BEA WebLogic Workshop™ Help

Version 8.1 SP4
December 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Table of Contents

Developing Enterprise JavaBeans....................1

Getting Started with EJB Project....................3

Developing Entity Beans....................7

Getting Started with Entity Beans....................9

Defining an Entity Bean....................13

Automatic Primary Key Generation....................19

Entity Relationships....................22

Query Methods and EJB QL....................27

The Life Cycle of an Entity Bean....................34

Accelerating Entity Bean Data Access....................38

Bean−Managed Persistence....................48

Developing Session Beans....................50

Getting Started with Session Beans....................51

Defining a Session Bean....................54

The Life Cycle of a Session Bean....................57

Developing Message−Driven Beans....................61

Getting Started with Message−Driven Beans....................62

Sending JMS Messages....................65

Processing JMS Messages....................69

Advanced Enterprise JavaBeans Topics....................71

Switching EJB Development to WebLogic Workshop....................72

EJBGen Tag Inheritance....................74

EJBs and Transactions....................77

Table of Contents

Summary of IDE Windows for EJB Project...81

Developing Enterprise JavaBeans

These topics show how to develop Enterprise JavaBeans. WebLogic Workshop provides you with the tools to make EJB development much easier, taking care of many implementation details for you and allowing you to focus on design.

Note. WebLogic Workshop 8.1 supports the development, importing, and building of EJB2.0 compliant Enterprise JavaBeans. In addition, the WebLogic Platform supports the deployment only of EJB1.1 compliant Enterprise JavaBeans. For more information on deploying existing EJBs, see How Do I: Add an Existing Enterprise JavaBean to an Application? For more information on EJB specifications, see http://java.sun.com/products/ejb/index.jsp.

Topics Included in This Section

Getting Started with EJB Project

Provides an overview of Enterprise JavaBeans and EJB Project, the role of ejbgen tags, and EJB controls.

Developing Entity Beans

This topics discusses the development of entity beans.

Developing Session Beans

This topics discusses the development of session beans.

Developing Message−Driven Beans

This topics discusses the development of message−driven beans.

Advanced Enterprise JavaBeans Topics

This topics discusses advanced Enterprise JavaBean development issues.

Summary of IDE Windows for EJB Project

Introduces new users of WebLogic Workshop to the various windows and panes using during EJB development.

Related Topics

Getting Started: Enterprise JavaBeans

This tutorial provides a quick introduction to developing EJBs.

Tutorial: Enterprise JavaBeans

This advanced tutorial provides a step−by−step guide to developing Enterprise JavaBeans.

Enterprise JavaBean Samples

This topic presents a collection of Enterprise JavaBean samples. Each sample describes a small set of EJB development features and techniques.

Enterprise JavaBeans

This 'How Do I...?' section presents a number of step−by−step procedures on how to create Enterprise JavaBeans with EJB Project.

Enterprise JavaBean Annotations Reference

This topic provides reference documentation about the ejbgen tags used with Enterprise JavaBeans classes.

Getting Started with EJB Project

When you are building WebLogic platform applications, EJB project is the development environment for Enterprise JavaBeans. EJB project provides a number of tools that facilitate the development of the EJBs which encompass the business logic for your enterprise application. This topic provides an overview of EJBs and EJB project in platform applications. It includes the following sections:

- What is an Enterprise JavaBean?
- What is EJB Project?
- EJB Project and ejbgen Tags
- Building and Deploying EJBs
- What are EJB Controls?

What is an Enterprise JavaBean?

An EJB is a server−side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application, as opposed to code that provides infrastructure and plumbing for the application. In an inventory control application, for example, the EJBs might implement the business logic in methods called checkInventoryLevel and orderProduct. By invoking these methods, remote clients can access the inventory services provided by the application.

EJBs always execute within an EJB container, which provides system services to EJBs. These services include transaction management, persistence, pooling, clustering and other aspects of infrastructure. The J2EE and EJB architecture is built on a number of underlying technology standards, such as the JDBC API for database connectivity, JMS for messaging, and JNDI for naming and directory functionality. To learn more about these technology standards, see your favority J2EE documentation and http://java.sun.com.

In J2EE 1.4 there are three types of EJBs: session, entity, and message−driven. Each of these types is described briefly in the following sections.

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 is never shared by clients. A session EJB is not persistent, so when the client terminates, its session EJB disconnects and is no longer associated with the client.

Entity EJBs

An entity EJB represents a business object in a persistent storage mechanism. Some examples of business objects are customers, orders, and products. The persistent storage mechanism is a relational database. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table. Unlike session beans, entity beans are persistent, allow shared access, have primary keys, and may participate in relationships with other entity beans.

Message−Driven EJBs

A message−driven EJB is an enterprise bean that is able to listen for Java Message Service (JMS) messages. The messages may be sent by any JMS−compliant component or application. Message−driven EJBs provide a mechanism for J2EE applications to participate in relationships with message−based legacy applications.

EJB Interfaces

EJB 2.0 exposes four types of interfaces for session and entity beans, called the local home interface, the local business interface (or simply, the local interface), the remote home interface, and the remote business interface (or simply, the remote interface). Client applications can obtain an instance of the EJB with which to communicate by using the remote home interface. The methods in the remote home interface are limited to those that create or find EJB instances. Once a client has an EJB instance, it can invoke methods of the EJB's remote business interface to do real work. The business interface directly accesses the business logic encapsulated in the EJB. Interactions between EJBs defined in the same WebLogic Workshop application, as well as interactions between EJBs and web services or page flows in the same WebLogic Workshop application, can use the local interfaces instead, which provides a performance advantage over remote interfaces. In other words, the local home and business interfaces define the methods that can be accessed by other beans, EJB controls, web services, and page flows in the same WebLogic Workshop application, while the remote home and business interfaces define the methods that can be accessed by other applications.

Message−driven beans do not have these interfaces, because these beans' methods do not get invoked directly by other beans or client applications. Instead they process messages from client applications or other EJBs that are delivered via the Java Message Service (JMS). When a message is delivered, the EJB container calls the message−driven bean's onMessage method to process the message. For more information on Java Message Service, see your favorite J2EE documentation. WebLogic Workshop also provides JMS controls to work with a Java Message Service. For more information, see JMS Controls.

Deployment Descriptor

During runtime, information about how EJBs should be managed by the EJB container is read from a deployment descriptor. The deployment descriptor describes the various beans packaged in an EJB JAR file, settings related to transaction management, and EJB QL for find methods, to name a few examples. Deployment descriptor settings can be changed without having to make changes to the actual beans, allowing for the fine−tuning of EJBs.

EJB JAR

The EJB JAR contains one or more EJBs, including their interface definitions, any related Java classes that are being used by the EJBs, and a deployment descriptor describing these EJBs.

What is EJB project?

EJB project is the development environment for session, entity, and message−driven beans in a WebLogic platform application. An EJB project contains one or more EJBs. A WebLogic platform application can have one or more EJB projects, as well as other types of projects such as web (service) projects, and Java control projects, thereby creating an integrated development environment.

EJB project provides a Design View, property editor, building and debugging windows, and various wizards to facilitate the design and development of EJBs. For instance, in Design View you can define business (component) methods, define CMP fields, and define methods for local and/or remote interfaces. A property editor is used to edit deployment descriptor settings and ejbgen tag settings (see below). Various wizards make it easy to import EJBs into EJB projects, create ejbCreate methods, and define entity relationships. Build and debugging windows are used to monitor the build process and to monitor the EJB when running/debugging the application. For more information on the visual environment, see Summary of IDE Windows for EJB Project. For step−by−step instructions on using the Design View and the various wizards, see the 'How Do I...?' section Enterprise JavaBeans.

A key advantage of developing EJBs in EJB project is that one file is used to store the definition of the EJB class, its interfaces, and deployment descriptor specific settings. Instead of managing the overhead of using several java files to store this information, one file with an *ejb* extension is used to represent an EJB. This is accomplished by using ejbGen, which is described next.

EJB Project and ejbgen Tags

During Enterprise JavaBean development, special Javadoc tags with the prefix ejbgen are inserted in the EJB source file. These ejbgen tags are used to mark methods to be exposed in remote and home interfaces, store deployment descriptor settings, and various other types of information. In many cases you will not have to add these ejbgen tags directly. Instead these tags are added when, for example, you define a business methods to be local, or create a primary key field for an entity bean. You can also use the property editor to edit ejbgen tag settings.

The use of a single EJB file is only for development purposes. When you build an EJB Project, the build process uses these tags to generate the various interfaces and deployment descriptor files as prescribed in the J2EE specification.

Building and Deploying EJBs

When you build an EJB project, the EJBs' source code is compiled and checked for errors. The build output of an EJB project is an EJB JAR, containing the various JAVA and CLASS files for the bean class, its interfaces, and any dependent value or primary key classes, as well as the deployment descriptor for these beans. After the build completes, the beans are (re)deployed on the server. You can also deploy, undeploy and redeploy EJBs separately from building.

What are EJB Controls?

When you want to use an EJB from a client application (such as a page flow), you must look up the EJB in the JNDI registry and obtain the EJB's home interface, before you can create an EJB instance and invoke the EJB's business methods. EJB controls provide an alternative approach to make locating and referencing the EJB easy. Once you have created the EJB control, the client application can define the EJB control and use its methods, which have the same method names as the EJB it controls, to execute the desired business logic without having to be involved with locating and referencing the EJB itself. In other words, EJB controls take care of the prepatory work necessary to use an EJB, allowing you to focus on the business logic instead. For more information, see EJB Controls.

See Related Topics

Getting Started: Enterprise JavaBeans

The WebLogic Workshop Development Environment

Getting Started with Java Controls

Enterprise JavaBean Annotations Reference

How Do I: Create an EJB Project?

How Do I: Create an Enterprise JavaBean?

Developing Entity Beans

An entity EJB models a real−world business object that persists as a record in a relational database. Entity beans are persistent, allow shared access, have primary keys, and may participate in relationships with other entity beans. All topics listed below discuss development of container−managed persistence (CMP) entity beans, with the exception of the last topic which discusses bean−managed persistence entity beans.

Topics Included in This Section

Getting Started with Entity Beans

This topic provides an overview of entity beans.

Defining an Entity Bean

This topic discusses how to create an entity bean in WebLogic, what an entity bean definition minimally must contain, how to remove a bean instance, and provides a short introduction to the various interfaces extended/implemented by an entity bean definition.

Automatic Primary Key Generation

This topic discusses how to auto−generate primary keys when creating a new entity bean.

Entity Relationships

This topic discusses how to define an entity relationship between two CMP entity beans.

Query Methods and EJB QL

This topic discusses how to use EJB QL and WebLogic QL in the definition of CMP select and find methods.

The Life Cycle of an Entity Bean

This topic discusses the life cycle of an entity bean.

Accelerating Entity Bean Data Access

This topic discusses a number of WebLogic−specific features to reduce data access latency for CMP entity beans.

Bean−Managed Persistence

This topic introduces bean−managed persistence (BMP) entity beans.

Related Topics

Tutorial: Enterprise JavaBeans

This tutorial provides a step−by−step guide to developing Enterprise JavaBeans.

Enterprise JavaBean Samples

This topic presents a collection of Enterprise JavaBean samples. Each sample describes a small set of EJB development features and techniques.

Enterprise JavaBeans

This 'How Do I...?' section presents a number of step−by−step procedures on how to develop Enterprise JavaBeans.

Enterprise JavaBean Annotations Reference

This topic provides reference documentation about the ejbgen tags used with Enterprise JavaBeans classes.

Getting Started with Entity Beans

This topic provides an overview of CMP entity beans development. It contains the following sections:

- What are CMP Entity Beans?
- Home and Business Interfaces
- CMP Fields
- Create Methods
- Component Methods
- Home Methods
- Finder and Select Methods
- Relations
- Other Methods

What are CMP Entity Beans?

An entity bean represents a business object in a persistent storage mechanism. In other words, entity beans are used to model real−world objects with properties that need to be stored and remembered over time. Examples of business objects are customers, products, orders, credit cards, and addresses. Typically, each entity bean has an underlying table in a relational database, and each bean instance corresponds to a row in that table. An entity bean has one or more primary keys, or unique indices, to uniquely identify an object, that is, a particular record in the database.

Container−managed persistence (CMP) entity beans are entity beans for which the EJB container takes care of mapping property and relationship fields to the underlying database and knows how to insert, update and delete data for an entity bean. In contrast, for bean−managed persistence (BMP) entity beans you must write the code to ensure database connectivity and data storage. BMP entity beans are discussed separately in the topic Bean−managed persistence.

Home and Business Interfaces

An entity bean can have four different interfaces, called the local home interface, the local business interface (or simply, the local interface), the remote home interface, and the remote business interface (or simply, the remote interface). The local interfaces define the bean's methods that can be used by other EJBs, EJB controls, web services and page flows defined within the same application. That is, if you define an entity bean and only plan to use it within that application, you can use local interfaces. In contrast, the remote interfaces define the bean's methods that be invoked by EJBs, EJB controls, web services and page flows defined in other applications.

To determine what interfaces are defined for an entity bean, ensure you are in Design View and go to the ***Naming*** section in the ***Property Editor***. The *Remote EJB* and *Local EJB* sections refer to the remote and local interfaces; within each section the *Home class name* refers to the home interface, and the *Bean class name* refers to the business interface. In Source View, the attributes are part of the @ejbgen:file−generation Annotation. When you define a new entity bean, by default only the local interfaces are defined, reflecting the design assumption that in your model entity beans are not accessed directly by other client applications but indirectly, with session or message−driven beans defined in the same application as intermediaries.

Client applications and other session or entity beans can obtain an instance of an entity bean with which to communicate by using methods in the (remote or local) home interface. Methods in the home interface include create methods, the findByPrimaryKey method, and other finder methods that return a single reference or a set of references to entity bean instances. In addition, *Home* methods are defined in the local interface. The (remote or local) business interface contains the methods that manipulate an entity bean instance. These methods include field accessor (getter and setter) methods and component methods.

CMP Fields

Container−managed persistence (CMP) fields contain the business object's properties. For example, a Customer bean might contain first name, last price, gender, and age fields. Because the EJB container takes care of the mapping of these properties to a database, CMP fields are virtual fields in an entity bean; that is, these fields are not defined in the entity bean itself but correspond to columns in the database table. The entity bean only defines the accessor (getter and setter) methods. A CMP field can serve as a primary key field, meaning that this field uniquely identifies the entity bean instance or, if multiple primary keys are defined, in combination with the other primary keys uniquely identifies the entity bean instance. For more information, see How Do I: Define a Container−Managed Persistence (CMP) Field?

Create Methods

An ejbCreate method is used to create a new instance of an entity bean, that is, insert a new record in the underlying database. At least one ejbCreate method must be defined, but multiple ejbCreate methods are not uncommon. Each ejbCreate method defined for an entity bean has the signature *public <prim−key−class> ejbCreate(parameters)*. The <prim−key−class> can be a primitive type, such as Integer, when a single primary key is defined, or it can be a separate primary key class. By default, WebLogic auto−generates a primary key class, with the name provided as an attribute in the @ejbgen:file−generation Annotation. To find out which primary key class is used for an entity bean, ensure you are in Design View and go to the ***General*** section in the ***Property Editor***. In Source View, this attribute is part of the @ejbgen:entity Annotation.

Multiple ejbCreate methods can only be distinguished by their parameter composition. In the home interface, ejbCreate methods are exposed as create methods and can correspondingly be distinguished only by the unique set of parameters each one requires. For more information, see How Do I: Add a Create Method to an Entity Bean?

Component Methods

Component methods are the business methods that are invoked on an entity bean instance. A simple example of a business method is updateCustomer(firstName, lastName, age), which is used to update the customer's first name, last name and age. This method will in turn invoke the bean's setFirstName, setLastName and setAge methods to update the CMP fields holding this information. For more information, see How Do I: Add a Component Method to an Entity or Session Bean?

Home Methods

A home method is a business method that relates to the entity bean but is not specific to a single bean instance. For instance, a Customer bean might have a home method returning the total number of customers between 25 and 35 years of age. A home method is defined as a *ejbHome<method−name>* method in the bean, and is exposed as a *<method−name>* method on the bean's home interface. For example the method ejbHomeGetNCustomers defined in the bean class is exposed as getNCustomers in its home interface. For more information, see How Do I: Add a Home Method to an Entity or Session Bean? and the Home Methods Sample.

Finder and Select Methods

Finder and select methods are methods that execute queries on the database, using the EJB QL or WebLogic QL query languages. A finder method is defined in the bean's home interface and returns a reference to a single bean instance or to a set of references to bean instances. In contrast, a select method is not defined in any interface and can only be invoked internally, for instance by a bean's component method. A select method can return a reference to a single bean instance, a set of bean instances, or one or more individual CMP fields.

In addition to the finder and select methods defined for the bean, the method findByPrimaryKey(<prim−key−class>) is automatically defined by WebLogic in the home interface(s) defined for the bean class. This method returns a reference to the bean instance that is unique defined by the method's parameter. As before, the <prim−key−class> can be a primitive type, such as Integer, when a single primary key is defined, or it can be a separate primary key class. To find out which primary key class is used for an entity bean, ensure you are in Design View and go to the ***General*** section in the ***Property Editor***. In Source View, this attribute is part of the @ejbgen:entity Annotation.

For more information on how to define finder or select methods, see How Do I: Add a Finder Method to an Entity Bean? and How Do I: Add a Select Method to an Entity Bean? For more information on the EJB QL and WebLogic QL query languages, see Query Methods and EJB QL, the Finder Methods Sample, and the Select Methods Sample.

Relations

Entity relationships are used to model dependencies between business objects. For example, a customer can have one or more credit cards, and a product has a manufacturer. Relations between two entity beans can be defined such that for a customer, you can easily access its credit cards by using the accessor method getCreditCards. The entity relation accessor methods, also known as the CMR field accessor methods are defined in the bean's business interface. For more information, see Entity Relationships, How Do I: Add a Relation to an Entity Bean?, and Tutorial: Enterprise JavaBeans.

Other Methods

An entity bean has several predefined methods, as well as a number of callback methods, invoked by the EJB container during certain operations, that an entity bean must implement. In WebLogic these callback methods are by default automatically implemented. In many cases you will find it unnecessary to use these methods, with the possible exception of the remove methods used to remove an entity bean instance. To learn more about predefined methods and remove methods in particular, see Defining an Entity Bean. To learn more about callback methods, see the Callback Methods section of that same topic and The Life Cycle of an Entity Bean.

Related Topics

Automatic Primary Key Generation Sample

Finder Methods Sample

Home Methods Sample

Select Methods Sample

Value Object Sample

@ejbgen:file−generation Annotation

@ejbgen:entity Annotation

Defining an Entity Bean

This topic discusses how to create an entity bean and what an entity bean minimally must contain. Furthermore, it describes how to remove a bean instance, and discusses the various interfaces extended by an entity bean. This topic contains the following sections:

- Creating an Entity Bean
- Defining a Basic Entity Bean
- Removing an Entity Bean Instance
- Callback Methods

Creating an Entity Bean

To create an entity bean, you can choose one of the following methods:

- ***Create an entity bean from scratch***. If you are designing a new entity bean in EJB project, you can define a new entity bean. To find out exactly how to do this, see How Do I: Create an Enterprise JavaBean? When you create a new entity bean, by default the local interfaces and various other defaults are defined. For more details, see @ejbgen:file−generation Annotation.
- ***Create an entity bean from a database table***. If you have already designed a database table and want to create an entity bean with CMP fields mapped to the database columns, you can create an entity bean from a database table. To find out exactly how to do this, see How Do I: Generate an Entity Bean from a Database Table? In addition to mapping the CMP fields and defining an ejbCreate method with the (compound) primary key as the parameter, by default local interfaces and various other defaults are defined. For more details, see @ejbgen:file−generation Annotation. After you have created an entity bean from a database table, you can further expand its definition, including making changes that will require changes to the underlying database table.
- ***Import an entity bean***. If you have developed entity beans in another development environment, you can import these into WebLogic. To import an EJB, you need the EJB Jar file as well as the source files. You can import multiple EJBs at the same time. If you import multiple entity beans with defined entity relations, these relation definitions will be imported as well. For more information, see How Do I: Import an Enterprise JavaBean? After you have imported an entity bean, you can enhance its definition.

  Note. If you have existing entity beans that you plan to invoke in the application, for instance via another EJB or an EJB control, but you do not intend to change their definitions, you can suffice by adding the EJB Jar to the application. For more information, see How Do I: Add an Existing Enterprise JavaBean to an Application?

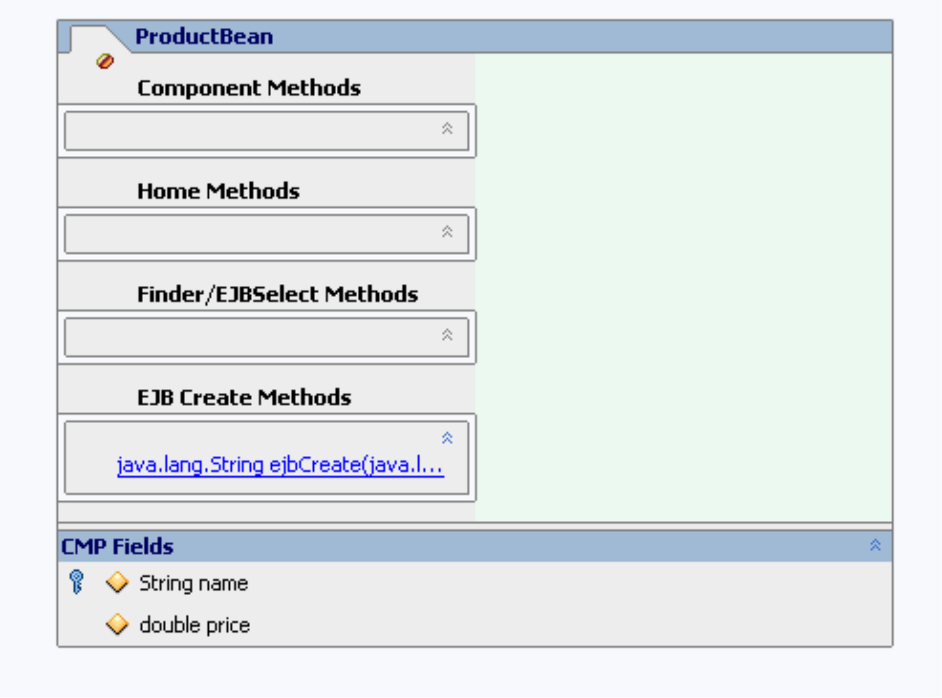
Automatic Table Creation

When you are developing a new entity bean using any of the three methods specified above, you can make iterative development easier by enabling automatic table creation. You can have WebLogic create the table when it is not present, and you can allow WebLogic to drop an existing table and recreate it if the definition of the entity bean does not match the table specifications. You enable automatic table creation using the create−table property located in the ***JAR Settings*** section of the ***Property Editor***. The default setting is CreateOnly. To recreate a table if the definition of the entity bean does not match the table specification, use DropAndCreate. For more information regarding the possible settings, see @ejbgen:jar−settings Annotation.

Note. Automatic table creation is meant to facilitate development, and is disabled in production mode.

Defining a Basic Entity Bean

The following figure shows the design view of a basic entity bean called *ProductBean*:



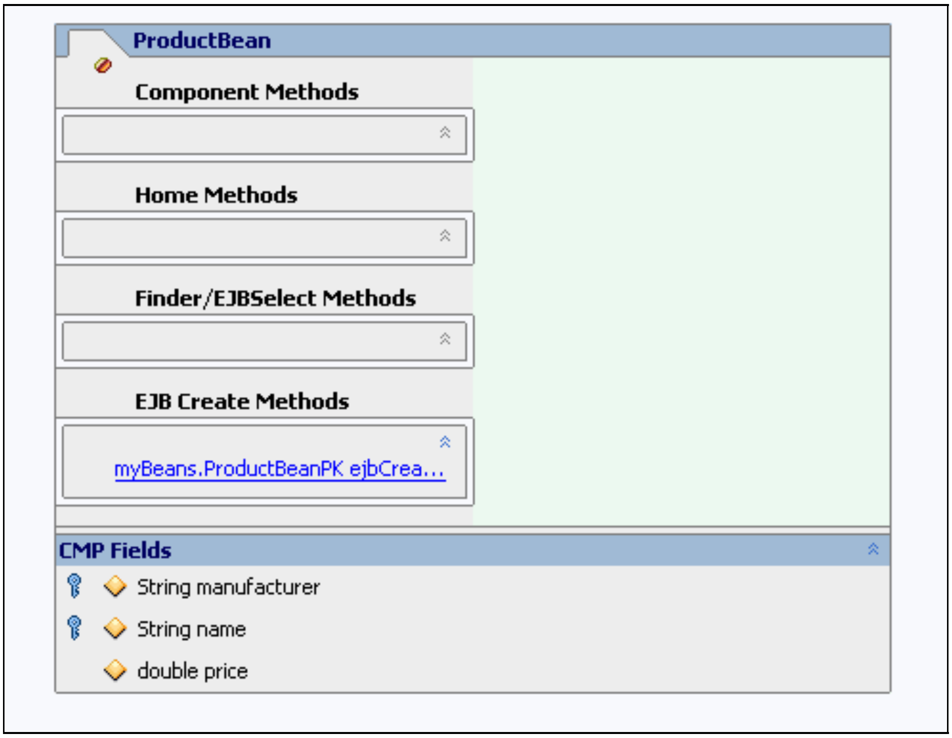
This bean was developed by defining it from scratch, adding the two CMP fields, labeling one of these as the primary key, and creating an ejbCreate method (for more details, see How Do I: Add a Create Method to an Entity Bean?). These steps were accomplished in design view. This entity bean allows you to add a new product, find a product using its primary key, and getting and setting its CMP field values. The source code of this bean is given below:

```
package myBeans;

import javax.ejb.*;
import weblogic.ejb.*;

/**
 * @ejbgen:entity prim-key-class="java.lang.String"
 *   ejb-name = "Product"
 *   data-source-name="cgSampleDataSource"
 *   table-name = "product"
 *   abstract-schema-name = "Product"
 *
 * @ejbgen:jndi-name
 *   local  = "ejb.ProductLocalHome"
 *
 * @ejbgen:file-generation local-class = "True" local-class-name = "Product" local-home = "True
 *    local-home-name = "ProductHome" remote-class = "False" remote-home = "False"  remote-home
 *    remote-class-name = "ProductRemote" value-class = "False" value-class-name = "ProductValu
 *
 */
abstract public class ProductBean extends GenericEntityBean implements EntityBean
```

```
{

    /**
     * @ejbgen:cmp−field primkey−field="true" column="Name"
     * @ejbgen:local−method
     */
    public abstract String getName();

    /**
     * @ejbgen:local−method
     */
    public abstract void setName(String arg);

    /**
     * @ejbgen:cmp−field column="Price"
     * @ejbgen:local−method
     */
    public abstract double getPrice();

    /**
     * @ejbgen:local−method
     */
    public abstract void setPrice(double arg);

    public java.lang.String ejbCreate(java.lang.String Name, double Price)
    {
      setName(Name);
      setPrice(Price);

      return null; // FIXME return PK value
    }

    public void ejbPostCreate(java.lang.String Name, double Price)
    {
    }
}
```

In WebLogic, all the information needed to make an entity bean is stored in a single file, instead of separate JAVA files for the bean class, the local business interface, the local home interface, its primary key class, and so forth. When you build an EJB, these other classes are auto−generated. Various ejbgen annotations are used to hold the information required to make this generation possible. For example, the @ejbgen:file−generation annotation specifies the names of the local home and business interface for the ProductBean. The @ejbgen:local−method annotations on the accessor methods specify that these methods are defined in the local business interface. To verify that these JAVA and corresponding CLASS files are generated, expand the JAR file created during a build (located in the ***Modules*** folder in the ***Application*** pane), and locate and open the generated files in the folder reflecting the package name. For the ProductBean, the ProductBean.java (bean definition), Product.java (local interface definition), and ProductHome.java (local home interface definition) files are auto−generated.

If the ProductBean were to use multiple primary keys, the ProductBeanPK.java file containing the definition of the compound primary key class would also be auto−generated. For instance, the following figure shows a redefined ProductBean with an additional primary key manufacturer:



Notice in the above figure that the compound primary key is the return value of the ejbCreate method. Also, the auto−generated method findByPrimaryKey(myBeans.ProductBeanPK primaryKey) uses the compound primary key to obtain a bean instance reference.

Finally a value object class can also be auto−generated. For more information, see the Value Object Sample

Removing an Entity Bean Instance

Any entity bean must define at least one ejbCreate method to create a new instance. Also, the EJB container automatically defines the findByPrimaryKey method in the home interface(s), which return a bean instance using its primary key (class) as the method parameter. In addition, all the bean's interfaces will extend a particular interface which contains various useful methods. Specifically:

- The local interface extends javax.ejb.EJBLocalObject
- The local home interface extends javax.ejb.EJBLocalHome
- The remote interface extends javax.ejb.EJBObject
- The remote home interface extends javax.ejb.EJBHome

Complete details about these interfaces and the methods they define can be found in your favorite J2EE documentation and the API reference at http://java.sun.com. One of the more frequently used methods provided by these 'extended' interfaces is a remove method, used to remove an entity bean instance. In other words, when you invoke the remove method on an entity bean, you remove the bean and the underlying record in the database. Remove methods are defined in all the interfaces. For instance, to remove a bean instance via the local home interface, you can invoke a remove method that takes instance's primary key as the parameter. To remove a bean instance via the local interface, you can invoked the remove method for the instance you want to remove. Both approaches are shown below; the session bean's method deleteViaHome deletes an instance of the Product bean via its local home interface, while deleteViaBusiness delete a Product bean instance via the local interface:

```
/**
 * ...
 *
 * @ejbgen:ejb-local-ref link="Product"
 */
public class SomeSession extends GenericSessionBean implements SessionBean
{
   ...

    /**
     * @ejbgen:local-method
     */
    public void deleteViaHome(myBeans.ProductBeanPK thePk)
    {
       try {
          javax.naming.Context ic = new InitialContext();
          ProductHome productHome = (ProductHome)ic.lookup("java:comp/env/ejb/Product");
          productHome.remove(thePk);
       }
       catch(NamingException ne) {
          ...
       }
       catch(RemoveException re) {
          ...
       }
    }

   /**
     * @ejbgen:local-method
     */
    public void deleteViaBusiness(myBeans.ProductBeanPK thePk)
    {
       try {
          javax.naming.Context ic = new InitialContext();
          ProductHome productHome = (ProductHome)ic.lookup("java:comp/env/ejb/Product");
          Product theProduct = productHome.findByPrimaryKey(thePk);
          theProduct.remove();
       }
       catch(NamingException ne) {
          ...
       }
       catch(FinderException ne) {
          ...
       }
       catch(RemoveException re) {
          ...
       }
    }

   ...
}
```

Callback Methods

Every entity bean must implement the javax.ejb.EntityBean interface. This interface defines callback methods that are called by the EJB container at specific times. The callback methods are setEntityContext, unsetEntityContext, ejbActivate, ejbPassivate, ejbLoad, ejbStore, and ejbRemove. When you define an entity bean from scratch or via a database table, it will extend weblogic.ejb.GenericEntityBean, which contains empty implementations of these callback methods. In other words, you will only need to define these methods if you need to override the empty implementation. If you import an entity bean, these callback methods will probably be implemented directly in the bean's ejb file.

For more details about the callback methods and their role in the interaction between the entity bean and the EJB container, see The Life Cycle of an Entity Bean.

Related Topics

The Life Cycle of an Entity Bean

@ejbgen:file−generation Annotation

@ejbgen:local−method Annotation

How Do I: Add a Create Method to an Entity Bean?

Automatic Primary Key Generation

In WebLogic it is possible to automatically generate a primary key to be used when creating a new CMP entity bean instead of providing primary key values. You can auto−generate primary keys in various vendor−specific ways − using Oracle, SQLServer, or SQLServer2000 − or you can use a vendor−neutral named sequence table. In all cases auto−generated primary keys are of type Integer or Long.

The topics in this section are:

- Primary Key Generation Using Oracle's Sequence
- Primary Key Generation Using SQL Server's IDENTITY
- Primary Key Generation Using a Named Sequence Table
- Defining the CMP Entity Bean

Primary Key Generation Using Oracle's Sequence

Oracle provides the 'sequence' utility to automatically generate unique primary keys. To use this utility to auto−generate primary keys for a CMP entity bean, you must create a sequence table and use the ejbgen:automatic−key−generation tag to point to this table.

In your Oracle database, you must create a sequence table that will create the primary keys, like is shown in the following example:

```
create sequence myOracleSequence
start with 1
nomaxvalue;
```

This creates a sequences of primary key, starting with 1, followed by 2, 3, and so forth. The sequence table in the example uses the default increment 1, but you can change this by specifying the increment keyword, such as increment by 3. When you do the latter, you must specify the exact same value in the cache−size attribute of the ejbgen:automatic−key−generation tag:

```
@ejbgen:automatic−key−generation type="Oracle" name="myOracleSequence" cache−size="3"
```

If you have specified automatic table creation in the CMP bean's project settings, the sequence table will be created automatically when the entity bean is deployed. For more information, see @ejbgen:jar−settings Annotation. For more information on the definition of a CMP entity bean, see below.

Primary Key Generation Using SQL Server's IDENTITY

In SQL Server (2000) you can use the 'IDENTITY' keyword to indicate that a primary−key needs to be auto−generated. The following example shows a common scenario where the first primary key value is 1, and the increment is 1:

```
CREATE TABLE Customer (Customer_ID int IDENTITY(1,1), FirstName varchar(30) LastName varchar(3
```

In the CMP entity bean definition you need to specify SQLServer(2000) as the type of automatic key generator you are using. You can also provide a cache size:

```
@ejbgen:automatic−key−generation type="SQLServer"
```

If you have specified automatic table creation in the CMP bean's project settings, the sequence table will be created automatically when the entity bean is deployed. For more information, see @ejbgen:jar−settings Annotation. For more information on the definition of a CMP entity bean, see below.

Note. The SQLServer2000 option is the same as SQLServer, except that SQLServer uses @@IDENTITY column to get the generated key value and SQLServer2000 uses the SCOPE_IDENTITY() function instead.

Primary Key Generation Using a Named Sequence Table

A named sequence table is similar to the Oracle sequence functionality in that a dedicated table is used to generate primary keys. However, the named sequence table approach is vendor−neutral. To auto−generate primary keys this way, create a named sequence table using the two SQL statements shown in the example:

```
CREATE Table MyNamedSequence (SEQUENCE number);

INSERT into MyNamedSequence VALUES (0);
```

In the CMP entity bean definition you need to specify the named sequence table as the type of automatic key generator you are using. You can also provide a cache size:

```
@ejbgen:automatic−key−generation name="MySequence" type="MyNamedSequenceTable" cache−size="100"
```

If you have specified automatic table creation in the CMP bean's project settings, the sequence table will be created automatically when the entity bean is deployed. For more information, see @ejbgen:jar−settings Annotation. For more information on the definition of a CMP entity bean, see the next section.

Note. When you specify a cache−size for a named sequence table, a series of unique values are reserved for entity bean creation. When a new cache is necessary, a second series of unique values is reserved, under the assumption that the first series of unique values was entirely used. This guarantees that primary key values are always unique, although it leaves open the possibility that primary key values are not necessarily sequential. For instance, when the first series of values is 10...20, the second series of values is 21−30, even if not all values in the first series were actually used to create entity beans.

Defining the CMP Entity Bean

When defining a CMP entity bean that uses one of the primary key generators, you point to the name of the primary key generator table to obtain primary keys, using the ejbgen:automatic−key−generation tag. Also, you must define a primary key field of type Integer or Long, to set and get the auto−generated primary key. However, the ejbCreate method does not take a primary key value as an argument. Instead the EJB container adds the correct primary key to the entity bean record.

The following example shows what the entity bean might look like. Notice that the bean uses the named sequence option describe above, and that ejbCreate method does not take a primary key:

```
 /**
 * @ejbgen:automatic−key−generation name="MyNamedSequence" type="NamedSequenceTable" cache−size
 *
 * @ejbgen:entity prim−key−class="java.lang.Integer"
 *   ejb−name = "Customer"
 *   data−source−name = "MyCustomerDataSource"
 *   table-name = "customer"
 *   abstract-schema-name = "Customer"
 *
 * ...
 */
abstract public class Customer extends GenericEntityBean implements EntityBean
{
    ...

    /**
     * @ejbgen:cmp-field primkey-field="true" column="Customer_ID"
     * @ejbgen:local-method
     */
    public abstract Integer getCustomer_ID();

    /**
     * @ejbgen:local-method
     */
    public abstract void setCustomer_ID(Integer arg);

    public java.lang.Integer ejbCreate(java.lang.String LastName, java.lang.String FirstName)
    {
      setLastName(LastName);
      setFirstName(FirstName);

      return null; // FIXME return PK value
    }
}
```

Related Topics

@ejbgen:automatic−key−generation Annotation

Automatic Primary Key Generation Sample

@ejbgen:jar−settings Annotation

Entity Relationships

Entity relationships are used to model real−world dependencies between business concepts with CMP entity beans. This topics gives an overview of the seven relationship types, and for each of these relationships describes implementation details in the EJBs and the underlying database tables. For more detailed information, see your favorite J2EE documentation.

The topics in this section are:

- One−to−One, Unidirectional
- One−to−One, Bidirectional
- One−to−Many, Unidirectional
- One−to−Many, Bidirectional
- Many−to−One, Unidirectional
- Many−to−Many, Unidirectional
- Many−to−Many, Bidirectional

One−to−One, Unidirectional

In a one−to−one unidirectional relationship, object A relates to object B. In addition, given object A you can find a reference to object B, but not the other way around. An example of such a relationship is between a concertgoer and a ticket, assuming the perspective of the ticket counter. Each concertgoer requires exactly one ticket, and the concertgoer will have a reference to his/her ticket. However, given a ticket you don't know the concertgoer. That is, if a lost ticket is returned to the ticket counter, it is not possible to trace it back to the concertgoer.

The Concertgoer EJB will have a CMR field to set and get a reference to a Ticket object. In contrast, there is no reference from the Ticket to the Concertgoer, meaning that there is no direct way to find out if a particular ticket has been assigned to a concertgoer or not and if it has, who the concertgoer is. (To find this out, you would have to run a query on the Concertgoer EJBs and check each referenced ticket .)

In this particular example, Ticket objects are probably created independently of Concertgoer objects, and the CMR set method is used to associate the Concertgoer with the Ticket. Also, when the concertgoer returns the ticket (and removes himself from the ticket counter database), the Ticket object may not be deleted because it can be resold. When the one−to−one unidirectional relationship is more dependent, as between a Customer and his/her Address, the Customer EJB may have an setAddress business method, which creates a new Address object first, and then uses the CMR set method to set the reference, as is shown in the following code snippet:

```
public void setAddress(String street, String apt, String city, String state, String zip) thr
{
    Address currentAddress = this.getAddress( );
    if (currentAddress == null) {
        // Customer's current address not known.
        newAddress = addressHome.create(street, apt, city, state, zip);
        setAddress(newAddress);
    }
    else {
        // Update customer's current address.
        currentAddress.setStreet(street);
```

```
            currentAddress.setApt(apt);
            currentAddress.setCity(city);
            currentAddress.setState(state);
            currentAddress.setZip(zip);
        }
    }
```

Notice that first the CMR get method is used to get the reference to the current address. if the address is not known, a new address object is created after which the reference is set. If there is already a reference to an address, the address object is updated to reflect the new address.

When a customer is removed from the database, the home address can likely be removed as well. To do so, you can specify a cascade delete for this entity relationship, which automatically removes the home address when the customer is removed. For more information, see the @ejbgen:relation Annotation.

With respect to persistent storage of this relationship, one table will have the foreign key column information, that is hold the a copy of the primary key of the other EJB. Typically the Concertgoer table will have an 'Ticket_Index' foreign−key column holding the primary key value of the address (assuming that the Ticket EJB defines only one primary key field; if there are multiple primary key columns, there are multiple foreign key columns holding these values). It is, however, also possible that the Ticket table holds the primary key value of the Concertgoer. Regardless of the implementation in the database table, the EJB container will ensure that the relationship is correctly represented.

One−to−One, Bidirectional

In a one−to−one bidirectional relationship, object A relates to object B and both reference each other. An example of such a relationship is between a concertgoer and a creditcard, again assuming the perspective of the ticket counter. Each concertgoer has only one credit card (to purchase a ticket), and if a credit card is inadvertently left behind at the ticket counter, it can be returned to the owner.

The Concertgoer EJB will have a CMR field to set and get a reference to a CreditCard object. In addition, the CreditCard object will have a CMR field to set and get a reference to a Concertgoer object. Also, the bidirectionality of this relationship is ensured by the EJB container. That is, when you use the Concertgoer's CMR set method to set a reference to a CreditCard object, the CMR field of the CreditCard object is automatically updated to hold a reference to this Concertgoer. Similarly, when you change or remove a reference in one object, this change is automatically applied to the other object. As far as creating and deleting the object a CMR field references, similar design considerations apply as discussed above for one−to−one unidirectional relationships. With respect to creating a new creditcard for a concertgoer, it is possible that the Concertgoer EJB has a business method setCreditCard, which creates a new creditcard record and then sets the reference, similar to the setAddress method shown above. If on the other hand the CreditCard EJB defines a create method that creates the CreditCard object and sets the reference to the Concertgoer object, the reference must be set in the ejbPostCreate step. An example of this is shown below.

With respect to persistent storage of this relationship, there is again quite some flexibility how this is implemented in the database tables. One of the possibilities is that only the ConcertGoer table has a foreign key column holding the primary key value of a creditcard. There are other possibilities as well. Regardless of the implementation in the database table, the EJB container will ensure that the relationship is correctly represented.

One−to−Many, Unidirectional

In a one−to−many unidirectional relationship, object A relates to many objects B, there are references from object A to all objects B, but not the other way around. An example of such a relationship is between a concertgoer and CDs purchased after the concert. A concertgoer can purchase many CDs, but for a purchased CD, again inadvertently lost and found, it is not possible to trace it back to the concertgoer who made the purchase.

The Concertgoer EJB will have a CMR field to set and get a Collection/Set of references to CD objects. In contrast, there is no CMR field in the CD object. The choice of java.util.Collection versus java.util.Set depends on whether from a design perspective it makes sense to have a collection with potentially duplicate references to the same object, or to have a set without duplicate references.

In this particular example, Concertgoer objects are probably created independently of CD objects, and a new reference is added to the CMR field uses the Collection/Set's add method, while a reference is deleted without deleting the CD object using the Collection/Set's remove method. When the relationship is more dependent, as between a Band and its Recordings, the Band EJB may have an addRecording business method, which creates a new Recording object first, and then adds it to the collection, as is shown in the following code snippet:

```
public void addRecording(String recording) throws CreateException
{
   Recording album = recordingHome.create(getBandName(), recording);
   Collection recordings = getRecordings();
   if(album != null) {
      recordings.add(album);
   }
 }
```

Notice that this method first creates the album, then uses the CMR get method to obtain a collection of the current recordings of the band, and then adds the new recording to the collection. Without the last step the recording would be created, but would not be referenced by the band.

When the Band object is deleted from the database, it might or might not make sense to cascade delete its recordings, depending on the real−world scenario you are representing.

With respect to persistent storage of this relationship, the only possible implementation is for the CD table to have a foreign key column holding the primary value of the concertgoer. As you might notice, the model and the actual implementation are reversed; the EJB container again ensures that the relationship is correctly represented.

Note. WebLogic at present doesn't support the use of a join table to implement one−to−many relationships.

One−to−Many, Bidirectional

In a one−to−many bidirectional relationship, object A relates to many objects B, there are references from object A to all objects B, and each object B references object A. An example of such a relationship is between a concertgoer and a creditcard, assuming the perspective of the *concertgoer*. Each concertgoer can have many credit cards, and if a credit card is inadvertently lost, it can be returned to the owner. Notice that this is the second example of a relationship between a concertgoer and creditcard. Above a one−to−one bidirectional relationship was defined, assuming the perspective of the ticket counter instead of the concertgoer.

Note. Realize that one−to−many bidirectional relationships and many−to−one bidirectional relationships are conceptually identical and are therefore listed as one type of relationship.

The Concertgoer EJB will have a CMR field to set and get a Collection/Set of references to CreditCard objects. The CreditCard object will have a CMR field to set and get a reference to a ConcertGoer object. Again, the bidirectionality of this relationship is ensured by the EJB container. That is, when you set/add the reference to one of the EJBs, the reference is automatically updated for the other EJB. As far as creating and deleting the object a CMR field references, similar design considerations apply as discussed above.

With respect to persistent storage of this relationship, the only possible implementation is for the CreditCard table to have a foreign key column holding the primary value of the concertgoer.

Note. WebLogic at present doesn't support the use of a join table to implement one−to−many relationships.

Many−to−One, Unidirectional

In a many−to−one unidirectional relationship, many objects A relate to one object B, there is a reference from each object A to object B, but not the other way around. An example of such a relationship is between a concert and a venue, assuming the perspective of a concertgoer. There are many different concerts at the same venue but the concertgoer is probably less concerned to know the concerts given the venue. However, if the latter is a business requirement, you will model this as a one−to−many (venue−to−concerts) bidirectional relationship.

The Concert EJB will have a CMR field to set and get a reference to a Venue object. In contrast, the Venue object will not have a CMR field. When a new concert is scheduled, the venue likely needs to be known at this time. In other words, when you create a Concert object, the reference to the Venue object should be set as part of the create procedure, as is shown in the following example:

```
abstract public class ConcertBean extends GenericEntityBean implements EntityBean
{
   ...

   public Integer ejbCreate(String bandName, Venue theVenue) {
       setBandName(bandName);
       return null;
   }

   public void ejbPostCreate(String bandName, Venue theVenue) {
       setVenue(theVenue);
   }

   ...
```

Notice that ejbCreate sets the name of the performing band, while the reference to the Venue object is set in the corresponding ejbPostCreate method. References must by design always be set in the ejbPostCreate method, because the primary key(s) needed to set the reference may not be available yet until after creation of the object.

Cascade deletions will most likely not make sense for this relationship. That is, removing one concert should not lead to the destruction of the venue, as many other concerts are scheduled for this venue.

With respect to persistent storage of this relationship, the only possible implementation is for the Concert table to have a foreign key column holding the primary value of the venue.

Note. WebLogic at present doesn't support the use of a join table to implement one−to−many relationships.

Many−to−Many, Unidirectional

In a many−to−many unidirectional relationship, object A relates to many objects B, object B relates to many objects A, there are references from each object A to its objects B, but not the other way around. An example of such a relationship is between concertgoers and concerts. A concertgoer will attend many concerts, each concerts will attract many concertgoers, given a concertgoer you might want to know the concerts he/she attended, but given a concert you likely don't want to know the particular concertgoers that were attending it.

The Concertgoers EJB will have a CMR field to set and get a collection/set of references to Concert objects. In contrast, the Concert object will not have a CMR field. As far as manipulating the CMR field, and creating and deleting the objects the CMR field references, similar design considerations apply as discussed above. Cascade deletions typically do not make sense for many−to−many relationships.

With respect to persistent storage of this relationship, a join table is used. Each record in a join table has two foreign−key columns, one holding the primary key value of a concertgoer and the other holding the primary key value of the concert (again assuming that both EJBs are defined to have one unique primary key field).

Many−to−Many, Bidirectional

In a many−to−many bidirectional relationship, object A relates to many objects B, object B relates to many objects A, there are references from each object A to its objects B as well as references from each object B to its objects A. An example of such a relationship is between a passenger and a flight. A passenger can take multiple flights, a flight is typically booked by multiple passengers, given a passenger you want to know the flights, and for a flight you want to know exactly which passengers should be on the airplane.

The Passenger EJB will have a CMR field to set and get a collection/set of references to Flight objects. The Flight EJB will have a CMR field to set and get a collection/set of references to Passenger objects. As far as manipulating the CMR field, and creating and deleting the objects the CMR field references, similar design considerations apply as discussed above. In this particular example it is possible that a passenger books a flight for several passengers at the same time. You can use the Collection/Set's addAll method to add multiple references. Also notice that bidirectionality is again assured by the EJB container, and that cascade deletions typically do not make sense for many−to−many relationships.

With respect to persistent storage of this relationship, a join table is used. Each record in a join table has two foreign−key columns, one holding the primary key value of a concertgoer and the other holding the primary key value of the concert (again assuming that both EJBs are defined to have one unique primary key field).

Related Topics

How Do I: Add a Relation to an Entity Bean?

@ejbgen:relation Annotation

Query Methods and EJB QL

Find and select methods for CMP (2.0) entity beans are defined using EJB QL. This query language, similar to SQL used in relational databases, is used to select one or more entity EJBs or entity bean fields. The WebLogic platform fully supports EJB QL 2.0 and offers a number of additional methods that can be used in conjunction with EJB QL.

Note. The EJB QL is used for all query methods, with the exception of findByPrimaryKey, which is automatically generated by the EJB container.

The topics in this section are:

- Find Methods
- Select Methods
- Standard EJB QL Operators
- WebLogic QL

Find Methods

A finder method is invoked by other EJBs or client applications on a CMP entity bean's local or remote home interface, and returns local or remote references to one or more instances of this entity bean that match the query. A find method can return a reference to a single entity instance, such as findByPrimaryKey, or to multiple entity instances returned as a java.util.Collection. Finder methods must start with the prefix find.

The following list shows common uses of EJB QL queries with find methods:

- ***Select All***. The method Collection findAll(), defined in the home interface of the *CustomerBean*, returns all records in the database table:

  ```
  SELECT OBJECT(o) from CustomerBean as o
  ```

  Notice that an EJB QL query always uses the EJB's abstract schema name CustomerBean to reference it. This name is typically the same as the EJB's descriptive name. For more details on these names, see the @ejbgen:entity Annotation.
- ***Input Parameters***. The method Collection findByItemName(java.lang.String itemname), defined in the home interface of the *ItemsBean*, returns all references to Itemsbean instances matching the item name:

  ```
  SELECT OBJECT(o) from ItemsBean as o WHERE o.itemname = ?1
  ```

  Notice that the method argument itemname is matched to input parameter ?1.
- ***Literal Values***. The method Collection findUSManufacturers(), defined in the home interface of the *ManufacturerBean*, returns all references to ManufacturerBean instances who are US manufacturers:

  ```
  SELECT OBJECT(o) from ManufacturerBean as o WHERE o.usManufacturer = 1
  ```
- ***Relationship Queries***. A *ManufacturerBean* and an *ItemsBean* are defined to have an entity relation, such that each item has a manufacturer, and a manufacturer can produce multiple items. For each item, the ItemsBean's CMR field manufacturer stores a unique index to a manufacturer. The method Collection findAllManufacturers(), defined in the home interface of the ManufacturerBean, queries

the *ItemsBean* to return the different manufacturers for all the items. It is possible that multiple items are created by the same manufacturer, yielding multiple references to the same manufacturer in the returned results:

```
SELECT OBJECT(m) from ItemsBean as o, IN(o.manufacturer) AS m
```

Notice that the keyword IN is used to return object references via a CMR field.

- ***Unique Records***. A *ManufacturerBean* and an *ItemsBean* are defined to have an entity relation, such that each item has a manufacturer, and a manufacturer can produce multiple items. For each item, the ItemsBean's CMR field manufacturer stores a unique index to a manufacturer. The method Collection findDistinctManufacturer(), defined in the home interface of the ManufacturerBean, queries the ItemsBean to return the different manufacturers for all the items. It is possible that multiple items are created by the same manufacturer, yielding multiple references to the same manufacturer, but the keyword DISTINCT is used to not return duplicates:

  ```
  SELECT DISTINCT OBJECT(m) from ItemsBean as o, IN(o.manufacturer) AS m
  ```

Note. The Finder Methods Sample allows you to run these and other EJB QL queries against a prefilled database table and examine the returned results. For more detailed information on EJB QL queries, see your favorite J2EE documentation.

Select Methods

A select method is defined using EJB QL and it can either return (local or remote) references to entity beans or values of an individual CMP field. A select method is not defined in the EJB's interfaces. In other words, it is a private method that can only be used internally by a CMP entity bean class. When returning object references, a select method can return a reference to a single entity instance, or to multiple entity instances which are returned as a java.util.Collection or java.util.Set. Select methods must start with the prefix ejbSelect.

The following list shows common uses of EJB QL queries with select methods:

- ***Object References***. A select method can use the same queries as find methods to return object references. The method java.util.Collection ejbSelectAll(), defined in the *ItemsBean* class (but not its interfaces), returns all records in the database table:

  ```
  SELECT OBJECT(o) from ItemsBean as o
  ```
- ***CMP Fields***. The method java.util.Collection ejbSelectItemNames(), defined in the *ItemsBean* class, returns only the item names of all items:

  ```
  SELECT o.itemname from ItemsBean as o
  ```

  Just as with a query that returns object references, a query that returns CMP fields can have input parameters, literal values, relationship querying with the IN keyword, and the DISTINCT keyword to avoid duplicate records. The method java.util.Collection ejbSelectByItemName(java.lang.String itemname), defined in the *ItemsBean* class, returns only the item names of the items that match the argument itemname:

  ```
  SELECT o.itemname from ItemsBean as o WHERE o.itemname = ?1
  ```

Note. The Select Methods Sample allows you to run these and other EJB QL queries against a prefilled database table and examine the returned results.

Standard EJB QL Operators

EJB QL 2.0 defines a number of standard operators. Some of these, like IN, DISTINCT, and the use of '.' as a navigational operator (for instance, to access a EJB's CMP field) have been described above. Other operators include:

- The comparison operators <, >, <=, >=, =, and <>.
- The logical operators NOT, AND, and OR.
- The arithmetic operators +, − (unary); *, /, +, and −.
- The arithmetic functions ABS(number), returning the absolute value of a (int, double, or float) number, and SQRT(double) returning the square root.
- LIKE is used for pattern matching with String fields. Use '%' to match with any number of characters and '_' to match with exactly one character (use '\' if these characters actually occur in the pattern). For instance, the following query will select all items whose price ends in '.95':

  ```
  SELECT OBJECT(o) from ItemsBean as o WHERE o.price LIKE '%.95'
  ```
- The String functions CONCAT(String1, String2), LENGTH(String), LOCATE(StringToFind, ContainingString [, starting position]) − equivalent to java.lang.String.indexOf −, and SUBSTRING(String, startposition, endposition).
- BETWEEN specifies a range of values (inclusive). For instance, the following query returns all items between $20 and $40:

  ```
  SELECT OBJECT(o) from ItemsBean as o WHERE o.price BETWEEN 20.00 AND 40.00
  ```
- IS NULL is used to test for null fields. CMP fields that hold an object (such as a String) and CMR fields that hold a single object can be null, while CMP fields holding primitive values and CMR fields holding a collection of objects cannot. For instance, the following query returns all items whose manufacturer is not known (assuming the same entity relationship as mentioned above):

  ```
  SELECT OBJECT(o) from ItemsBean as o WHERE o.manufacturer IS NULL
  ```
- IS EMPTY is used to test for empty sets, in particular CMR fields that holds a collection of objects. For example, the following query returns all manufacturers without known items in the database:

  ```
  SELECT OBJECT(o) from ManufacturerBean as o WHERE o.items IS EMPTY
  ```
- MEMBER OF is used to evaluate whether an object is part of a collection−based relationship. For example, the following query returns the manufacturer of a given item:

  ```
  SELECT OBJECT(o) from ManufacturerBean as o, IN (o.items) AS allItems, ItemsBean oneItem
      WHERE oneItem.itemname = ?1 AND oneItem MEMBER OF allItems
  ```

For more detailed information on EJB QL queries and the operators defined in this language, see the Finder and Select Methods Samples, or your favorite J2EE documentation.

WebLogic QL

The WebLogic platform offers an EJB QL extension, called WebLogic QL to run advanced queries. The following advanced query types are available:

- Ordering Query Results

- Using Aggregate Functions
- Returning Multiple CMP Fields
- Using Subqueries
- Using Oracle Hints

Ordering Query Results

To order the results of a find or select query, you can used the WebLogic keyword ORDERBY. You can specify in ascending order (the default) or descending order, and sort on multiple fields.

The following query returns ItemBean instances, ordered by item name, that are produced by a US manufacturer:

```
SELECT OBJECT(i) from ManufacturerBean as o, IN(o.items) AS i WHERE o.usManufacturer = 1 ORDERB
```

The following query for a select method returns all item names, ordered by price in descending order and by the quantity in stock in ascending order:

```
SELECT o.itemname from ItemsBean as o ORDERBY o.price DESC, o.quantityavailable ASC
```

Note. Because the actual ordering is done by the underlying DBMS, the exact order is database dependent.

Aggregate Functions

The following WebLogic aggregate functions are supported:

- AVG() returns the average value of this field. Can be used in combination with DISTINCT, as in AVG(DISTINCT o.quantityavailable).
- COUNT() returns the number of occurrences of a field. Can be used in combination with DISTINCT.
- `GROUP BY` aggregates entire records on the basis of a field value. An example is shown below.
- MIN() returns the minimum value of a field.
- MAX()Returns the maximum value of a field.
- SUM() returns the sum of a field. Can be used in combination with DISTINCT.

The following query returns all ItemsBean instances that are more expensive than the average price of all the items:

```
SELECT OBJECT(o) from ItemsBean as o WHERE o.price > (SELECT AVG(s.price) FROM ItemsBean AS s)
```

Returning Multiple CMP Fields

WebLogic QL supports select methods that, unlike the standard EJB QL, return the results of multi−field queries in the form of a `java.sql.ResultSet`. It is not possible to return object references as a ResultSet.

The following query returns the name and the quantity in stocks of all the items in the database:

```
SELECT o.itemname, o.quantityavailable from ItemsBean as o
```

The following query, which combines this functionality with the aggregate and sorting capabilities, returns the sorted names of all manufacturers of the items, the number of items offered by a manufacturer, and the average price of these items. The *ManufacturerBean* and *ItemsBean* are defined to have an entity relation, such that each item has a manufacturer, and a manufacturer can produce multiple items. For each item, the ItemsBean's CMR field manufacturer stores a unique index to a manufacturer instance:

```
SELECT o.manufacturer.manufacturername, COUNT(o.itemnumber), AVG(o.price) from ItemsBean as o
   GROUP BY o.manufacturer.manufacturername
   ORDERBY o.manufacturer.manufacturername
```

Using Subqueries

WebLogic QL supports the use of subqueries. One example of such a query was already shown above:

```
SELECT OBJECT(o) from ItemsBean as o WHERE o.price > (SELECT AVG(s.price) FROM ItemsBean AS s)
```

This query returns all ItemsBean instances that are more expensive than the average price of all the items, the average price of the items first being computed in a subquery. Subqueries follow the same syntax rules as standard WebLogic QL queries. You must, however, make sure that unique identifiers are used in the subquery and outer query. For instance, in the above query identifier o is used for ItemsBean in the outer query, while identifier c is used for ItemsBean in the subquery.

Note. It is also possible to create nested subqueries. The nesting depth is limited by the underlying database's nesting capabilities.

Correlated and Uncorrelated SubQueries

The above query is an example of an *uncorrelated* subquery. That is, the subquery is evaluated independently of the containing query. In contrast, in *correlated* subqueries values drawn from the main query are used in the subquery. Correlated queries are usually more processing intensive because the subquery is re−evaluated for every possible value obtained from the main query.

The following correlated subquery returns the five cheapest ItemsBean instances:

```
SELECT OBJECT(mainItem) from ItemsBean as mainItem WHERE 5 >
   (SELECT COUNT(subItem.itemname) FROM ItemsBean AS subItem
           WHERE subItem.price < mainItem.price)
```

In this example the subquery evaluates for every mainItem, which is an instance of ItemsBean from the outer query, how many other items are less expensive, using the COUNT aggregate function. If there are less than 5 items cheaper than the items currently evaluated in the outer query, the outer query will include this instance in the returned collection.

Return Values

In the subquery examples shown above, the subquery returns a single value obtained through an aggregate function on a CMP field (COUNT and AVG are shown; other possible functions are MIN, MAX, and SUM as described above). The return value of a subquery can also be a single value for a CMP field, a set of values for a CMP field, a single object instance or multiple objects. The subquery cannot return values for multiple CMP fields.

When the subquery returns a set (of field values or objects), you must use one of these additional operators in the outer query to evaluate against the set:

- [NOT] IN evaluates whether the value in the outer query is (not) contained in the set returned by the subquery.
- [NOT] EXISTS evaluates whether the set returned by the subquery is (not) empty.
- ANY, in combination with one of the comparison operators (<, >, <=, >=, =, and <>), evaluates whether the value of the outer value is less than (greater than, and so forth) *any* of the values in the set returned by the subquery.
- ALL, in combination with one of the comparison operators (<, >, <=, >=, =, and <>), evaluates whether the value of the outer value is less than (greater than, and so forth) *all* of the values in the set returned by the subquery.

The following query uses a correlated subquery to select all items in the subquery that are more expensive than the current item in the outer query. The outer query only selects an object when the set returned by the subquery is empty, using NOT EXISTS. In other words, this query returns the most expensive item(s):

```
SELECT OBJECT(mainItem) from ItemsBean as mainItem WHERE NOT EXISTS
   (SELECT OBJECT(subItem) FROM ItemsBean as subItem  WHERE subItem.price > mainItem.price)
```

The following query uses a uncorrelated subquery that returns a set of CMP field values, and uses '> ALL' to select objects in the outer queries whose field value is larger than all the values in the set. Specifically this query uses a subquery to select the quantities in stock of all the items by a certain manufacturer, and the outer query returns all the object for which larger quantities are in stock:

```
SELECT OBJECT(mainItem) from ItemsBean as mainItem WHERE mainItem.quantityavailable > ALL
   (SELECT subItem.quantityavailable from ItemsBean as subItem WHERE subItem.manufacturer.manuf
```

Note. Queries using the ANY or ALL keyword do not properly run on PointBase.

Using Oracle Hints

To pass hints to an Oracle Query, you can use the WebLogic QL keyword SELECT_HINT and enter the quoted hint, as is shown in the following example:

```
SELECT OBJECT(o) from ItemsBean as o WHERE o.price > 20 SELECT_HINT '/*+ FULL(ItemsBean) */'
```

For more information on hints, see your favorite Oracle documentation.

Related Topics

How Do I: Add a Finder Method to an Entity Bean?

Finder Methods Sample

@ejbgen:finder Annotation

How Do I: Add a Select Method to an Entity Bean?

Select Methods Sample

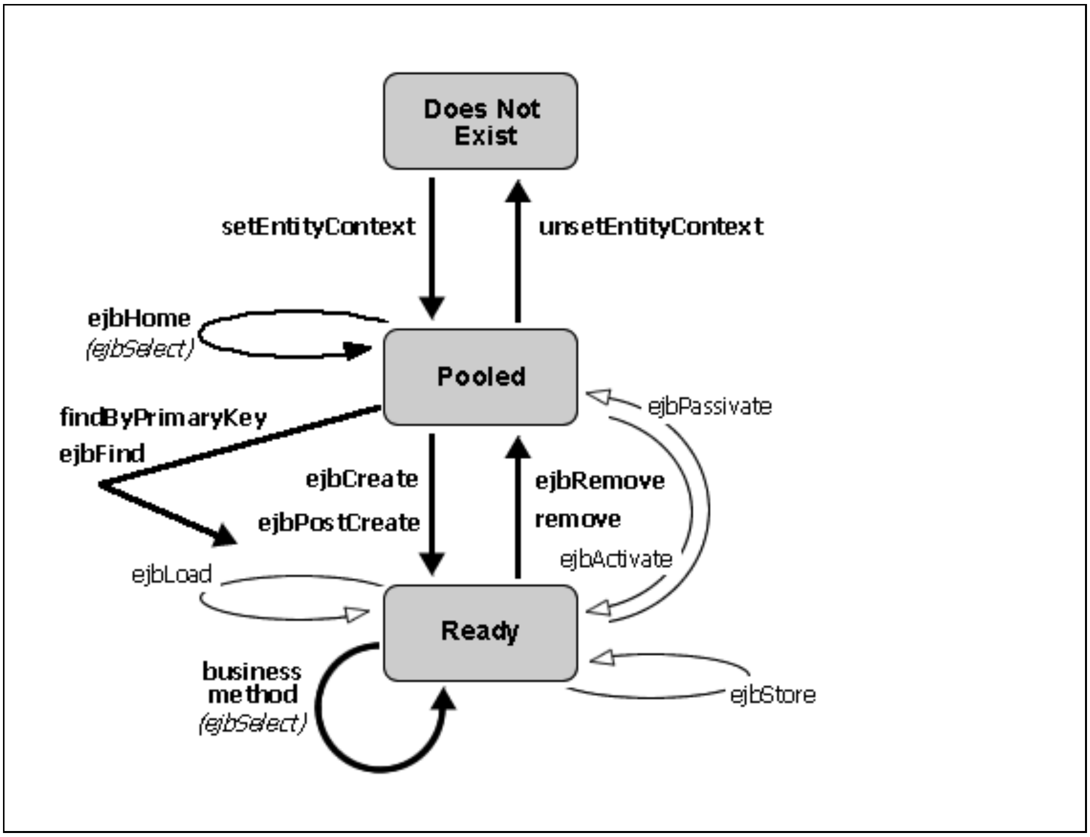
@ejbgen:select Annotation

The Life Cycle of an Entity Bean

The following figure shows the life cycle of an entity bean. An entity bean has the following three states:

- ***Does Not Exist***. In this state, the bean instance simply does not exist.
- ***Pooled***. When WebLogic server is first started, several bean instances are created and placed in the pool. A bean instance in the pooled state is not tied to particular data, that is, it does not correspond to a record in a database table. Additional bean instances can be added to the pool as needed, and a maximum number of instances can be set (for more information, see the @ejbgen:entity Annotation).
- ***Ready***. A bean instance in the ready state is tied to particular data, that is, it represents an instance of an actual business object.

The various state transitions as well as the methods available during the various states are discussed below.



Moving from the Does Not Exist to the Pooled State

When WebLogic server creates a bean instance in the pool, it calls the callback method public void setEntityContext(EntityContext ctx). This method has the parameter javax.ejb.EntityContext, which contains a reference to the entity context, that is, the interface to the EJB container. The entity context contains a number of methods to self−reference the entity bean object, identify the caller of a method, and so forth. Complete details about the javax.ejb.EntityContext can be found in your favorite J2EE documentation and the API reference at http://java.sun.com.

If you want to use the EntityContext reference in the entity bean, you must implement this callback method and store the reference. In addition, this method is also frequently used to look up the home interface of other beans later invoked in one of the bean's methods. The following code sample shows both:

```
/**
 * @ejbgen:ejb-local-ref link="Recording"
 *
 * ...
 */
abstract public class BandBean extends GenericEntityBean implements EntityBean
{
  private EntityContext ctx;
  private RecordingHome recordingHome;


  public void setEntityContext(EntityContext c) {
      // store the reference to the EntityContext
      ctx = c;
                // look up the home interface of the RecordingBean
      try {
         javax.naming.Context ic = new InitialContext();
         recordingHome = (RecordingHome)ic.lookup("java:/comp/env/ejb/Recording");
      }
      catch(Exception e) {
         System.out.println("Unable to obtain RecordingHome: " + e.getMessage());
      }
  }

  ...
```

The Pooled State

When a bean instance is in the pooled state, it is not tied to any particular business object. When in the pooled state, the methods defined in the home interface can be invoked, effectively transitioning it from the pooled to the ready state, with the exception of ejbHome methods. When a home method is invoked, a result that is not bean instance specific is returned to the caller, and the bean instance remains in the pooled state. Home methods in turn often invoke ejbSelect methods to query bean instances. For more information, see the Home Methods Sample.

Moving from the Pooled to the Ready State

The following methods move a bean instance from the pooled to the ready state to represent a business object:

- ejbCreate and ejbPostCreate. When the create method is invoked on the bean's home interface, the ejbCreate and ejbPostCreate methods are invoked. The bean instance moves to the ready state and represents this newly created business object. After creation, a (local or remote) reference to this object is returned to the caller, enabling the caller to invoke business methods on this instance. The ejbPostCreate method is used to set references to other entity beans as part of the creation of a new bean instance. For more information, see Entity Relationships.
- findByPrimaryKey. When this method is invoked on the bean's home interface with the (compound) primary key as the parameter, the bean instance moves to the ready state and represents the business object uniquely identified by the parameter. Also a (local or remote) reference to this object is returned to the caller, enabling the caller to invoke business methods on this instance.
- Finder methods. When a Finder method is invoked on the bean's home interface, one or a set of references to objects matching the queries are returned to the caller. In many cases the corresponding bean instance(s) are not loaded with data and move to the ready state until at a later point when a

business method is actually invoked on this object, a concept known as lazy loading. However, you can specify eager loading, that is, tell the EJB container to load (part of) the data and move the entity bean instance(s) to the ready state as a part of the finder method execution. For more information, see Accelerating Entity Bean Data Access.

The Ready State

When a bean instance is in the ready state, it represent data for a business object. At this point any business method, that is any component method and accessor method, can be invoked on this object. (A component method may in turn call an ejbSelect method.) After a business method executes, the bean returns to the ready state to allow another business method invocation.

From the perspective of the EJB container, the execution of a component method is sandwiched between two synchronization steps:

1. Before a business method is executed, the EJB container updates the fields of the bean instance with the latest data from the database table to ensure that the bean instance has the latest data. Just after the data is updated, the EJB container invokes the callback method ejbLoad. If your entity bean needs to execute some custom logic as part of this synchronization step, you can use implement it using this callback method.
2. The business method executes and completes.
3. The EJB container now updates the database table to ensure that it contains the latest data from the entity bean instance. In other words, if the business method changes data values, this synchronization step ensures these changes are stored. Just prior to updating the database table, the EJB container invokes the callback method ejbStore. If your entity bean needs to execute some custom logic as part of this synchronization step, you can implement it using this callback method.

Because a record in a database table can be accessed by multiple bean instances at the same time, these synchronization steps ensure that each bean instance always has the latest data. However, in some cases these synchronization steps might be overkill and unnecessarily slow down performance. For instance, an entity bean might be read−only, reading data that is changed rarely if at all. In these cases one can safely bypass the synchronization steps without risking violations to data integrity. For more information, see Accelerating Entity Bean Data Access.

Moving from the Ready to the Pooled State

When a caller invokes a remove method to delete an entity bean instance and its underlying record in the database table, the EJB container will delete the bean instance. Just prior to deleting the instance, it will call the callback method ejbRemove. If your entity bean needs to execute some custom logic prior to deletion, you can implement it using this callback method. After the data is deleted, the bean instance returns to the pooled state. The bean instance is no longer tied to any particular business object, and can be used to execute a home method or one of the methods that will tie it to a new set of data and move it to the ready state.

For more information on how to delete an entity bean instance, see Defining an Entity Bean.

Activation and Passivation

To more optimally manage resources, the EJB container might passivate a bean instance by moving it from the ready state to the pooled state. During passivation the entity bean instance is dissociated from the business object it represents, and become available to represent another set of data. Conversely, a passivated bean might be activated, meaning that it moves from the pooled to ready state to represent a business object.

It should be noted that the caller (a client application or another EJB) of the entity bean instance will be unaware of passivation having taken place. The caller's reference to the entity bean instance is still maintained and valid; that is, if the caller subsequently invokes a business method on this entity bean instance, an instance from the pooled state will be moved to the ready state to represent this business object.

A bean instance can be passivated when none of its business method are invoked. Passivation occurs after synchronization has completed, guaranteeing that the database has stored any changes to the business object. Just prior to actual passivation, the callback method ejbPassivate is invoked. If your entity bean needs to execute some custom logic prior to passivation, you can implement it using this callback method.

When a previously passivated bean instance is activated to service business method invocation, the callback method ejbActivate is invoked. If your entity bean needs to execute some custom logic prior to activation, you can implement it using this callback method. For instance, you might use this callback method to reinitialize values of nonpersistent fields, that is, fields not stored in the database. After the callback method executes and completes, the s*ynchronization − business method invocation − synchronization* procedure described above follows as during any other business method invocation; that is, first synchronization happens during which the latest bean instance is updated with the latest data of the database, followed by the invocation of the ejbLoad callback method. After this completes the business method is invoked, and when this completes, the second synchronization happens during which the ejbStore callback method is invoked and the latest bean instance data is stored to the database.

Moving from the Pooled to the Does Not Exist State

To more optimally manage resources, or when WebLogic server shuts down, the EJB container might remove a bean instance from the pooled state to the does not exist state, allowing it to be garbage collected. Just prior to its destruction, the callback method unsetEntityContext is invoked. If your entity bean needs to execute some cleanup prior to garbage collection, you can implement it using this callback method.

Related Topics

Accelerating Entity Bean Data Access

Entity Relationships

Accelerating Entity Bean Data Access

WebLogic platform provides a number of features to reduce data access latency for CMP entity beans. This topic discusses the following techniques:

- Using Eager Loading in Query Methods
- Using Eager Loading in Relationships
- Query Methods and Modified Data
- Using Read−Only Entity Beans
- Using Optimistic Concurrency
- Using Caching Between Transactions
- Application−Level Caching
- Passing Method Arguments By Reference
- Sorting and Batching Database Operations

Note. This topic is meant to specifically point out a number of Weblogic−specific features that speed up data access. Design patterns, the use of local interfaces, and other J2EE design and architecture issues that affect performance are not addressed here. For more information on those topics, please consult your favorite J2EE documentation.

Using Eager Loading in Query Methods

A field−group represents a subset of container−managed persistence (cmp) and container−managed relation (cmr) fields of a single entity bean. You can associate a group with a query method (or relationship; see below), so that when a bean is loaded as the result of executing a query, only the fields mentioned in the group are loaded from the database when the query method returns, a concept known as *eager loading*. An entity bean's field that is not part of that group is not loaded at that point, but will be loaded if you subsequently access this field (through a get method), a concept known as *lazy loading*. Data access will be improved if you take advantage of eager loading while avoiding lazy loading, that is, by making sure that all and only the fields that you need are returned by the query method.

Note. Eager loading is way of specifying what data are loaded and cached for the duration of a transaction. If a finder method is used in one transaction, and in a subsequent transaction the bean returned by that finder method is accessed (using a get method), the accessed field is loaded anew (as the data might have changed in the meantime). If caching between transactions is enabled, then eager loading will be used the first time to cache the data, but subsequently the cached data is used instead. For more information about caching, see the various caching sections below.

To enable eager loading of field groups, you must do the following:

- Verify that ejbgen:entity finders−load−bean is set to true (or was left unspecified). When set to false, query methods will always just load the primary key values of the beans returned by the query.
- Define your cmp and cmr fields, and group related fields together using the group−names attribute. You can make a field a member of multiple groups if necessary.
- Define your select and/or find method(s), and use the group−name attribute to specifies which field group should be loaded eagerly.

In the following code snippet two field groups are defined, as well as a findAll method that uses eager loading of one of these groups. Particularly relevant lines of code are shown in bold:

```
/**
 * @ejbgen:entity prim-key-class="java.lang.Integer"  ...
 *
 * ...
 *
 * @ejbgen:finder group-name="queryGroup"
 * ejb-ql="SELECT OBJECT(o) from CachedOne as o"
 * generate-on="Local"
 * signature="Collection findAll()"
 */
abstract public class CachedOne extends GenericEntityBean implements EntityBean
{

    /**
     * @ejbgen:cmp-field group-names="queryGroup" primkey-field="true" column="Index"
     * @ejbgen:local-method
     */
    public abstract Integer getIndex();

    /**
     * @ejbgen:local-method
     */
    public abstract void setIndex(Integer arg);

    /**
     * @ejbgen:cmp-field group-names="queryGroup" column="Name"
     * @ejbgen:local-method
     */
    public abstract String getFieldOne();

    /**
     * @ejbgen:local-method
     */
    public abstract void setFieldOne(String arg);

    /**
     * @ejbgen:cmp-field group-names="nonQueryGroup" column="FieldTwo"
     * @ejbgen:local-method
     */
    public abstract String getFieldTwo();

    /**
     * @ejbgen:local-method
     */
    public abstract void setFieldTwo(String arg);

    ...
```

Using Eager Loading in Relationships

If a query method returns (one or) a set of entity bean object(s), the cmp and cmr fields of each object are loaded at this point (subject perhaps to eager loading of field groups, as discussed above). If subsequently a cmr field (relationship field) accessor method is used to, for instance, get a set of related entity beans referenced via the field, those beans are loaded at this point, requiring a separate database read. To reduce the number of database queries and enhance performance, you can enable eager loading of relationships, such that in the results returned by a query method, related entity beans have also been loaded.

Eager loading of relationships is possible for all entity relationships, with the exception of many−to−many relationships. For more information about relationships, see Entity Relationships. It is also possible to nest eager loading of relationships, that is, for the beans that are loaded as part of a relationship, you can in turn load their entity relationships (see below).

Note. Eager loading is way of specifying what data are loaded and cached for the duration of a transaction. If a finder method is used in one transaction, and in a subsequent transaction the bean returned by that finder method is accessed (using a get method), the accessed field is loaded anew (as the data might have changed in the meantime). If caching between transactions is enabled, then eager loading will be used the first time to cache the data, but subsequently the cached data is used instead. For more information about caching, see the various caching sections below.

To enable eager loading of relationships, you must do the following:

- Verify that ejbgen:entity finders−load−bean is set to true (or was left unspecified). When set to false, query methods will always just load the primary key values of the beans returned by the query.
- In the ***Property Editor***, specify the database type to indicate which DBMS is used. Eager loading of relationships uses database−dependent query (outer join) syntax. For more information, see the database−type attribute of @ejbgen:entity Annotation.
- Specify the entity relationship(s) you want to load eagerly. To do so, insert an ejbgen:relationship−caching−element tag and specify the cmr fields that store a reference to the related bean(s). Use the caching−name attribute to name this specification. For more information, see @ejbgen:relationship−caching−element Annotation. If an entity bean has several entity relationships, and different query methods should eagerly load different relationships, you can create multiple eager relationship loading definitions, that is, insert multiple ejbgen:relationship−caching−element tags.
- Define your select and/or find method(s), and use the caching−name attribute to specifies which relationship(s) should be eagerly loaded.

In the following example a findByName method takes a String argument and returns the band(s) matching that name. For the returned band(s), its recordings, represented by RecordingBean instances and referenced through the cmr field recordings, are eagerly loaded at the same time. (The BandBean and RecordingBean are engaged in a one−to−many bidirectional entity relationship.) The code snippet shows the definition of the finder method, the entity relationship, and the eager relationship loading specification:

```
 * @ejbgen:finder caching−name="LoadRecordings"
 *     ejb−ql="SELECT DISTINCT OBJECT(a) FROM BandEJB AS a WHERE a.name = ?1"
 *     signature="java.util.Collection findByName(java.lang.String n0)"
 * @ejbgen:relation role−name="BandEJB−has−Recordings" cmr−field="recordings"
 *     target−ejb="Recording"
 *     multiplicity="One" name="Recording−BandEJB"
 * @ejbgen:relationship−caching−element caching−name="LoadRecordings" cmr−field="recordings"
 */
abstract public class BandBean extends GenericEntityBean implements EntityBean
{
   ...
```

Nested Eager Loading of Relationships

For entity bean instances that are loaded as the result of eager loading of relationships, it is possible to in turn load their related beans. You can nest eager loading of relationships by using the id attribute on the top−level ejbgen:relationship−caching−element tag, and using the parent−id attribute on the nested tag. There is no limit to the number of levels you can nest, although in practice using multiple levels is likely going to negatively affect performance, due to the large number of beans that are being loaded.

Building on the example used above, recording beans also have a many−to−one bidirectional relationship with producer beans. That is, for each recording there is one producer (and for each producer there are many recordings). The following sample shows how to use the id and parent−id attributes to also load the producer for each recording that is returned by the BandBean's findByName query method. The BandBean definition is shown first:

```
 * @ejbgen:finder caching−name="LoadRecordings"
 *     ejb−ql="SELECT DISTINCT OBJECT(a) FROM BandEJB AS a WHERE a.name = ?1"
 *     signature="java.util.Collection findByName(java.lang.String n0)"
 * @ejbgen:relation role−name="BandEJB−has−Recordings" cmr−field="recordings"
 *     target−ejb="Recording"
 *     multiplicity="One" name="Recording−BandEJB"
 * @ejbgen:relationship−caching−element id="LR_ID" caching−name="LoadRecordings"
 *     cmr−field="recordings"
 */
abstract public class BandBean extends GenericEntityBean implements EntityBean
{
   ...
```

The RecordingBean defines the entity relationship with the ProducerBean, and an eager loading definition that is nested inside the BandBean's LoadRecordings eager loading definition, using the parent−id attribute:

```
 * @ejbgen:relation role−name="Recordings−have−Producer" fk−column="newColumn "
 *    cmr−field="producer" target−ejb="Producer"
 *    multiplicity="Many" name="Recording−Producer"
 * @ejbgen:relationship−caching−element caching−name="LoadProducer"
 *    cmr−field="producer" parent−id="LR_ID"
 */
abstract public class RecordingBean extends GenericEntityBean implements EntityBean
{
   ...
```

Query Methods and Modified Data

In addition to eager loading of group−names and relationships, there is a third way to increase performance of query methods. In line with the EJB 2.0 specification, updates made by a transaction are by default reflected in the results of query methods that are subsequently invoked during the same transaction. This behavior is set by the include−updates attribute on the query method, that is, its ejbgen:finder or ejbgen:select tag.

If the include−updates attribute is set to true (or left unspecified), the EJB container saves all changes made at this point in the transaction to the database before executing the query. This guarantees that the changes show up in the query results but it also slows down performance. When this attribute is set to false, the updates of the current transaction are not reflected in the query. This results in better performance, although it does not necessarily guarantee the most current data to be returned by a query. However, you can safely set this attribute to false if within a transaction you do not re−query the data you just modified in that transaction, which is a common scenario.

Note. If the include−updates attribute is set to true (or left unspecified), changes are always stored before executing a query, thereby overriding the setting to wait until the end of a transaction before data is stored. However, if the transaction subsequently fails, any changes made during that transaction are still rolled back. For more information, see EJBs and Transactions.

Using Read−Only Entity Beans

When you need to frequently read data from a table but you do not, or only very occasionally, have to write to this table, you can improve performance by creating a read−only bean. This read−only entity bean is used for all read operations, while the occasional write operations to the table is performed by another EJB. Make sure that the following attributes are set in the Property Editor for a read−only bean:

- The attribute concurrency−strategy is ReadOnly.
- The attribute cache−between−transactions is set to true.
- The attribute read−timeout−seconds should be set to an acceptable amount, depending on how often the cached data is updated.
- Make sure the related caching attributes idle−timeout−seconds and max−beans−in−cache are set to appropriate values.

All these attributes are part of the @ejbgen:entity Annotation. To learn more about caching between transactions, read the section below.

Invalidating Read−Only Entity Beans

When the data that a read−only entity bean accesses is changed by the companion EJB with write operations, you might want to invalidate the read−only entity bean, thus forcing a refresh of the cache and ensuring that the read−only bean always returns the most current data.

To invalidate a read−only bean, you must set the 'write' entity bean's attribute invalidation−target, located in the Property Editor under ***Deployment***, to the ejb−name of the read−only entity bean. This attribute is part of the @ejbgen:entity Annotation.

Using Optimistic Concurrency

The *Optimistic* concurrency strategy for an entity bean entails that, unlike the *Database* or *Exclusive* strategy, no locks in the EJB container or database are placed during a transaction (also see the Bean Caching and Concurrency Options Summarized section below. To learn more about transactions, see EJBs and Transactions). The EJB container verifies that none of the data used by the transaction has been changed (by another source during another transaction) before committing the transaction. If the verified data has changed, the EJB container rolls back the transaction. The EJB container can verify that the used data has not changed in various ways, as specified by the verify−columns and verify−rows attributes on the ejbgen:entity tag. The options for the verify−columns attribute are as follows:

- ***Read***. The EJB container checks all the columns in the table that have been read during the transaction. This most stringent option typically increases the amount of checking the EJB container must perform and lowers performance. However, it ensures that the data used in this transaction has not been changed in the mean−time.
- ***Modified***. The EJB container checks only the columns that have been updated by the current transaction. This option typically increases performance, but it allows that columns that were not changed in the current transaction were changed in the meantime by another transaction. This interleaving of updates may or may not be desirable in all circumstances
- ***Version***. The EJB container checks a version number on the record to verify that the record has not been changed. For more information, see below.
- ***TimeStamp***. The EJB container checks a timestamp on the record to verify that the record has not been changed. For more information, see below.

Note. The EJB container does not check Oracle Blob/Clob fields for optimistic concurrency when using the Read or Modified option. Use version number or timestamp checking instead.

The options for the verify−rows attribute are as follows:

- ***Read***. The EJB container checks any row that is read by the transaction. This most stringent option typically increases the amount of optimistic checking the EJB container must perform and lowers performance. However, it ensures that none of the data used in this transaction has been changed in the meantime.

  Note. If verify−rows is set to Read, verify−columns cannot be set to Modified.
- ***Modified***. The EJB container checks only the rows that have been updated by the current transaction. This option typically increases performance, but it allows that rows that were not changed in the current transaction were changed in the meantime by another transaction. This interleaving of updates may or may not be desirable in all circumstances

Apart from the one noted exception, verify−rows and verify−columns work in conjunction. For instance, setting verify−columns to Read and verify−rows to Modified ensures that all columns of only modified rows are checked; Setting verify−columns to Version and verify−rows to Read means that all rows are checked via the version number; and so forth.

Note. If the EJB is mapped to multiple tables, optimistic checking is only performed on the tables that are updated during the transaction. In a clustered server environment, notifications for updates of optimistic data are broadcast to other cluster members to help avoid optimistic conflicts.

Using a Version Number

When you specify the Version option on verify−columns, the EJB container checks via a version number that a record has not changed in the meantime by another transaction. The EJB container manages this version number, that is, it updates the version number of a record when its data has changed. However, you must define a version column in the database table to hold that version number. The column can be of type Integer or Long. In addition, you must indicate that this column is to be used for the version number by the EJB container, by specifying the column name in the optimistic−column attribute of the ejbgen:entity tag. Mapping this column to a cmp field in the entity bean is optional.

Using a Timestamp

When you specify the Timestamp option on verify−columns, the EJB container checks via a time stamp that a record has not changed in the meantime by another transaction. The EJB container manages this time stamp, that is, it updates the time stamp of a record when its data has changed. However, you must define a column in the database table to hold that timestamp, which is an instance of java.sql.Timestamp. (The exact type of this column is database−dependent). In addition, you must indicate that this column is to be used for the timestamp by the EJB container, by specifying the column name in the optimistic−column attribute of the ejbgen:entity tag. Mapping this column to a cmp field in the entity bean is optional.

Note. When the EJB container updates a timestamp, it enforces at least a one−second difference between the old and new timestamp value.

Using Caching Between Transactions

The EJB 2.0 architecture specifies that for entity beans, the CMP entity bean's are synchronized with the database before business methods are invoked. In WebLogic, this synchronization is specifically performed at the start of each transaction (directly after which the ejbLoad callback is invoked to notify the beans of the completed synchronization). When multiple sources may update the same data, synchronizing at this point ensures that the bean has the most current version of the data.

When only the EJB instance is guaranteed to ever access the data it represents, synchronizing repeatedly is unnecessary. Because no other clients or systems update the EJB's underlying data, the cached version of the EJB data is always up to date. To avoid these unnecessary synchronizations, you can set the ejbgen:entity cache−between−transactions attribute to true. When you set this attribute, synchronizations are only performed when:

- The entity bean does not yet exist in the cache. Its data is read from the database to fill the cache.
- The entity bean's transaction rolls back. Its data is read from the database to undo any changes made to the data within the rolled back transaction.
- The time to use cached data, as set by the read−timeout−seconds attribute, has expired, forcing a read from the database.
- The entity bean was cached but has been removed from the cache since then because its idle time exceeded the idle−timeout−seconds attribute.

Note. If you incorrectly set cache−between−transactions set to true, that is, if the data that the EJB maps can be changed by multiple sources, data integrity violations may result.

To set the maximum number of beans in the cache, you can set the max−beans−in−cache attribute. These and the other caching attributes are part of the @ejbgen:entity Annotation.

Bean Caching and Concurrency Options Summarized

The concurrency strategy for an entity bean is set using the ejbgen:entity concurrency−strategy attribute, which can be done through the Property Editor. Similarly, in the Property Editor you can enable caching between transactions using the ejbgen:entity cache−between−transactions attribute, but enabling caching is not possible for all concurrency strategies. The following list gives an overview of the various valid combinations:

- ***Database***. This is the default concurrency strategy. When data is used in a transaction, the EJB container allocates a separate entity bean to represent that data. Any required locking actions are not handled by the container but are deferred to the database.

  Using this concurrency strategy, the EJB container loads the latest data from the database at the beginning of the transaction. When the data is to be committed (just prior to calling the ejbStore callback), this request to store data is passed along to the database. The database handles all required lock management and provides deadlock detection.

  Deferring locks to the underlying database enables concurrent access to entity EJB data. However, using database locking requires more detailed knowledge of the underlying datastore's lock policies, which might reduce portability among different database systems. Also, the container never caches the intermediate state of the EJB instance between transactions. That is, setting the ejbgen:entity cache−between−transactions attribute to True is not a valid option for this concurrency strategy.

- ***Exclusive***. The EJB container places an exclusive lock on the data represented by an entity bean instance when the bean is associated with a transaction. Any concurrent requests for the same data are blocked, even if it is a read−only request to this data, thus potentially greatly reducing performance.

  Using this concurrency strategy, you can either enable or disable caching between transactions. However, caching is not allowed for this concurrency strategy in a clustered server environment.
- ***ReadOnly***. Used only for read−only entity beans. When you specify this concurrency strategy, you must also enable caching between transactions. For more information, see the Read−Only Entity Beans section above.
- ***Optimistic***. When you specify this concurrency strategy, no locks in the EJB container or database are placed during a transaction. The EJB container verifies that none of the data used by the transaction has been changed (by another source) before committing the transaction. Using this concurrency strategy, you can either enable or disable caching between transactions. For more information, see the Using Optimistic Concurrency section above.

Note. Concurrency settings may affect transaction isolation levels. For more information, see EJBs and Transactions.

Application−Level Caching

Application−level caching allows multiple entity beans that are part of the same application to share a single runtime cache. Application−level caching offers the following advantages:

- It reduces the number of entity bean caches, and hence the effort to configure the cache.
- It ensures better utilization of memory and heap space, because of reduced fragmentation. For instance, if a particular EJB experiences a burst of activity, it can make use of all memory available to the combined cache, while other EJBs that use the cache are paged out. If two EJBs use different caches, when one bean's cache becomes full, the container cannot page out EJBs in the other bean's cache
- It simplifies management, as application−level caching enables a system administrator to tune a single cache, instead of many caches.

Application−level caching is not the best choice, however, for applications that experience high throughput. Because one thread of control exists per cache at a time, high throughput can create a bottleneck situation as tasks compete for control of the thread.

To Configure the Application−Level Cache

To create the application−level cache, you must manually edit weblogic−application.xml, located in the META−INF directory of your WebLogic application. Locate the ejb section of the file, and add the entity−cache within this section, as shown in the following example:

```
<weblogic−application>
   <ejb>
      <entity−cache>
         <entity−cache−name>large_account</entity−cache−name>
         <max−cache−size>
            <megabytes>1</megabytes>
         </max−cache−size>
      </entity−cache>
   </ejb>
</weblogic_application>
```

The most common xml elements for entity caches are:

- entity−cache. Defines the entity cache. You can define optionally more than one application level cache to cache entity beans.
- entity−cache−name. Specifies a unique name for an entity bean cache. The name must be unique within an ear file.
- max−beans−in−cache. This optional element specifies the maximum number of entity beans that are allowed in the application−level cache. If the limit is reached, beans may be passivated. The default is 1000. If 0 is specified, then there is no limit. This mechanism does not take into account the actual amount of memory that different entity beans require.
- max−cache−size. This optional element is used to specify a limit on the size of an entity cache in terms of memory size expressed either in terms of bytes or megabytes.

For more information, see Enterprise Application Deployment Descriptor Elements.

To Reference the Application−Level Cache

To use the application−level entity cache for an entity bean, you must add an @ejbgen:entity−cache−ref tag to its definition, and use the name attribute to refer to the application−level entity cache name. If you have specified the application−level cache maximum using the max−cache−size, instead of max−beans−in−cache, you should also use the tag's estimated−bean−size attribute to specify an estimated bean size. These estimates are primarily used to estimate the relative amounts of different entity bean instances that can be held in the application−level cache.

If you don't reference the application−level cache, a separate cache is used for the entity bean.

Passing Method Arguments By Reference

When you invoke entity beans through their remote interfaces, WebLogic must pass method arguments by value (due to classloading requirements). When EJBs are invoked through their local interface, WebLogic can pass method arguments by reference, improving the performance of method invocation (because parameters are not copied).

In WebLogic passing arguments by value is done by default. To enable passing arguments by reference, you must set the ejbgen:entity enable−call−by−reference to True.

Sorting and Batching Database Operations

When an entity bean is part of a transaction, the EJB container can take advantage of this by executing the SQL commands on the database table at the end of the transaction when the transaction commits instead of at various points during the transaction. (To learn more about transactions, see EJBs and Transactions). In addition, the EJB container can execute related changes in a single SQL instead of executing separate SQL commands for each individual change. The ejbgen:entity order−database−operations and ejbgen:entity enable−batch−operations attributes are used to enable this. By default, both attributes are set to True.

When database operations are ordered, insert, update and delete commands are sorted to ensure data consistency. These sorted database operations are executed at commit time instead of during the transaction. For instance, if during a transaction a customer receives a new credit card and the old credit card is destroyed, a reference is first made from the customer record to the new credit card record before the old credit card record is deleted. Reversing the order of the SQL commands would violate referential integrity.

If multiple instances of an entity bean are changed in the transaction, enabling batch operations increases performance by allowing the EJB container to update/insert/delete multiple records in the database at the same time instead of issuing separate SQL commands for each entity bean instance change.

Note. Batch operations are disabled if the transaction involves Automatic Primary Key Generation with SQL Server or an update to an OracleBlob or OracleClob CMP field (see the @ejbgen:cmp−field Annotation). The total number of instances inserted in a batch operation cannot exceed the entity bean's max−beans−in−cache attribute.

Related Topics

@ejbgen:cmp−field Annotation

@ejbgen:cmr−field Annotation

@ejbgen:finder Annotation

@ejbgen:select Annotation

@ejbgen:entity Annotation

Bean−Managed Persistence

When you use container−managed persistence (CMP) for entity beans, the EJB container handles the interaction with the underlying database. The EJB container ensures synchronization of data just prior and just after a business method executes (see The Life Cycle of an Entity Bean), it deletes the underlying database record when the entity bean instance is deleted, for an ejbCreate method it inserts the data in the database, it automatically generates the findByPrimaryKey method and handles the database query to find a database record, and it interprets EJB QL and WebLogic QL queries on finder and select methods to generate the corresponding database queries. In most cases entity bean development is done using CMP.

In contrast, when you use bean−managed persistence (BMP) for entity beans, the EJB container will still provide a numbers of services (such as transaction management) but leaves the persistence management up to the bean class. Bean−managed persistence might be necessary when your bean represents an unusual set of data and/or interacts with a legacy system for data storage. The current topic introduces some of the main difference between CMP and BMP, and some of the main differences developing CMP entity beans in WebLogic. For detailed information on CMP bean development, see your favorite J2EE documentation.

Main Differences Between CMP and BMP

Compared to CMP, when you develop BMP entity beans you must take into account the following:

- You are responsible for synchronizing the entity bean instance and the underlying database record before and after a business method invocation. To do so you must implement the callback methods ejbLoad and ejbStore. (The EJB container invokes the various callback methods for CMP and BMP entity beans alike; see The Life Cycle of an Entity Bean.)
- The ejbCreate method(s) must take care of storing the new data in the database.
- The findByPrimaryKey method must be explicitly defined and must take care of locating the data corresponding to the (compound) primary key in the database.
- Removing the data from the database when a bean instance is removed must be handled by the bean class. To do so you must implement the ejbRemove callback method.
- Query methods cannot be implemented using EJB QL or WebLogic QL. Instead the finder or select method(s) must define the database queries.

BMP Implementation

When you define a CMP entity bean in WebLogic, you should take into account the following:

- To create a new BMP entity bean file, the first step is creating it as you would a CMP entity bean (see How Do I: Create an Enterprise JavaBean?). When the file is created and opened, go to source view and add persistence−type="bmp" to the @ejbgen:entity tag. When you close and re−open the file, you will notice that design view is no longer available and, correspondingly, that the various wizards to for instance add an entity relation or a create method are no longer available.
- You need to add an @ejbgen:resource−ref tag to the datasource (the database). Because the EJB container doesn't handle persistence, you should remove the attributes data−source−name and table−name from the @ejbgen:entity tag.
- The database table for the BMP entity bean is not created for you by WebLogic. You must define the table in the database.
- You must define the property fields and accessor methods.

- You must define the findByPrimaryKey method and the various callback methods as mentioned in the previous section.
- WebLogic will still generate the various interfaces for you. To that effect, you also still use the appropriate @ejbgen tags to ensure that a particular method is added to the appropriate interface. The findByPrimaryKey method will automatically be added to the defined home interface(s).

An implemented BMP entity bean can be found in the BMP Entity Bean Sample.

Related Topics

BMP Entity Bean Sample

The Life Cycle of an Entity Bean

@ejbgen:entity Annotation

@ejbgen:resource−ref Annotation

Developing Session Beans

A session bean is used to model business processes or tasks for a client on the application server. The topics listed below discuss development of session beans.

Topics Included in This Section

Getting Started with Session Beans

This topic provides an overview of session beans.

Defining a Session Bean

This topic discusses how to create a session bean in WebLogic, what a session bean definition minimally must contain, and provides a short introduction to the various interfaces extended/implemented by an session bean definition.

The Life Cycle of a Session Bean

This topic discusses the life cycle of stateful and stateless session beans.

Related Topics

Tutorial: Enterprise JavaBeans

This tutorial provides a step−by−step guide to developing Enterprise JavaBeans.

Enterprise JavaBean Samples

This topic presents a collection of Enterprise JavaBean samples. Each sample describes a small set of EJB development features and techniques.

Enterprise JavaBeans

This 'How Do I...?' section presents a number of step−by−step procedures on how to develop Enterprise JavaBeans.

Enterprise JavaBean Annotations Reference

This topic provides reference documentation about the ejbgen tags used with Enterprise JavaBeans classes.

Getting Started with Session Beans

This topic provides an overview of session bean development. It contains the following sections:

- What are Session Beans?
- Stateful and Stateless
- Home and Business Interfaces
- Create Methods
- Component Methods
- Other Methods

What are Session Beans?

Session beans are used to execute business tasks for a client on the server. A session bean typically implements a certain kind of activity, such as ordering products or signing up for courses, and in executing the business rules typically invokes entity beans. For instance, ordering products is likely to involve stored information about products, customers, and credit cards, while signing up for courses is likely going to require invoking entity beans representing students and courses.

Stateful and Stateless

There are two types of session beans, stateful and stateless. A stateful session bean maintains conversational state. In other words, a stateful session bean remembers the calling client application from one method to the next. For a stateful session bean, the results produced by one method might be co−dependent on the results of its prior methods invoked by the same client. A stateful session bean maintains this conversation with the client until the conversation times out or the client explicitly ends the conversation by invoking the bean's remove method.

In contrast, a stateless session bean does not maintain any conversational state, that is, it does not remember which client invoked one of its methods, and does not maintain an internal state between methods. Each session bean method is independent, and the only client input is the data passed in its parameters.

Stateful session beans are tied to a particular client for the duration of the conversation, while stateless session beans are only tied to a particular client for the duration of a method execution. After method execution completes, a stateless session bean is ready to serve another client application. Consequently, a small number of stateless session beans can be used to serve a large number of client applications. Stateless session beans tend to be more commonly preferred over stateful session beans for this reason. When the client application is a page flow or a conversational web service, conversational state is remembered by the client application itself, making it possible to use a stateless session bean while maintaining a continuous session with the user of the client application. When you develop a new session bean in WebLogic, by default a stateless session bean is defined.

Home and Business Interfaces

Like an entity bean, a session bean can have four different interfaces, called the local home interface, the local business interface (or simply, the local interface), the remote home interface, and the remote business interface (or simply, the remote interface). The local interfaces define the bean's methods that can be used by other EJBs, EJB controls, web services and page flows defined within the same application. That is, if you define a session bean and only plan to use it within that application, you can use local interfaces. In contrast, the remote interfaces define the bean's methods that be invoked by EJBs, EJB controls, web services and page flows defined in other applications.

To determine what interfaces are defined for a session bean, ensure you are in Design View and go to the ***Naming*** section in the ***Property Editor***. The *Remote EJB* and *Local EJB* sections refer to the remote and local interfaces; within each section the *Home class name* refers to the home interface, and the *Bean class name* refers to the business interface. In Source View, the attributes are part of the @ejbgen:file−generation Annotation. When you define a new session bean, by default only the remote interfaces are defined.

A session bean's (remote or local) home interface contains the create methods used to obtain a reference to the bean instance. Its (remote or local) business interface contains the component methods that are used to encapsulate a particular piece of business logic.

The Create Methods

For a stateless session bean, you must define exactly one ejbCreate() method with no parameters. This method must be invoked to obtain to a reference to a session bean instance. Once you have obtained a reference, you can invoke the session bean's component methods. If you call a stateless session bean via an EJB control, you do not need to call the create method explicitly; the EJB control will create a reference for you when you call a component method.

A stateful session bean must have at least one ejbCreate method and, like entity beans, can have multiple ejbCreate methods. One of these methods must be invoked to obtain a reference to the session bean instance before you can invoke the session bean's component methods. If you call a stateful session bean via an EJB control, you must first call (one of) its create methods to obtain a reference.

Unlike with stateless session beans, when you can call a stateful session bean's create method to obtain a reference and subsequently invoke several component methods, each method is guaranteed to be handled by the same bean instance on the server. For more information, see The Life Cycle of a Session Bean.

Component Methods

Component methods are the business methods that are invoked on a session bean instance. A simple example of a business method is reserveTickets(customer, movieName, date), which is used to reserve tickets for a movie. For more information, see How Do I: Add a Component Method to an Entity or Session Bean?

In principle there is no difference between component methods for a stateful and a stateless session bean. However, the component methods of a stateless session bean must be passed all the necessary data to execute business logic as parameters, while this is not necessary for the component methods of a stateful session bean. For instance, for a stateful session bean the component method reserveTickets() can be used to make ticket reservations for a movie, after the component method setCustomer(customer) is called to set the customer data, setMovie(name) is called to make the movie selection, and setDate(date) is called to set the movie time. For a stateless session bean, these parameters must be passed to the component method making the actual reservations, as in reserveTickets(customer, movieName, date).

Other Methods

A session bean has several predefined methods, as well as a number of callback methods, invoked by the EJB container during certain operations, that a session bean must implement. In WebLogic these callback methods are by default automatically implemented. In many cases you will find it unnecessary to use these methods. To learn more about these methods, see Defining a Session Bean and The Life Cycle of a Session Bean.

Related Topics

Message−Driven Bean Sample

Value Object Sample

EJB Control

Defining a Session Bean

This topic discusses how to create a session bean, what a session bean minimally must contain, and provides an overview of the various interfaces extended by a session bean. This topic contains the following sections:

- Creating a Session Bean
- Defining a Basic Session Bean
- Predefined and Callback Methods

Creating a Session Bean

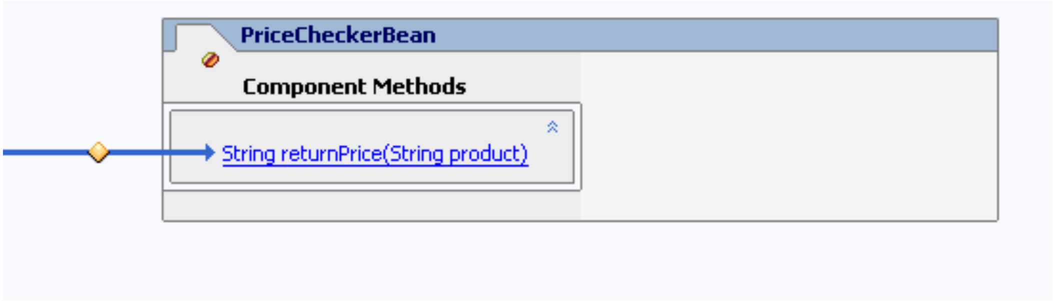
To create a session bean, you can choose one of the following methods:

- ***Create a session bean from scratch***. If you are designing a new session bean in EJB project, you can define a new session bean. To find out exactly how to do this, see How Do I: Create an Enterprise JavaBean? When you create a new session bean, by default the remote interfaces and various other defaults are defined. For more details, see @ejbgen:file−generation Annotation.
- ***Import a session bean***. If you have developed session beans in another development environment, you can import these into WebLogic. To import an EJB, you need the EJB Jar file as well as the source files. You can import multiple EJBs at the same time. For more information, see How Do I: Import an Enterprise JavaBean? After you have imported a session bean, you can enhance its definition.

  Note. If you have existing session beans that you plan to invoke in the application, for instance via another EJB or an EJB control, but you do not intend to change their definitions, you can suffice by adding the EJB Jar to the application. For more information, see How Do I: Add an Existing Enterprise JavaBean to an Application?

Defining a Basic Session Bean

The following figure shows the design view of a basic session bean called *PriceCheckerBean*:



This stateless session bean's component method receives the name of a product and returns the price when known, or a 'product unknown' message if the product cannot be found. It uses the *ProductBean*, shown in Defining an Entity Bean, to look up a product in the database and return its price. The source code of the PriceChecker bean is given below:

```
package myBeans;
```

```
import javax.ejb.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import weblogic.ejb.*;

/**
 * @ejbgen:session
 *   ejb−name = "PriceChecker"
 *
 * @ejbgen:jndi−name local="ejb.PriceCheckerLocalHome"
 *
 * @ejbgen:file−generation remote−class="false" remote−home="false" local−class="true"
 *   local−class−name = "PriceCheckerLocal" local−home="true" local−home−name = "PriceCheckerLo
 *
 * @ejbgen:ejb−local−ref link="Product"
 */
public class PriceCheckerBean extends GenericSessionBean implements SessionBean
{
  private ProductHome productHome;

  public void ejbCreate() {
     try {
        javax.naming.Context ic = new InitialContext();
        productHome = (ProductHome)ic.lookup("java:comp/env/ejb/Product");
     }
     catch (NamingException ne) {
        throw new EJBException(ne);
     }
  }

  /**
   * @ejbgen:local−method
   */
  public String returnPrice(String product)
  {
     Product theProduct;
     int visitNumber;

     try {
        theProduct = productHome.findByPrimaryKey(product);
     }
     catch(FinderException fe) {
        return "Product not known";
     }
     return "The price of this product is " + theProduct.getPrice();
  }
}
```

The @ejbgen:session annotation contains the actual name of the session bean. For stateful session beans this tag will contain the attribute type="Stateful". In the ejbCreate method a reference to the Product entity bean's local home interface is obtained. The JNDI reference *Product* used in the lookup method to look up the Product bean, is mapped to this bean's local interface using an @ejbgen:ejb−local−ref Annotation, which is defined at the top of the PriceChecker bean class definition. To learn more about JNDI naming, consult your favorite J2EE documentation or go to http://java.sun.com.

The method returnPrice implements the business logic for this class. It finds a particular product using the Product bean and returns its price. If the product cannot be found in the database, it returns a *Product not known* message instead.

In WebLogic, all the information needed to make a session bean are stored in a single file, instead of separate JAVA files for the bean class, the local business interface, the local home interface, and so forth. When you build a session bean, these classes are auto−generated. Various ejbgen annotations are used to hold the information required to make this generation possible. Specifically, the @ejbgen:file−generation annotation specifies the names of the local home and business interface for the PriceChecker bean, and the @ejbgen:local−method annotation on the component method specifies that the method should be defined in the local business interface. To verify that these JAVA and corresponding CLASS files are generated, expand the JAR file created during a build (located in the ***Modules*** folder in the ***Application*** pane), and locate and open the generated files in the folder reflecting the package name. For the PriceCheckerBean, the PriceCheckerBean.java (bean definition), PriceCheckerLocal.java (local interface definition), and PriceCheckerLocalHome.java (local home interface definition) files are auto−generated.

Predefined and Callback Methods

The interfaces of session (and entity) beans extend a particular interface which contains various useful methods. Specifically:

- The local interface extends javax.ejb.EJBLocalObject
- The local home interface extends javax.ejb.EJBLocalHome
- The remote interface extends javax.ejb.EJBObject
- The remote home interface extends javax.ejb.EJBHome

For example, the interfaces contain a remove method to remove a bean instance and, for a stateful session bean, end the conversation. Complete details about these interfaces and the methods they define can be found in your favorite J2EE documentation and the API reference at http://java.sun.com.

Every session bean must implement the javax.ejb.SessionBean interface. This interface defines callback methods that are called by the EJB container at specific times. The callback methods are setSessionContext, ejbActivate, ejbPassivate, and ejbRemove. When you define a session bean from scratch, it will extend weblogic.ejb.GenericSessionBean, which contains empty implementations of these callback methods. In other words, you will only need to define these methods if you need to override the empty implementation. If you import a session bean, these callback methods will probably be implemented directly in the bean's ejb file. For more details about the callback methods and their role in the interaction between the session bean and the EJB container, see The Life Cycle of a Session Bean.

Related Topics

The Life Cycle of a Session Bean

@ejbgen:file−generation Annotation

@ejbgen:local−method Annotation

@ejbgen:session Annotation

The Life Cycle of a Session Bean

This topic discusses the life cycle methods of session beans and contains the following sections:

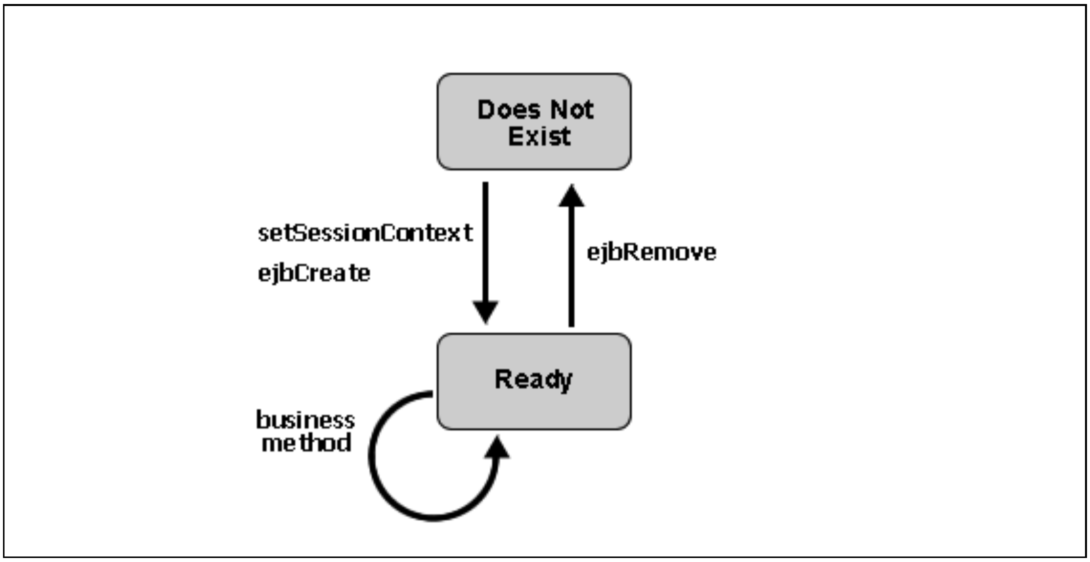
- The Life Cycle of a Stateless Session Bean
- The Life Cycle of a Stateful Session Bean

The Life Cycle of a Stateless Session Bean

The following figure shows the life cycle of a stateless session bean. A stateless session bean has two states:

- ***Does Not Exist***. In this state, the bean instance simply does not exist.
- ***Ready***. When WebLogic server is first started, several bean instances are created and placed in the Ready pool. More instances might be created by the container as needed by the EJB container.

The various state transitions as well as the methods available during the various states are discussed below.



Moving from the Does Not Exist to the Ready State

When the EJB container creates a stateless session bean instance to be placed in the ready pool, it calls the callback method public void setSessionContext(SessionContext ctx). This method has the parameter javax.ejb.SessionContext, which contains a reference to the session context, that is, the interface to the EJB container, and can be used to self−reference the session bean object. Complete details about the javax.ejb.SessionContext can be found in your favorite J2EE documentation and the API reference at http://java.sun.com.

After the callback method setSessionContext is called, the EJB container calls the callback method ejbCreate. You can implement this callback method to for instance obtain the home interfaces of other EJBs invoked by the session bean, as shown in Defining a Session Bean. The ejbCreate method is only called once during the lifetime of a session bean, and is not tied to the calling of the create method by a client application. For a stateless session bean, calling the create method returns a reference to a bean instance already in the ready pool; it does not create a new bean instance. The management of stateless session bean instances is fully done by the EJB container.

The Ready State

When a bean instance is in the ready state, it can service client request, that is, execute component methods. When a client invokes a business method, the EJB container assign an available bean instance to execute the business method. Once executed, the session bean instance is ready to execute another business method.

Moving from the Ready to the Does Not Exist State

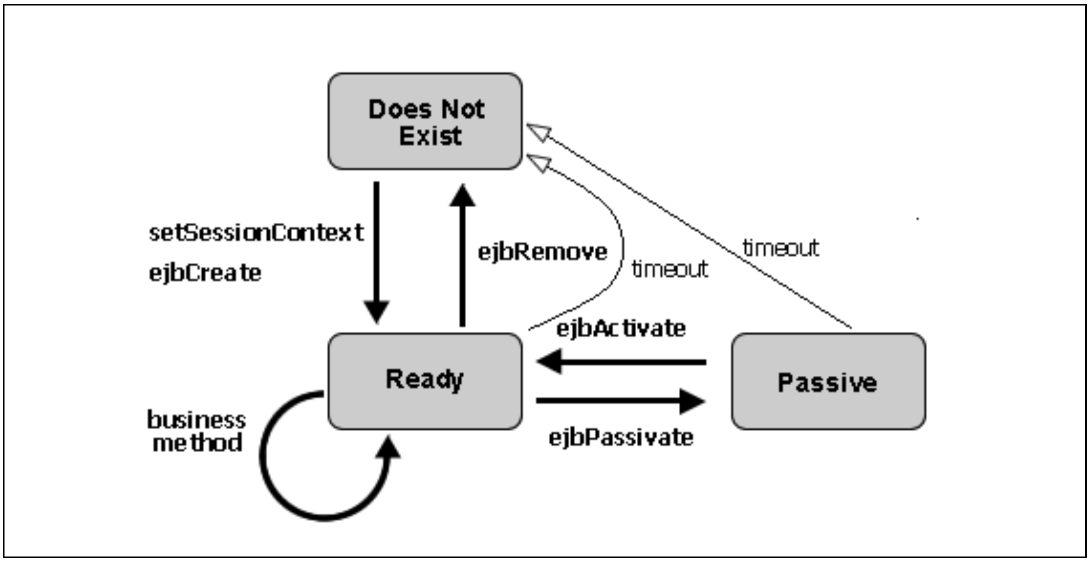
When the EJB container decides to reduce the number of session bean instances in the ready pool, it makes the bean instance ready for garbage collection. Just prior to doing this, it calls the callback method ejbRemove. If your session bean needs to execute some cleanup action prior to garbage collection, you can implement it using this callback method. The callback method is not tied to the remove method invoked by a client. For a stateless session bean, calling the remove method invalidates the reference to the bean instance already in the ready pool, but it does not move a bean instance from the ready to the does not exist state, as the management of stateless session bean instances is fully done by the EJB container.

The Life Cycle of a Stateful Session Bean

The following figure shows the life cycle of a stateful session bean. It has the following states:

- ***Does Not Exist***. In this state, the bean instance simply does not exist.
- ***Ready***. A bean instance in the ready state is tied to particular client and engaged in a conversation.
- ***Passive***. A bean instance in the passive state is passivated to conserve resource.

The various state transitions as well as the methods available during the various states are discussed below.



Moving from the Does Not Exist to the Ready State

When a client invokes a create method on a stateful session bean, the EJB container creates a new instance and invokes the callback method public void setSessionContext(SessionContext ctx). This method has the parameter javax.ejb.SessionContext, which contains a reference to the session context, that is, the interface to the EJB container, and can be used to self−reference the session bean object. Complete details about the javax.ejb.SessionContext can be found in your favorite J2EE documentation and the API reference at http://java.sun.com. After the callback method setSessionContext is called, the EJB container calls the callback method ejbCreate that matches the signature of the create method.

The Ready State

A stateful bean instance in the ready state is tied to a particular client for the duration of their conversation. During this conversation the instance can the execute component methods invoked by the client.

Activation and Passivation

To more optimally manage resources, the EJB container might passivate an inactive stateful session bean instance by moving it from the ready state to the passive state. When a session bean instance is passivated, its (non−transient) data is serialized and written to disk, after which the the bean instance is purged from memory. Just prior to serialization, the callback method ejbPassivate is invoked. If your session bean needs to execute some custom logic prior to passivation, you can implement it using this callback method.

If after passivation a client application continues the conversation by invoking a business method, the passivated bean instance is reactivated; its data stored on disk is used to restore the bean instance state. Right after the state has been restored, the callback method ejbActivate is invoked. If your session bean needs to execute some custom logic after activation, you can implement it using this callback method.The caller (a client application or another EJB) of the session bean instance will be unaware of passivation (and reactivation) having taken place.

If a stateful session bean is set up to use the NRU (not recently used) cache−type algorithm, the session bean can time out while in passivated state. When this happens, it moves to the does not exist state, that is, it is removed. Prior to removal the EJB container will call the callback method ejbRemove. If a stateful session bean is set up to use the LRU (least recently used) algorithm, it cannot time out while in passivated state. Instead this session bean is always moved from the ready state to the passivated state when it times out.

The exact timeout can be set using the idle−timeout−seconds attribute on the @ejbgen:session annotation. The cache−type algorithm can be set using the cache−type attribute on the same annotation.

Moving from the Ready to the Does Not Exist State

When a client application invokes a remove method on the stateful session bean, it terminates the conversation and tells the EJB container to remove the instance. Just prior to deleting the instance, the EJB container will call the callback method ejbRemove. If your session bean needs to execute some custom logic prior to deletion, you can implement it using this callback method.

An inactive stateful session bean that is set up to use the NRU (not recently used) cache−type algorithm can time out, which moves it to the does not exist state, that is, it is removed. Prior to removal the EJB container will call the callback method ejbRemove. If a stateful session bean set up to use the LRU (least recently used) algorithm times out, it always moves to the passivated state, and is not removed.

The exact timeout can be set using the idle−timeout−seconds attribute on the @ejbgen:session annotation. The cache−type algorithm can be set using the cache−type attribute on the same annotation.

Related Topics

The Life Cycle of an Entity Bean

@ejbgen:session Annotation

Developing Message−Driven Beans

An message−driven EJB is used to receive and process asynchronous messages using JMS. Message−driven EJBs are never directly invoked by other EJBs. However, they in turn can invoke methods of session and entity beans and send JMS messages to be processed by other message−driven EJBs. The topics listed below discuss development of message−driven beans.

Topics Included in This Section

Getting Started with Message−Driven Beans

This topic introduces message−driven bean development.

Sending JMS Messages

This topic discusses how to send JMS messages to−be−received by message−driven beans.

Processing JMS Messages

This topic discusses how message−driven beans can be used to process JMS messages.

Related Topics

Tutorial: Enterprise JavaBeans

This tutorial provides a step−by−step guide to developing Enterprise JavaBeans.

Enterprise JavaBean Samples

This topic presents a collection of Enterprise JavaBean samples. Each sample describes a small set of EJB development features and techniques.

Enterprise JavaBeans

This 'How Do I...?' section presents a number of step−by−step procedures on how to develop Enterprise JavaBeans.

Enterprise JavaBean Annotations Reference

This topic provides reference documentation about the ejbgen tags used with Enterprise JavaBeans classes.

Getting Started with Message−Driven Beans

This topic provides an overview of message−driven bean development. It contains the following sections:

- What Are Message−Driven Beans?
- Asynchronous and Concurrent Processing
- Topics and Queues
- The Life Cycle of a Message−Driven Bean

What are Message−Driven Beans?

Message−driven beans are server−side objects used only to process JMS messages. These beans are stateless, in that each method invocation is independent from the next. Unlike session and entity beans, message−driven beans are not invoked by other beans or client applications. Instead a message−driven bean responds to a JMS message.

Because message−driven beans are not invoked by other EJBs or clients, these beans do not have interfaces. For each message−driven bean a single method, onMessage, is defined to process a JMS message. Although message−driven beans cannot be invoked by other EJBs, they can in turn invoke other EJBs. Also, message−driven beans can send JMS messages. As with the other types of EJBs, the EJB container is responsible for managing the beans environment, including making enough instances available for processing and message−acknowledgement.

Asynchronous and Concurrent Processing

A core feature of message−driven beans is the notion of asynchronous processing. A client application can send a JMS message to execute a certain business task. After the message has been sent, the client application can continue right away and does not have to wait for the JMS message to be received and processed. This is especially helpful when the business task is complex, requires the use of entity (and session) beans, and takes a long time to complete. In contrast, if the same client application were to use a session bean to execute a certain business task, it would have to wait until the session bean method completed and returned control to the client application. The message façade design pattern formalizes this use of message−driven beans as an intermediary between client applications and entity beans to achieve asynchrony. An example of this pattern is shown in the Message−Driven Bean Sample.

Another important feature of message−driven beans is that JMS messages are processed concurrently. That is, although each bean instance handles a message at a time, the EJB container takes care of creating enough bean instances to handle the message load at a given moment. In WebLogic you can set the initial number and max number of bean instances created by the container. For more information, see the @ejbgen:message−driven Annotation.

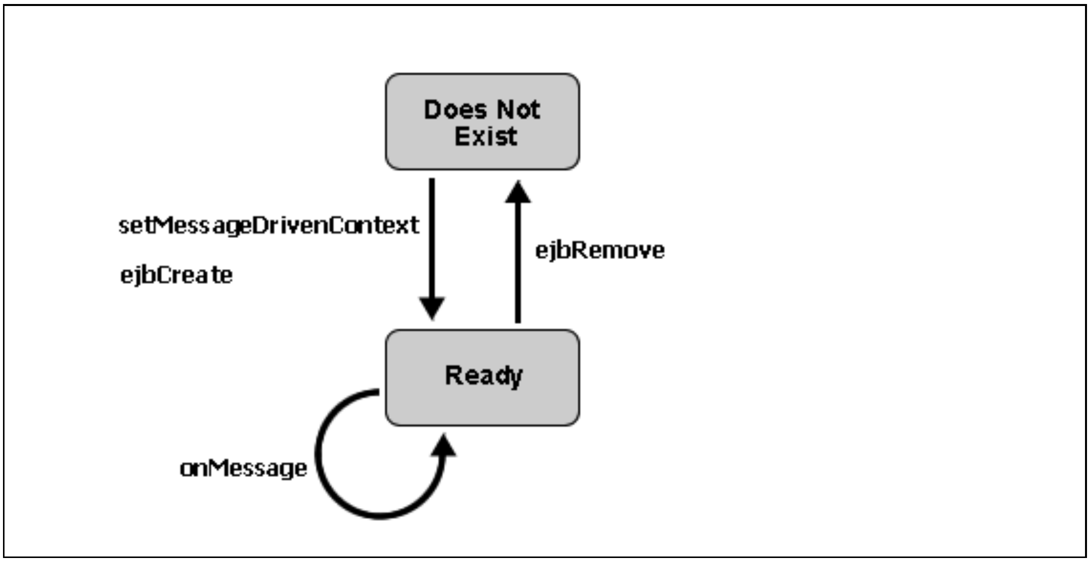
Because message−driven beans are stateless and processing of JMS messages occurs in an asynchronous message, there is no guarantee that messages are processed in the order they were sent. Therefore, sending multiple messages such that one message is dependent on the successful processing of another message might cause unexpected results. Instead, you should reconsider the granularity of your business task such that one message can initiate its execution, possibly by handling one piece of the task, and then sending a JMS message to be processed by another message−driven bean for the remainder of the business task.

Topics and Queues

A message−driven bean listens to a particular channel for JMS messages. There are two types of channels, namely topics and queues. Topics implement the publish−and−subscribe messaging model, in which a given message can be received by many different subscribers, that is, many different message−driven bean classes (not instances) listening to the same topic. In contrast, queues implement the point−to−point messaging model, in which a given message is received by exactly one message−driven bean class, even when multiple classes are listening to this queue.

The Life Cycle of a Message−Driven Bean

The EJB container is responsible for creating a pool of message−driven bean instances. When it creates an instance, it calls the setMessageDrivenContext() and the ejbCreate() methods. At this point the message−driven bean is ready to receive messages. When a bean instance is processing a JMS message, its onMessage method is being invoked. When the EJB container removes a bean instance, it first calls the ejbRemove method before the instance is ready for garbage collection. The life cycle of a message−driven bean is depicted in the following figure.



When defining a message−driven bean in WebLogic, in most cases you will implement the onMessage method to execute a particular business task, and use the ejbCreate method to implement actions that only need to be executed once, such as looking up the home interfaces of entity beans that are invoked by the message−driven bean's onMessage method. A typical example of simple message−driven bean is given below. Notice that the ejbCreate method is used to find the home interface of a Recording entity bean, while the onMessage method processes the message and invokes the Recording bean:

```
/**
 * @ejbgen:message−driven
 *   ejb−name = Statistics
 *   destination−jndi−name="jms.EJBTutorialSampleJmsQ"
 *   destination−type = javax.jms.Queue
 *
 * @ejbgen:ejb−local−ref
 * type="Entity"
 * name="ejb/recordLink"
 * local="bands.Recording"
 * link="Recording"
 * home="bands.RecordingHome"
 */
public class StatisticsBean extends GenericMessageDrivenBean implements MessageDrivenBean, Mess
{
  private RecordingHome recordingHome;

  public void ejbCreate() {
    try {
        javax.naming.Context ic = new InitialContext();
        recordingHome = (RecordingHome)ic.lookup("java:/comp/env/ejb/recordLink");
    }
    catch(NamingException ne) {
        System.out.println("Encountered the following naming exception: " + ne.getMessage());
    }
  }

  public void onMessage(Message msg) {
     try {
         // read the message
         MapMessage recordingMsg = (MapMessage)msg;
         String bandName = recordingMsg.getString("bandName");
         String recordingTitle = recordingMsg.getString("recordingTitle");
         ...
         // save the rating with the recording
         Recording album = recordingHome.findByPrimaryKey(new RecordingBeanPK(bandName, recordi
         album.setRating(rating);
     }
     catch(Exception e) {
        System.out.println("Encountered the following exception: " + e.getMessage());
     }
  }
}
```

You can implement the ejbRemove method if cleanup is required before the object is removed, and you can implement setMessageDrivenContext method if you need access to the javax.ejb.MessageDrivenContext provided by the EJB container. The MessageDrivenContext contains information about the container, in particular its transaction methods; for more information, see your favorite J2EE documentation. A message−driven bean defined in WebLogic by default extends weblogic.ejb.GenericMessageDrivenBean, which provides empty definitions for all these methods with the exception of the onMessage method; the definition of your bean must therefore implement the onMessage method.

Related Topics

Message−Driven Bean Sample

Sending JMS Messages

This topic discusses how to send JMS messages to be processed by message−driven beans from page flows, web services, and EJBs. It contains the following sections:

- Using a JMS Control
- Sending Messages from EJBs
- JMS Message Types

Note. This topic discusses only how to use JMS in the context of EJBs, page flows, and web services. To learn more about the JMS API, see your favorite J2EE book or the Java documentation at http://java.sun.com.

Using a JMS Control

In WebLogic you can use a JMS control to send JMS messages to a topic or queue. JMS controls can be used to send messages in page flows and web services; they cannot be used to send JMS messages in EJBs. You can use JMS controls to both send and receive messages. In this case you will only be using the JMS control to send messages, as a message−driven bean will be used to process received messages. For details on how to create JMS controls for topics and queues, see JMS Control.

The advantage of using a JMS control is that the details of the JMS messaging API are by and large taken care of by the JMS control. For example, the following code samples shows the definition of a JMS control to send messages to a queue:

```
/**
 *   @jc:jms send−type="queue" send−jndi−name="jms.SamplesAppMDBQ"
 *            connection−factory−jndi−name="weblogic.jws.jms.QueueConnectionFactory"
 */
public interface sendToQueueControl extends JMSControl, com.bea.control.ControlExtension
{
    /**
     * this method will send a javax.jms.Message to send−jndi−name
     */
    public void sendJMSMessage(Message msg);
    static final long serialVersionUID = 1L;
}
```

As the definition shows, you must still define the name of the queue to which to send the JMS message as well as the name of a QueueConnectionFactory. However, to send a message using this JMS control, you simply need to invoke the method sendJMSMessage and pass a message as the parameter. The JMS control handles the creation of a factory instance, a connection, session, and so forth.

When using a JMS control from a page flow or web service, you can use a built−in control to encapsulate all the message sending related logic, instead of defining this directly in the page flow or web service. For instance, the following built−in control code snippet uses the JMS control defined above to send 20 messages at one time to a queue.

```
...
public class MessageSenderImpl implements MessageSender, ControlSource
{
    /**
     * @common:control
     */
    private messageDriven.sendToQueueControl queueSend;

    /**
     * @common:operation
     * @common:message−buffer enable="true"
     */
    public void add20ViaQueue(int currentNum) throws JMSException
    {
        String name;
        for(int i = 0; i <20; i++) {
           MapMessage msg = queueSend.getSession().createMapMessage();
           msg.setStringProperty("Command", "Add");
           name = Integer.toString(currentNum + i);
           msg.setString("tokenName", name);
           queueSend.sendJMSMessage(msg);
        }
    }
}
```

For more information on built−in controls, see Building Custom Java Controls. The use of JMS controls and built−in controls in a page flow is demonstrated in the Message−Driven Bean Sample.

Sending Messages from EJBs

To send JMS message from an EJB, you cannot use JMS controls and must use the 'standard' JMS messaging API. (You can also use the standard messaging API in page flows and web services if you do not want to use a JMS control.) The following code sample shows a session bean sending a message to a queue. The queue and QueueConnectionFactory are defined using @ejbgen:resource−env−ref and @ejbgen:resource−ref annotations. In the bean's ejbCreate method the queue and QueueConnectionFactory are located (this only need to be done once), while the method executeTask contains the correct procedure to create a connect, session, sender, and message before sending the JMS message.

```
/**

 * @ejbgen:resource−ref
 *   auth="Container"
 *   jndi−name = "weblogic.jws.jms.QueueConnectionFactory"
 *   name = "jms/QueueConnectionFactory"
 *   type="javax.jms.QueueConnectionFactory"
 *
 * @ejbgen:resource−env−ref
 *  jndi−name="jms.ASampleJmsQ"
 *  name="jms/ASampleJmsQ"
 *  type = "javax.jms.Queue"
 *
 * ...
 */
public class ASessionBean extends GenericSessionBean implements SessionBean
{
   private QueueConnectionFactory factory;
   private Queue queue;

   public void ejbCreate() {
      try {
         javax.naming.Context ic = new InitialContext();
         factory = (QueueConnectionFactory) ic.lookup("java:comp/env/jms/QueueConnectionFactory
         queue = (Queue) ic.lookup("java:comp/env/jms/ASampleJmsQ");
      }
      catch (NamingException ne) {
         throw new EJBException(ne);
      }
   }

   /**
    * @ejbgen:local−method
    */
   public void executeTask(String prop1, String prop2)
   {
     try {
        //interpret the properties
        ...
        //send a message
        QueueConnection connect = factory.createQueueConnection();
        QueueSession session = connect.createQueueSession(true,Session.AUTO_ACKNOWLEDGE);
        QueueSender sender = session.createSender(queue);
        MapMessage recordingMsg = session.createMapMessage();
        recordingMsg.setString("Property1", prop1);
        recordingMsg.setString("Property2", prop2);
        sender.send(recordingMsg);
        connect.close();
     }
     catch(JMSException ne) {
        throw new EJBException(ne);
     }
   }
}
```

JMS Message Types

The JMS messaging API defines various message types, allowing for different types of information to be included in the body of a JMS message. Frequently used message types are *TextMessage* to send a text string, *MapMessage* to create sets of name−value pairs, and *ObjectMessage* to send a serializable object. For more details on the various message types, see your favorite J2EE documentation or the javax.jms.Message API documentation at http://java.sun.com.

Message Selectors

A JMS message can also contain message properties. There are different types of properties which can serve a number of functions. One common use of a message property is to set a message selector. A message selector is a property that is used by a message−driven bean to determine whether it should process this message or should leave its processing up to another message−driven bean listening to the same topic or queue.

The following example shows how to set a String property that will act as a message selector for a JMS MapMessage:

```
Message msg = session.createMapMessage();
msg.setStringProperty("MyMessage", "Build 126"");
```

A message−driven bean that is defined to specifically process this type of message will include a message selector property:

```
 * @ejbgen:message−driven
 *   message−selector="MyMessage = 'Build126'"
 *   ejb−name = MDBean
 * ...
 */
public class MDBean extends GenericMessageDrivenBean implements MessageDrivenBean, MessageListe
{
   ...
```

In other words, this bean will respond specifically to this type of message and will leave the processing of other messages up to other beans listening to this topic or queue. By using a message selector you can send different types of messages to the same channel instead of having to create a separate queue/topic for every type of message. For more details on the various message properties, see your favorite J2EE documentation or the javax.jms.Message API documentation at http://java.sun.com.

Related Topics

JMS Control

Building Custom Java Controls

Message−Driven Bean Sample

@ejbgen:resource−env−ref Annotation

@ejbgen:resource−ref Annotation

Tutorial: Enterprise JavaBeans

Processing JMS Messages

When a JMS message is sent to a topic or queue, a message−driven bean instance will interpret and process the message, as specified in its onMessage method. When a JMS message is sent to a topic, a publish−and−subscribe system, an instance of every message−driven bean class listening to this topic will in principle receive and process this message. However, if the message contains a message selector, only the message−driven bean(s) matching the message selector will process the message. When a JMS message is sent to a queue, a point−to−point system, only one message−driven bean will process the message, even when multiple bean classes are listening to the queue. Again, the use of a message selector might limit the bean processing the message. For more on message selectors, see Sending JMS Messages.

How the JMS message is processed fully depends on the business task that is being modeled. It might range from simply logging the message to executing a range of different tasks which include invoking methods on session and entity beans, or sending JMS messages to be processed by other message−driven beans. The following code sample shows one use of a message−driven bean. This bean responds only to JMS messages delivered via the jms/SamplesAppMDBQ queue and contain the message selector Command= 'Delete'. When processing a JMS message, an instance of this bean invokes the query method findAll of the entity bean *SimpleToken*, and subsequently deletes all SimpleToken records from the underlying database.

```
/**

 * @ejbgen:message−driven
 *   message−selector="Command = 'Delete'"
 *   ejb−name = DeleteViaQMD
 *   destination−jndi−name = jms/SamplesAppMDBQ
 *   destination−type = javax.jms.Queue
 *
 *  @ejbgen:ejb−local−ref link="SimpleToken"
 */
public class DeleteViaQMDBean
  extends GenericMessageDrivenBean
  implements MessageDrivenBean, MessageListener
{
  private SimpleTokenHome tokenHome;

  public void ejbCreate() {
     try {
        javax.naming.Context ic = new InitialContext();
        tokenHome = (SimpleTokenHome)ic.lookup("java:/comp/env/ejb/SimpleToken");
     }
     catch(NamingException ne) {
        System.out.println("Encountered the following naming exception: " + ne.getMessage());
     }
  }

  public void onMessage(Message msg) {
     try {
        Iterator allIter = tokenHome.findAll().iterator();
        while(allIter.hasNext()) {
            ((SimpleToken) allIter.next()).remove();
        }
     }
     catch(Exception e) {
        System.out.println("Encountered the following exception: " + e.getMessage());
     }
  }
}
```

Acknowledgement and Transactions

When a message−driven bean instance receives a message, and it is not part of a container−managed transaction, by default it immediately sends an acknowledgement to the JMS provider notifying that the message has been received. This will prevent the JMS provider from attempting to redeliver it. However, the acknowledgement only indicates that a message has been successfully received, but it does not guarantee that the message is successfully processed. For instance, a system exception might occur when attempting to locate an entity bean or update its records, causing the processing to fail.

If you want to ensure that a message is redelivered when processing fails, you can make the message−driven bean be part of a transaction. The easiest approach is to use container−managed transaction, where the EJB container is responsible for transaction management. To enable container−managed transaction for a message−driven bean, make sure that your cursor is placed inside the ejbgen:message−driven tag and use the ***Property Editor*** to set the ***transaction−type*** to *Container* and the ***default−transaction*** to *Required*. When JMS message processing executes successfully, any changes made during this business task, such as update to entity bean records, are committed and acknowledgement is sent to the JMS provider. However, when JMS message processing fails due to a system exception, any changes are rolled back and receipt is not acknowledged. In other words, after processing fails, the JMS provider will attempt to resend the JMS message until processing is successfully or until the maximum number of redelivery attempts specified for this topic or queue has been reached.

Note. When a message−driven bean is part of a transaction, it executes as part of its own transaction. In other words, if the transaction fails, changes that were made as part of the onMessage method are rolled back, but the occurrence of an exception has no direct effect on the EJB or client application sending the JMS message, as the sender and the message−driven bean are decoupled.

To learn more about container−managed transactions, see EJBs and Transactions. To learn more about bean−managed transaction (that is, explicit transaction management), please refer to your favorite J2EE documentation.

Related Topics

Message−Driven Bean Sample

EJBs and Transactions

@ejbgen:message−driven Annotation

Advanced Enterprise JavaBeans Topics

The following topics describe various advanced scenarios of Enterprise JavaBean development and a variety of other specialistic topics.

Topics Included in This Section

Switching EJB Development to WebLogic Workshop

This topic discusses a number of issues that are particularly of interests to experienced EJB developers who have recently switched to using the WebLogic platform.

EJBGen Tag Inheritance

This topic describes how to set up EJBGen tag inheritance.

EJBs and Transactions

This topic describes how to specify transactions for EJBs.

Related Topics

Role−Based Security

This topic discusses role−based security, including for Enterprise JavaBeans.

Handling EJB Exceptions

This topic discusses how to handle exceptions thrown by EJBs and EJB Controls.

Enterprise JavaBeans Annotations Reference

This topic provides documentation about the ejbgen tags used with Enterprise JavaBeans.

Switching EJB Development to WebLogic Workshop

This topic discusses a number of issues that are particularly of interests to experienced EJB developers who have recently switched to using the WebLogic platform. It includes the following sections:

- Using Existing Enterprise JavaBeans
- Building and Deploying Enterprise JavaBeans
- Iterative Development

Using Existing Enterprise JavaBeans

If you have existing EJBs that you want to use in WebLogic Workshop, you can import EJBs or add EJBs as modules to your WebLogic Workshop applications. If you are switching from a WebLogic Server environment, you could also WebLogic Workshop−enable the Server domain.

Importing EJBs

If you want to have existing EJBs available in the WebLogic Workshop for further development, you can import EJBs into EJB project. To import EJBs, you need the EJBs' source code and the corresponding EJB JAR(s). An Import Wizard guides you through this process and, for each EJB, translates the EJB's source file into one bean file with an *ejb* extension, inserting the necessary ejbgen tags to define interfaces, deployment descriptor files, CMP and entity relation accessor methods, and so forth.

For more information on ejbgen tags, see Getting Started with EJB Project. For step−by−step instructions on importing EJBs, see How Do I: Import an Enterprise JavaBean?

Add EJBs as a Module

If you have existing EJBs that you want to use within the application, but you don't need to do further development on these EJBs, you can add the EJB JAR as a module. When you add the EJB JAR as a module, the EJBs become application−scoped, that is, they are deployed as part of the application. New EJBs that you develop in EJB project can reference these EJBs using the local or remote interfaces, and you can create EJB controls to extend either interfaces (see below). Unlike importing EJBs, when you add the EJB JAR as a module, the JAR is added as is.

For step−by−step instructions on adding EJBs as a module, see How Do I: Add an Existing Enterprise JavaBean to an Application?

Enabling the WebLogic Server Domain

If you previously worked with WebLogic Server, you can WebLogic Workshop−Enable this domain, which allows you to continue using this domain in WebLogic Workshop. EJBs that are deployed on the server can be used in a WebLogic Workshop applications through these EJBs' remote interfaces.

For step−by−step instructions on enabling the server domain, see How Do I: WebLogic Workshop−Enable an Existing WebLogic Server Domain? If you want to create EJB controls for globally−scoped (or server scoped) EJBs in your application, you will need to add the corresponding EJB JAR as a library. For step−by−step instructions on adding EJBs as a library, see How Do I: Add an Existing Enterprise JavaBean to an

Application?

Building and Deploying Enterprise JavaBeans

When you are ready to build, you can build each EJB project in your application separately, or you can build the entire application. When you run a build:

- The source code in the EJB file is compiled.
- The EJB JAR is created in the Modules folder in the ***Application*** pane, containing the containing the various JAVA and CLASS files for the bean class, its interfaces, and any dependent value or primary key classes, as well as the deployment descriptor for these beans.
- Application−related files, such as application.xml and the .work file, are updated.
- The EJBs are (re)deployed on the server.

In WebLogic Workshop, EJB JARs are deployed on the server as part of the application. You can verify that the EJBs are correctly deployed using the WebLogic Console, which you can access by choosing ***WebLogic Server***−−>***WebLogic Console*** from the ***Tools*** menu. When an EJBs has its remote interfaces defined, it is also globally−scoped and can be referenced from other applications. Finally, it is also possible to deploy a stand−alone EJB JAR on the WebLogic Server instead using the WebLogic Console.

For step−by−step instructions on checking deployed EJBs through the WebLogic Server Console, see How Do I: Test an Enterprise JavaBean? For step−by−step instructions on deploying a stand−alone, EJB JAR on the WebLogic Server Console, see How Do I: Deploy a Stand−Alone Enterprise JavaBean on WebLogic Server?

Iterative Development

After building and deploying an EJB, WebLogic Workshop offers various alternatives to test the EJB. One approach is to create a web application to test the EJB. This requires that you develop the web applications in parallel to take into account changes to the EJB. An alternative is to use a test web service, with which you can directly invoke the EJB methods. The test web service is generated by WebLogic Workshop on the basis of an EJB Control. An EJB control is a WebLogic Workshop component used to locate and reference an EJB on the basis of its home or remote interfaces, and JNDI name or an application relative path. When you change the definition of the interfaces that the EJB control extends, you can simply rebuilds the EJBs, which triggers EJB redeployment, and regenerate the web service to incorporate the changes to the interface definitions.

For step−by−step instructions on using EJB controls and Test Web Services, see How Do I: Test an Enterprise JavaBean? The EJB Tutorial, in particular Step 9: Build and Run a Web Service Application, gives a specific example of creating and using an EJB control and a test web service. For more information on the debugger, see Debugging Your Application. For more information one EJB controls and Java Control projects, see Creating a New EJB Control and How Do I: Use an Enterprise JavaBean in a Web Application? The latter topic and the EJB Tutorial, in particular Step 4: Run a Web Application, also show how to use EJB controls or JNDI API to call an EJB from a web application.

Related Topics

Getting Started

Getting Started with WebLogic Workshop

EJBGen Tag Inheritance

When you develop Enterprise JavaBeans and want to set default values for ejbgen properties, you can use tag inheritance. For example, if you want the value for the max−beans−in−cache property to be 3000 for all entity beans that you develop, you can create a superclass that has this property set. You can inherit entire tags and individual attributes on a tag, and you can override an inherited tag (attribute) in a subclass if necessary.

An Example

In this example a BaseEntityBean is defined with several tags. The ejbgen:env−entry tag will be entirely inherited as well as the max−beans−in−cache attribute on the ejbgen:entity tag. A code snippet of the BaseEntityBean is given first:

```
 /**
 * @ejbgen:env−entry value="MyCompany" type="java.lang.String" name="CompanyName"
 *
 * @ejbgen:entity prim−key−class="Integer" max−beans−in−cache="3000"
 *   ejb−name = "BaseEntity"
 *
 * @ejbgen:...
 */
abstract public class BaseEntityBean
  extends GenericEntityBean
  implements EntityBean
{
   //Methods you want to be inherited
   ...
}
```

To inherit these tags, I simply need to create a subclass of the BaseEntityBean:

```
/**
 * @ejbgen:entity prim−key−class="java.lang.Integer"
 *   ejb−name = "InheritingBean"
 *   table−name = "InheritingBean"
 *   abstract−schema−name = "InheritingBean"
 *
 *   ...
 */
public abstract class InheritingBean extends BaseEntityBean
{
   ...
}
```

The InheritingBean has inherited the max−beans−in−cache property, that is the attribute on the ejbgen:entity tag, as well as the environment entry defined for the BaseEntityBean. If a tag (attribute) needs to be overriden, this can be done by simply defining the tag (attribute) in the InheritingBean, as is shown here for max−beans−in−cache:

```
 * @ejbgen:entity prim−key−class="java.lang.Integer"
 *   ejb−name = "InheritingBean"
 *   table−name = "InheritingBean"
 *   abstract−schema−name = "InheritingBean"
 *   max−beans−in−cache="235"
```

Resolving Inheritance Ambiguity

If you inherit tags that can appear more than once in the definition of an EJB, you must specify the id attribute on this tag to resolve ambiguity. Common examples of such tags are @ejbgen:ejb−ref, @ejbgen:env−entry, and @ejbgen:relation (also see the Related Topics section below).

For instance, let's redefine the BaseEntityBean used in the sample above to include two environment entries:

```
/**
 * @ejbgen:env−entry id="CompID" value="MyCompany" type="java.lang.String" name="CompanyName"
 * @ejbgen:env−entry id="DeptID" value="MyDepartment" type="java.lang.String" name="DepartmentN
 *
 * @ejbgen:entity prim−key−class="java.lang.Integer" ejb−name="BaseEntity" max−beans−in−cache="
 *
 * @ejbgen:...
 *
 */
abstract public class BaseEntityBean
  extends GenericEntityBean
  implements EntityBean
{
   //Methods you want to be inherited
   ...
}
```

Notice that in addition to MyCompany, a second environment entry MyDepartment has been added. Also notice that both tags have their id attribute set. In the definition of InheritingBean, the id value is used to refer to the proper environment entry, the DepartmentName in the example, and override its value attribute:

```
/**
 * @ejbgen:entity prim−key−class="java.lang.Integer"
 *   ejb−name = "InheritingBean"
 *   table−name = "InheritingBean"
 *   abstract−schema−name = "InheritingBean"
 *
 * @ejbgen:env−entry id="DeptID" value="Engineering"
 *   ...
 */
public abstract class InheritingBean extends BaseEntityBean
{
   ...
}
```

Related Topics

@ejbgen:ejb−local−ref Annotation

@ejbgen:ejb−ref Annotation

@ejbgen:env−entry Annotation

@ejbgen:relation Annotation

@ejbgen:resource−env−ref Annotation

@ejbgen:resource−ref Annotation

EJBs and Transactions

This topic discusses how to set transaction parameters for session, entity, and message−driven beans. It includes the following sections:

- What are Transactions?
- Bean and Container Managed Transactions
- Transaction Attributes
- Transaction Isolation

What are Transactions?

A transaction is unit of work containing activities that need to be completed together. For instance, for an online travel site, you might have a session bean method bookTravel that takes a credit card, ensures its valid, charges the price of the ticket on the credit card, and books the ticket. This method invoke methods on other EBJs to help accomplish these steps. For instance, methods of Customer, CreditCard and AirlineTicket beans will be invoked during the execution of this method. If all these steps complete successfully, the transaction succeeds. If any of these steps fails, the entire transaction fails and any changes made by previous steps are rolled back. For instance, if the ticket price is charged on the credit card, but booking the ticket fails, the charge on the credit card must be undone.

If all activities that make up the transaction execute successfully, all changes that were made as part of the transaction are committed. If the transaction fails, these changes are rolled back.

Reliable transactions have the following properties:

- *Atomic*. For a transaction to succeed, all steps that are part of the same transaction must complete. If any of these steps fails, the transaction fails. For the bookTravel method this means that if one of the steps throws an exception, the entire transaction fails, meaning that any changes already made as part of this transaction must be rolled back.
- *Consistent.* The transaction must leave the system in a correct state. This means that for the bookTravel method all activities that make up a transaction of this kind must be included. For instance, booking a ticket but not charging the customer will cause an incorrect state. Consistency also includes database integrity. For instance, after the bookTravel method completes successfully, a customer and airline ticket record must exist with referential integrity between these records.
- *Isolated*. A transaction must execute as if there were no other transactions running concurrently. In other words, a transaction must run without being interfered by other transactions (or causing interference to other transactions). Transaction Isolation is discussed in more detail below.
- *Durable*. If a transaction commits, the data changes made must be stored such that they survive system failure, for instance by storing the data in a database. For the bookTravel method, customer and airline tickets are represented by entity beans and an entity relationship is defined between these beans.

Runtime and Checked Exceptions

An EJB method can throw a runtime exception or a checked expection, that is, an exception that does not extend a RuntimeException and must be caught in a try−catch block. Runtime exceptions will cause the transaction to fail and changes to be rolled back. Checked exceptions do not cause transaction failure. For more information on runtime and checked exceptions, and how these should be handled by the client application, see Handling EJB Exceptions.

Bean and Container Managed Transactions

Transactions by session and message−driven beans can be managed by the bean or by the EJB container. Using container−managed transactions, also known as declarative transaction management is strongly recommended. When using container−managed transactions, the EJB container handles the transaction following the transaction attributes set for individual EJB methods (more on this below). Bean−managed transactions, also known as explicit transaction management, require you to explicitly handle the transaction management in the code. To enable bean−managed transaction, you must set the transaction−type attribute of the ejbgen:session or ejbgen:message−driven annotation to *Bean*. For session beans, this attribute is located in the ***Deployment*** section of the ***Property Editor***. The details of explicit transaction management are beyond the scope of this documentation. Transactions of entity beans (including BMP entity beans) are always container−managed.

Transaction Attributes

When the bookTravel method is invoked by by a client application, a transaction starts. The methods of Customer, CreditCard and AirlineTicket beans that are invoked by the bookTravel method are in principle part of the same transaction. In other words, these beans are likely participating in the same transaction and are part of the transaction scope. *Transaction attributes* determine whether an EJB's method actually participates in the same transaction.

The following transaction attributes are defined for session and entity beans:

- *NotSupported*. This method does not support a transaction. If the method is called as part of a transaction, the method is not part of the transaction scope and the transaction is suspended until the method has been executed.
- *Supports*. If the method is called as part of a transaction, it becomes part of the transaction scope. If the method call is not part of a transaction, the method executes as is without changing the transaction specifics.
- *Required*. If the method is called as part of a transaction, it become part of the transaction scope. If the method call is not part of a transaction, the method is executed within its own transaction, propagating the transaction scope to all methods it in turn invokes.
- *RequiresNew*. The method always executes within its own transaction, propagating the transaction scope to all methods it in turn invokes. If the method is called as part of a transaction, this transaction is suspended while this method executes within its own transaction. If the latter transaction fails, it does not affect the suspended transaction.
- *Mandatory*. If the method is called as part of a transaction, it become part of the transaction scope. If the method call is not part of a transaction, it will throw a javax.transaction.TransactionRequiredException or javax.transaction.TransactionRequiredLocalException.
- *Never*. If the method is called as part of a transaction, it will throw a RemoteException (to remote clients) or a EJBException (to local clients). If the method call is not part of a transaction, the method executes as is without changing the transaction specifics.

Message−driven beans only allow the NotSupported and Required transaction attributes. When set to Required, the message−driven bean executes as part of its own transaction. In other words, if the transaction fails, changes that were made as part of the onMessage method are rolled back, but the occurrence of an exception has no direct effect on the EJB or client application sending the JMS message, as the sender and the message−driven bean are decoupled. For more information, see Processing JMS Messages.

You can set transaction attributes on the entire EJB or on individual methods of session and entity beans. To set the transaction attribute for all methods of the EJB, use the default−transaction attribute of the ejbgen:entity, ejbgen:session or ejbgen:message−driven annotation. For session and entity beans, this attribute is located in the ***General*** section of the ***Property Editor***. To override the default transaction attribute for an individual method of a session or entity bean, set the transaction−attribute attribute for the method via the ***Property Editor*** or in the source code on the method's ejbgen tag (ejbgen:local−method, ejbgen:local−home−method, ejbgen:remote−method, or ejbgen:remote−home−method).

Transaction Isolation

The transaction isolation level determines how isolated a transaction is from another. The various isolation levels are defined with respect to three isolation phenomena:

- Dirty Reads. A transaction reads data written by another transaction that has not been committed yet. Because this data is uncommitted, a transaction failure would roll back these read changes.
- Nonrepeatable Reads. A transaction rereads data it has previously read and finds that data has been modified by another committed transaction in the meantime.
- Phantom Reads. A transaction reexecutes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another committed transaction in the meantime.

The following transaction isolation levels are defined for session and entity beans:

- *TransactionReadUncommitted*. A transaction may read any data currently on a data page, regardless of whether or not the data has been committed. Dirty reads, nonrepeatable reads, and phantom reads are possible.
- *TransactionReadCommitted*. A transaction is not allowed to read uncommitted data. Dirty reads are not possible, but nonrepeatable reads and phantom reads are.
- *TransactionRepeatableRead*. Once a transaction has read a set of data, repeated reads of the same data return the same values, even if other transactions have subsequently modified the data. Dirty reads and nonrepeatable reads are not possible, but phantom reads are.
- *TransactionSerializable*. A transaction ensures that no other transaction can read or write to the data it accesses. Dirty reads, nonrepeatable reads and phantom reads are not possible.

You can set the transaction isolation level using the isolation−level attribute on the ejbgen:method−isolation−level−pattern, ejbgen:local−method, and ejbgen:remote−method tags. All the methods invoked within a transaction must have the same isolation level. You cannot mix isolation levels.

The transaction isolation level setting interacts with an entity bean's concurrency strategy as set with the ejbgen:entity concurrency−strategy attribute. Specifically, when the *Optimistic* or the *Exclusive* concurrency strategies are used, the transaction isolation level setting is ignored.

Oracle Database

Even with an isolation−level setting of *TransactionSerializable*, Oracle does not detect serialization problems until commit time. The error message returned is:

```
java.sql.SQLException: ORA−08177: can't serialize access for this transaction
```

WebLogic provides special isolation−level settings to prevent this:

- *TransactionReadCommittedForUpdate*. When set, every SELECT query from that point on will have *ForUpdate* added to acquired locks on the selected rows. Consequently, if Oracle cannot lock the rows affected by the query immediately, then it waits until the rows are free. This condition remains in effect until the transaction succeeds or fails.
- *TransactionReadCommittedForUpdateNoWait*. When set, every SELECT query from that point on will have *ForUpdateNoWait* added to acquire locks on the selected rows. Consequently, if Oracle cannot lock the rows affected by the query immediately, then Oracle terminates the query before completion. This condition remains in effect until the transaction succeeds or fails.

Related Topics

@ejbgen:session Annotation

@ejbgen:entity Annotation

@ejbgen:message−driven Annotation

@ejbgen:local−method Annotation

@ejbgen:local−home−method Annotation

@ejbgen:remote−method Annotation

@ejbgen:remote−home−method Annotation

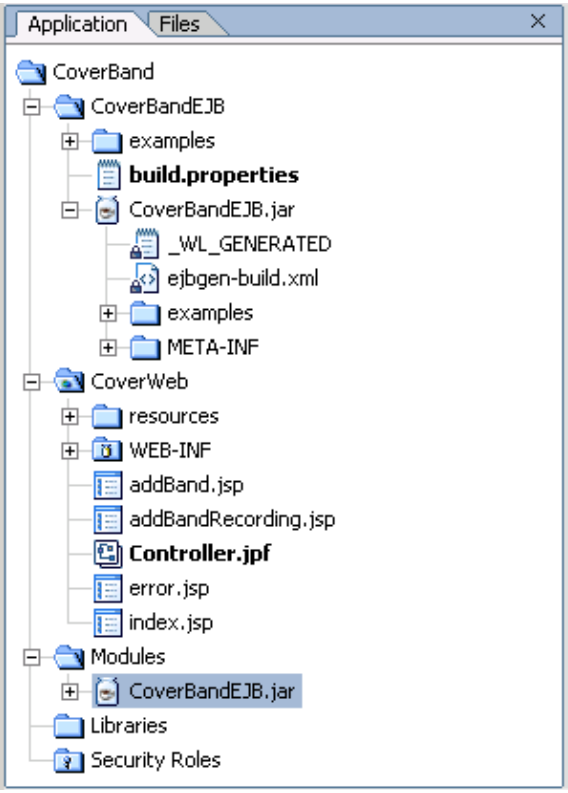
@ejbgen:method−isolation−level−pattern Annotation

Summary of IDE Windows for EJB Project

An EJB project populates several panels in the WebLogic Workshop IDE with controls that you use to work on the project. This topic introduces the various tabs and windows to users who are new to working with WebLogic Workshop and EJB project.

Application Tab

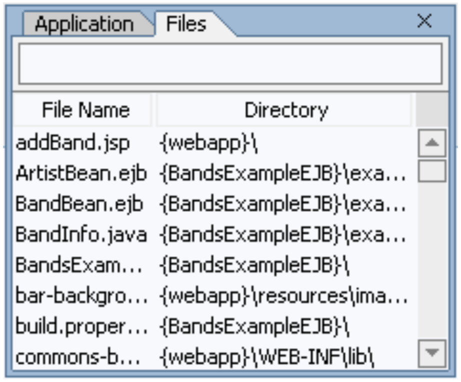
The Application tab is a tree view of all the files in the application, organized by project. For example, in the below picture the application *CoverBand* contains two projects: CoverBandEJB, an EJB project, and CoverWeb, a Web project. The EJB project contains all the EJB−related files in the application.



- CoverBand
  - CoverBandEJB
    - examples
    - **build.properties**
    - CoverBandEJB.jar
      - _WL_GENERATED
      - ejbgen-build.xml
      - examples
      - META-INF
  - CoverWeb
    - resources
    - WEB-INF
    - addBand.jsp
    - addBandRecording.jsp
    - **Controller.jpf**
    - error.jsp
    - index.jsp
  - Modules
    - CoverBandEJB.jar
  - Libraries
  - Security Roles

Files Tab

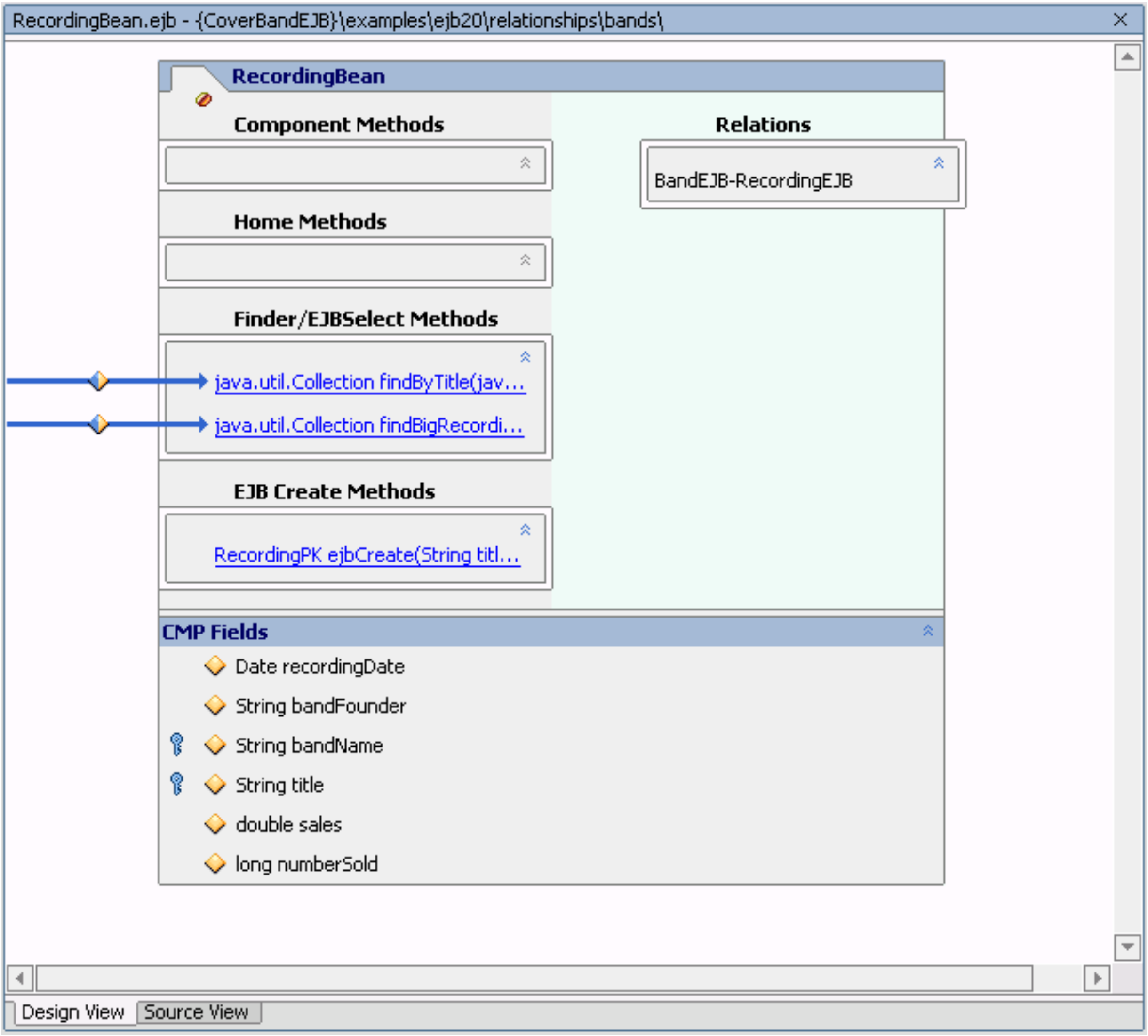
The Files tab, in the same window as the Application tab, shows the files that make up the application.



File Name	Directory
addBand.jsp	{webapp}\
ArtistBean.ejb	{BandsExampleEJB}\exa…
BandBean.ejb	{BandsExampleEJB}\exa…
BandInfo.java	{BandsExampleEJB}\exa…
BandsExam…	{BandsExampleEJB}\
bar-backgro…	{webapp}\resources\ima…
build.proper…	{BandsExampleEJB}\
commons-b…	{webapp}\WEB-INF\lib\

Design View

The Design view lets you select any component of an EJB for editing. Right−click in an empty area to bring up the shortcut menu from which you create new methods, relations, and CMP fields. The diamonds on the CMP fields and various methods show in what interface(s) these are defined. Right−click the arrows/CMP field names to change an interface definition.



BandEJB-RecordingEJB

java.util.Collection findByTitle(jav…

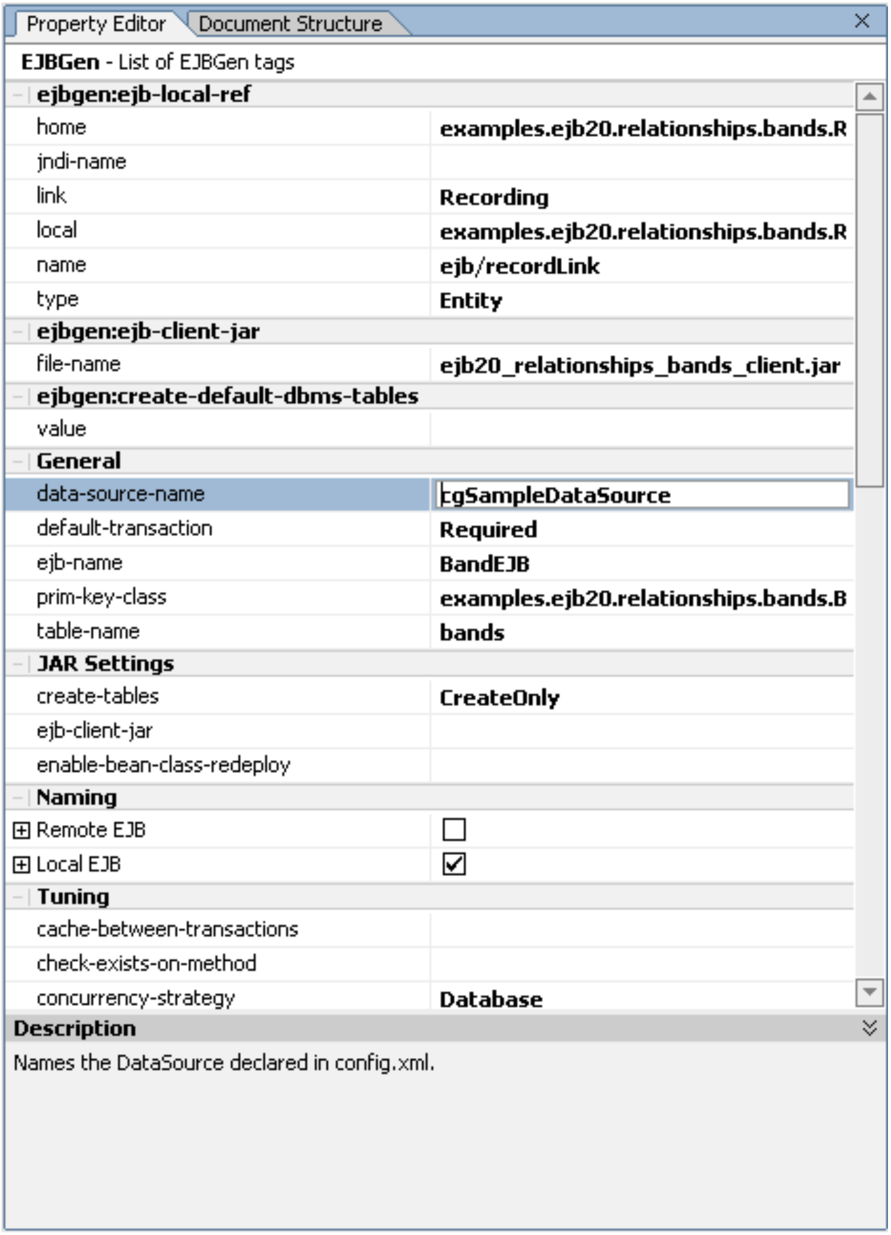
java.util.Collection findBigRecordi…

RecordingPK ejbCreate(String titl…

- Date recordingDate
- String bandFounder
- String bandName
- String title
- double sales
- long numberSold

Property Editor

Use the property editor to enter ejbgen tag annotations. When you select an ejbgen tag, a description appears at the lower end of the editor.



EJBGen - List of EJBGen tags

ejbgen:ejb-local-ref	
home	**examples.ejb20.relationships.bands.R**
jndi-name	
link	**Recording**
local	**examples.ejb20.relationships.bands.R**
name	**ejb/recordLink**
type	**Entity**
ejbgen:ejb-client-jar	
file-name	**ejb20_relationships_bands_client.jar**
ejbgen:create-default-dbms-tables	
value	
General	
data-source-name	**EgSampleDataSource**
default-transaction	**Required**
ejb-name	**BandEJB**
prim-key-class	**examples.ejb20.relationships.bands.B**
table-name	**bands**
JAR Settings	
create-tables	**CreateOnly**
ejb-client-jar	
enable-bean-class-redeploy	
Naming	
⊞ Remote EJB	☐
⊞ Local EJB	☑
Tuning	
cache-between-transactions	
check-exists-on-method	
concurrency-strategy	**Database**

Description

Names the DataSource declared in config.xml.

For more information about ejbgen tags, see the Enterprise JavaBean Annotation Reference.

Build, Debug Output, and Breakpoints Windows

The ***Build*** window logs the results of the build. If the build window is not shown, choose ***Views*****−−>*****Windows*****−−>*****Build***. The ***Debug Output*** and ***Breakpoints*** windows can be used when (test) running the EJB to examine debugging output and, if applicable, breakpoints.

Related Topics

How Do I: Create a New Application?

How Do I: Create a New Project?

How Do I: Start or Stop WebLogic Server?

Debugging Your Application