx method, you should store the object in a member variable of the application invoking the EJB control.

9. Handling EJB Exceptions

The EJB control makes it easy to use an existing, deployed EJB from within an application. This topic describes how to handle exceptions that might be thrown by the target EJB or the EJB control itself.

9.1. Checked Exceptions

If the target EJB method invoked via an EJB control throws a checked exception (that is, an exception that does not extend a RuntimeException), a try-catch block must catch the exception. It is generally considered a best practice to catch exceptions that occur within the EJB method (thrown by other methods the EJB method uses), and either overcome the exception or, when this is impossible, rethrow these exceptions either as an application exception or an EJBException, depending on whether the failure is due to a system-level or business logic error, and whether the transaction should be automatically rolled back.

An application exception is a checked exception that is either defined by the bean developer and does not extend a RemoteException, or is predefined in the javax.ejb package (that is, CreateException, DuplicateKeyException, FinderException, ObjectNotFoundException, or RemoveException). The EJB's method explicitly defines any application exception in the throws statement, and a try-catch block must catch the exception.

If the EJB control uses the EJB control's remote interfaces, a RunTimeException or an EJBException thrown by the EJB method is nested by the EJB container inside a

RemoteException, and the RemoteException is propagated to the client. Because a RemoteException is a checked exception, the client must catch the exception. For more information on RemoteException, see your favorite J2EE reference documentation or the J2SE API documentation at http://java.sun.com. An example of catching a RemoteException is given below.

9.2. Runtime Exceptions

A java.lang.RuntimeException and its subtypes, including EJBException, can be thrown by an EJB method via its corresponding EJB control. Although these exceptions do not have to be explicitly caught in your code, it is generally a good idea to catch these exceptions in the client application invoking an EJB control, which uses the EJB's local interfaces. (Remember that for remote interfaces, the EJBException is rethrown by the EJB container as a RemoteException.)

As mentioned above, a checked exceptions caught in a bean method is often rethrown as an EJBException. You can checked for such a nested exception by invoking the getCause() or getCausedByException() methods on the caught EJBException. An example of nesting and catching an exception through EJBException is given below.

The EJB control will throw a `org.apache.beehive.controls.api.ControlException` when it has a problem locating/referencing the EJB. Although this is a subtype of RuntimeException and therefore does not have to be caught explicitly, it again might be a good idea to catch it in the client application.

9.3. A Nested Exception Example

The following example demonstrates the rethrowing of a checked exception inside an EJBException by an EJB method, and the catching of this exception on the client side. Rethrowing the exception inside an EJBException in the example is done solely to illustrate the mechanics of exception handling, and should not be considered recommended design. As mentioned above, checked exceptions should be rethrown either as an application exception or an EJBException, depending on whether the failure is due to a system-level or business logic error, and whether the transaction should be automatically rolled back.

The first code snippet shows the definition of the MusicBean method addRecording. Notice that FinderException, which can be thrown by findByPrimaryKey, is rethrown inside an EJBException, and that CreateException, which can be thrown by the BandBean's business method addThisRecording, is rethown inside an EJBException:

```
  public void addRecording(String band, String recording)
```

```
{

   try {
      Band bandBean = bandHome.findByPrimaryKey(new BandPK(band));
      if(bandBean != null) {
         bandBean.addThisRecording(recording);
      }
   }
   catch(CreateException ce) {
      throw (EJBException) new EJBException(ce).initCause(ce);
   }
   catch(FinderException fe) {
      throw (EJBException) new EJBException(fe).initCause(fe);
   }
}
```

On the client side, the MusicBean's method is invoked via an EJB control, and an EJBException is caught and checked. If the client uses an EJB control that locates the EJB via its local interfaces, you can catch the EJBException directly and retrieve the nested exception. The following code snippet shows how this is done in a page flow's action method, using an EJB control that locates the EJB via its local interfaces:

```
@Control()
private EJBControls.MusicBeanControl library;

...

@Jpf.Action(
     forwards = {
        @Jpf.Forward(name="success", path="addRecording.jsp")
     }
)
protected Forward addARecording(AddARecordingForm form)
{
   String recording = (form.getRecordingName()).trim();
   String bandChoice = form.getSelectedBand();
   if(recording.length() != 0) {
      try {
         library.addRecording(bandChoice, recording);
         allRecordings = library.getRecordings(bandChoice);
      }
      catch(EJBException ee) {
         Exception ne = (Exception) ee.getCause();
         if(ne.getClass().getName().equals("FinderException"))
            ...
         else if(...)
            ...
      }
   }
   ...
   return new Forward("success");
}
```

If the client uses an EJB control that references the EJB via its remote interfaces, you must catch the RemoteException instead and retrieve its nested exception. The following code snippet shows how this is done:

```
   @Control()
   private EJBControls.RemoteMusicBeanControl remoteLibrary;

   ...

   @Jpf.Action(
        forwards = {
           @Jpf.Forward(name="success", path="addRecording.jsp")
        }
   }
   protected Forward addARecording(AddARecordingForm form)
   {
     String recording = (form.getRecordingName()).trim();
     String bandChoice = form.getSelectedBand();
     if(recording.length() != 0) {
        try {
           remoteLibrary.addRecording(bandChoice, recording);
           allRecordings = library.getRecordings(bandChoice);
        }
        catch(RemoteException re) {
           EJBException ee = (EJBException) re.getCause();
           Exception ne = (Exception) ee.getCause();
           ...
        }
    }
    ...
    return new Forward("success");
  }
```