Developing Enterprise JavaBeans

These topics show how to develop Enterprise JavaBeans. BEA Workshop for WebLogic Platform 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.

Current Release Information:

- What's New in 9.2
- Upgrading from 8.1

Useful Links:

- Tutorials
- Tips and Tricks

Other Resources:

- Online Docs
- Dev2Dev
- Discussion Forums
- Development Blogs

Topics Included in This Section

Tutorial: Enterprise JavaBeans

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

Enterprise JavaBeans in Workshop for WebLogic

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

Developing Entity Beans

This topic discusses the development of entity beans.

Developing Session Beans

This topic discusses the development of session beans.

Developing Message-Driven Beans

This topic discusses the development of message-driven beans.

EJB Properties Dialog

This topic explains how to set project-level EJB properties.

Enterprise JavaBeans in Workshop for WebLogic

In Workshop for WebLogic, the WebLogic EJB project greatly eases development of Enterprise JavaBeans. This topic provides an overview of EJBs and the WebLogic EJB project in platform applications. It includes the following sections:

> **Note:** Workshop for WebLogic provides tool support for EJB-related technologies available with WebLogic Server. For a few of developing Enterprise JavaBeans outside the context of Workshop for WebLogic, see Programming WebLogic Enterprise JavaBeans.

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 favorite J2EE documentation and http://java.sun.com.

In Java EE 5 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.

You'll find more information on developing session beans in Developing Session Beans.

Note: For a few of developing session Enterprise JavaBeans outside the context of Workshop for WebLogic, see Session EJBs.

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.

Developing Entity Beans provides information on how Workshop for WebLogic supports this component type.

Note: For a few of developing session Enterprise JavaBeans outside the context of Workshop for WebLogic, see Entity EJBs.

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.

For more information, see Developing Message-Driven Beans.

Note: For a few of developing session Enterprise JavaBeans outside the context of Workshop for WebLogic, see Message-Driven EJBs.

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). When you use annotations to develop EJBs, these interfaces are generated for you by the IDE. Attribute values in the annotations specify the names for the generated interfaces.

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 Workshop for WebLogic application, as well as interactions between EJBs and web services or page flows in the same Workshop for WebLogic 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 Workshop for WebLogic 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. Workshop for WebLogic also provides JMS controls to work with a Java Message Service.

Deployment Descriptor

During run time, 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.

> **Note:** The deployment descriptor should never be checked in to a source control system. The descriptor is repeatedly updated by Workshop for WebLogic during development.

Annotation attribute values in your EJB source code specify the values to be used in descriptors, which are then automatically generated by the IDE.

> **Note:** For more on deployment descriptors, see EJB Deployment Descriptors.

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 the WebLogic EJB project?

The WebLogic EJB project is the development environment for session, entity, and message-driven beans in a Workshop for WebLogic application. A WebLogic EJB project contains one or more EJBs. A Workshop for WebLogic application can have one or more WebLogic EJB projects, as well as other types of projects such as web service projects and web application projects.

When you create a WebLogic EJB project and choose to associate it with an a WebLogic EAR project, a dependency is automatically created from all WebLogic Web projects currently in the EAR to the new EJB project (the creation of these dependencies can be disabled). To your WebLogic EJB project you can add EJB source code by using any of three source code templates:

WebLogic Entity Bean, WebLogic Session Bean, and WebLogic Message Driven Bean.

As with other source artifacts, you can use the Annotation view to edit annotation attribute values that determine how EJB interfaces are generated, what goes into deployment descriptors, and so on.

A key advantage of developing EJBs in the WebLogic 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, you can use one file to represent an EJB. This is accomplished by using EJBGen annotations.

> **Note:** When developing Enterprise JavaBeans with Workshop for WebLogic be sure to use the WebLogic EJB Project project type, rather than the EJB Project also provided. The WebLogic EJB Project includes WebLogic EJB Extensions provided with the IDE.

WebLogic EJB Project and EJBGen Annotations

When developing EJBs in Workshop for WebLogic, you use EJBGen annotations in the EJB source file. These annotations are used to mark methods to be exposed when generating remote and home interfaces and to specify deployment descriptor settings. In the IDE, you can find generated deployment descriptors will be under src/META-INF; generated interfaces are located under . apt_src.

You can edit these annotations directly or use the Annotations view. You might find that using the Annotations view, which is aware of the constant values that some annotation attributes require, makes things easier.

For more about EJBGen annotations, see the EJBGen Reference.

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. In addition, when you add, change, or remove EJB source files, Workshop for WebLogic runs the EJBGen tool to regenerate interfaces and deployment descriptors.

If the WebLogic EJB project is associated with a WebLogic Server server, the EJB module will be deployed to the server (either as a standalone EJB module or a the child of an EAR); if changes were made to the project since the last publish, ejbc is executed to generate and compile WebLogic EJB container classes for the EJBs.

What are EJB Controls?

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.

See Related Topics

Applications and Projects

Tutorial: Building Enterprise JavaBeans

Tutorial: Building Enterprise JavaBeans

This tutorial introduces you to the basics of building Enterprise JavaBeans with BEA Workshop for WebLogic Platform (Workshop for WebLogic). Through the tutorial you'll build a very simple application that includes a session bean, an entity bean, and a Java Page Flow (as a client).

> **Note:** This tutorial requests that you create a new workspace; if you already have a workspace open, this will restart the IDE. Before beginning, you might want to launch help in standalone mode to avoid an interruption the restart could cause, then locate this topic in the new browser. See Using Help in a Standalone Mode for more information.

Before You Begin

In general, this tutorial assumes you already know something about Enterprise JavaBeans and how they fit into applications built on J2EE. The Enterprise JavaBeans subject is large enough to fill — indeed, it has filled — entire books, so it's not described in detail here.

However, if you happen not to be familiar with Enterprise JavaBeans, this tutorial includes brief notes along the way about, for example, what entity and session beans are. But it's just enough, really, to keep things moving along. For further reading, you might get started with the following:

- Sun Microsystems' Enterprise JavaBeans Technology home at the Sun web site
- BEA System's WebLogic Server documentation on Programming WebLogic Enterprise JavaBeans on eDocs
- BEA Workshop for WebLogic Platform's documentation Developing Enterprise JavaBeans

Focus of this Tutorial

This tutorial introduces you to Enterprise JavaBeans development with Workshop for WebLogic. As you work through this tutorial, you will:

- Learn about three types of Workshop for WebLogic projects, including the WebLogic EAR project, WebLogic EJB project, and WebLogic web project.
- Create two types of Enterprise JavaBeans: an entity bean that stores visit to a web page and a session bean that models tracking and greeting visitors.
- Learn how to use EJBGen annotations to speed EJB development.
- Learn how to use an EJB control to ease client development.
- Test your EJBs using a web project as a test client.

Application Overview

The visitor tracking application you'll build with this tutorial includes the following components:

- A `VisitBean` entity bean that keeps track of visits to the test web page.
- A `VisitTrackerBean` session bean that greets visitors and checks the entity bean for the number of visits made to the web page.
- A `VisitWebTest` web project within which to build a page flow that will be a test client for `VisitBean` and `VisitTrackerBean`.
- A `VisitTrackerBeanCtrl` EJB control that will be the test page flow's means for calling methods of the `VisitTrackerBean` session bean.

Steps in this Tutorial

Create a Workspace for Development
: You'll use Workshop for WebLogic to create a workspace, adding projects that will contain your component code.

Create the VisitBean Entity Bean
: Here, you'll write code for the EJB in your application that will represent a visit.

Create the VisitTrackerBean Session Bean
: The session bean you'll create here will be responsible for counting visits.

Create and Start a Server on Which to Test
: Specifying a WebLogic Server on which to test is easy; you'll do that in this step.

Create a Test Project
: You'll test the EJBs you've built with a page flow; here, you'll create a web project to get started.

Write Test Source Code and Test the EJBs
: At last, you'll try out your code using an EJB control in a page flow.

Related Topics

Applications and Projects

Click the following arrow to navigate through the tutorial:

Step 1: Create a Workspace for Development

In this step you'll get your workspace set up, then create two projects to get your EJB source code going. You'll start writing EJB source code in the next step.

In this section, you will:

- Start Workshop for WebLogic
- Create a workspace
- Create an EAR project
- Create an EJB project

To Start Workshop for WebLogic

If you haven't started Workshop for WebLogic yet, use these steps to do so.

... on Microsoft Windows

If you are using a Windows operating system, follow these instructions.

- From the **Start** menu, click **BEA Products > Workshop for WebLogic Platform 9.2**

...on Linux

If you are using a Linux operating system, follow these instructions.

- Run BEA_HOME/workshop92/workshop4WP/workshop4WP.sh

To Create a Workspace

You use a workspace to contain related source code. This one will end up containing both your EJB source and the source you'll test the EJB with.

1. If the **Workshop Launcher** dialog is not displayed, select **File > Switch Workspace**. Otherwise, skip to the next step.
2. In the **Workspace Launcher** dialog, click **Browse**, then browse to the directory that you want to contain your new workspace directory.

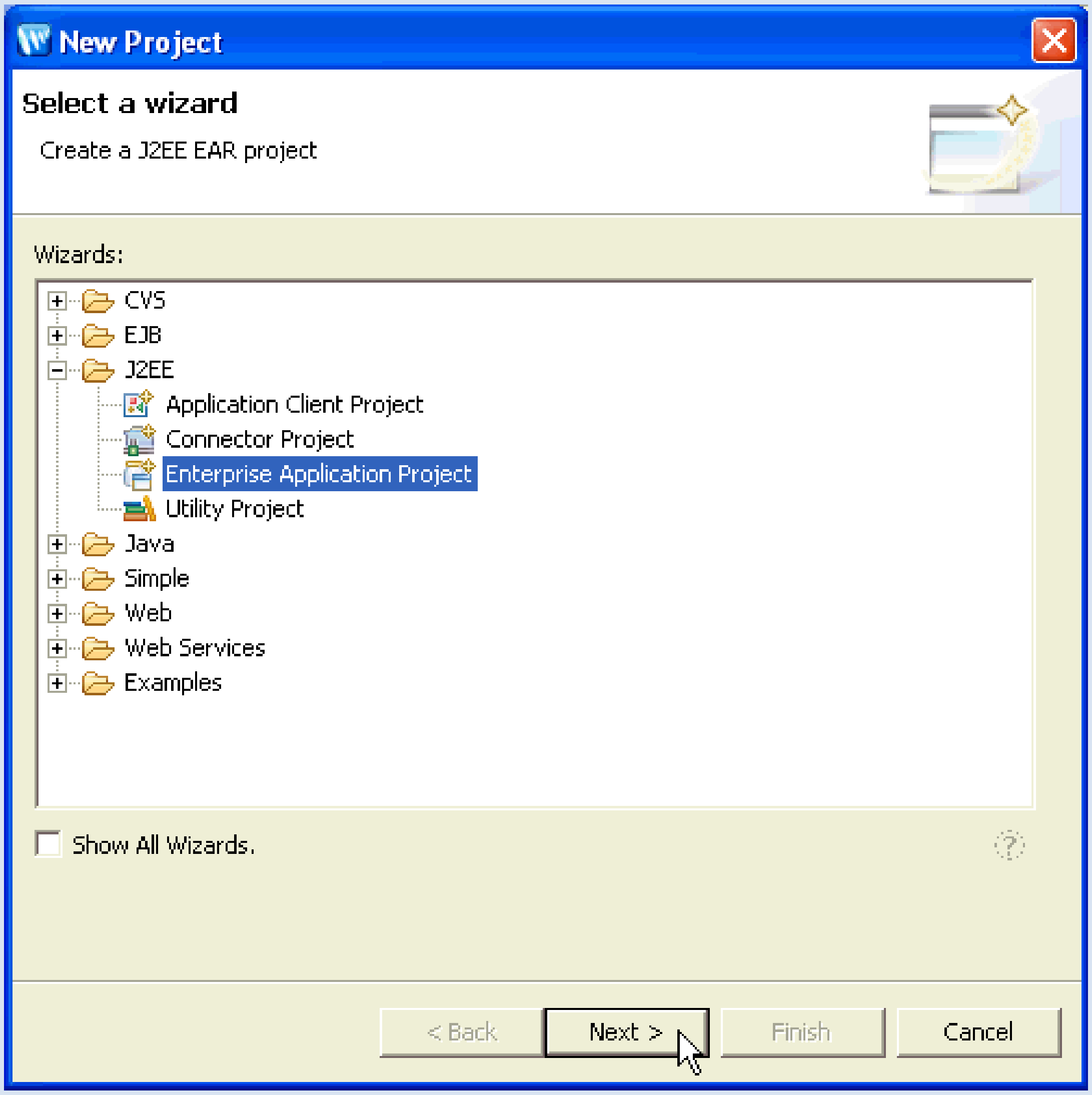
   This can be any directory. You'll be creating a new directory *inside* this one for your workspace.
3. When you have a directory selected, click **Make New Folder**. Name the new folder `EJBTutorial` press **Enter** to create the folder, then click **OK**.
4. In the **Workspace Launcher**, click **OK**.

Workshop for WebLogic will create a new empty workspace in the folder you created, then refresh to display the workspace. Note that the Navigator view is empty.

To Create an EAR Project

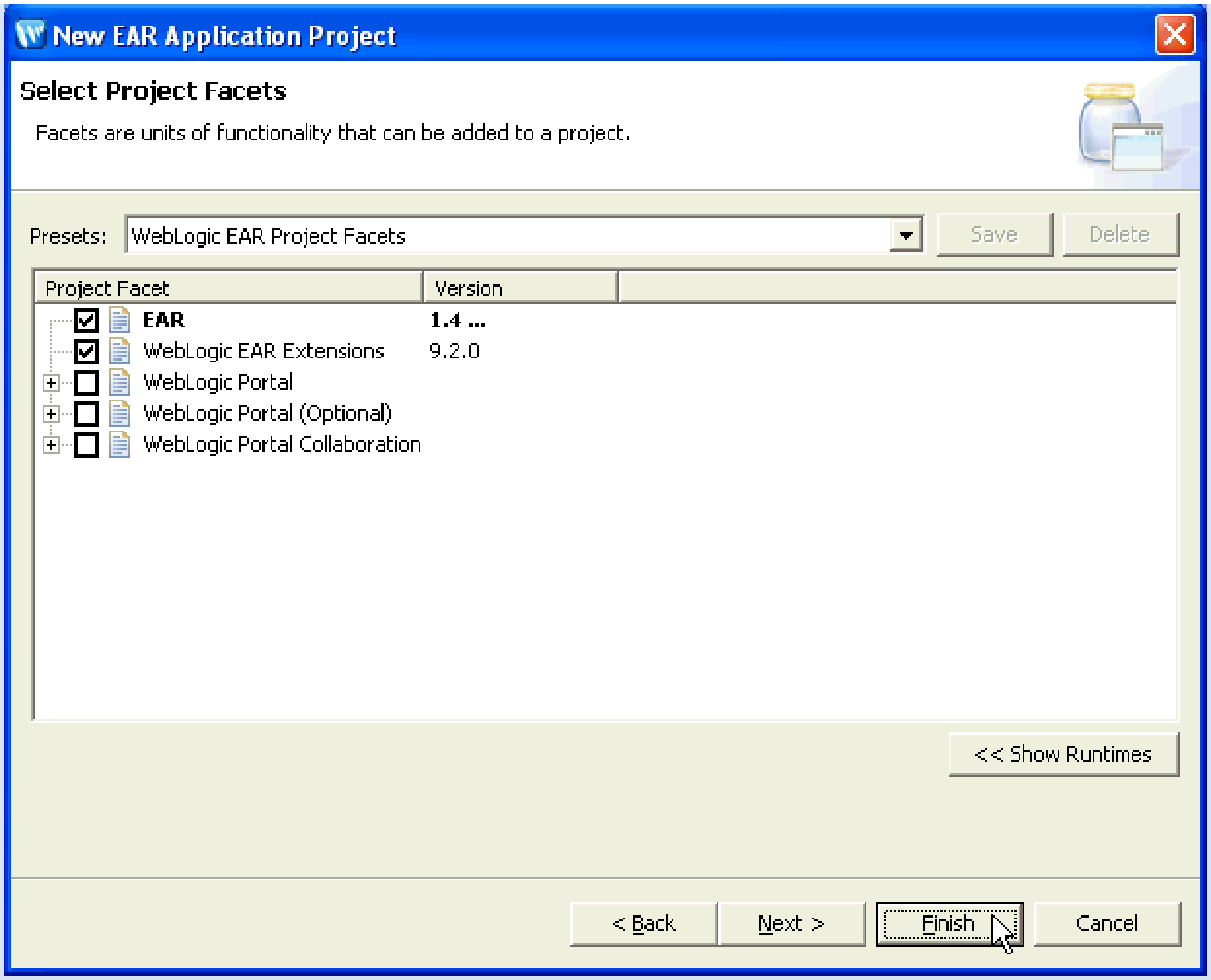
An EAR project is a special kind of project that represents an enterprise application. When you "add" a project to an EAR project, you're actually adding a project reference. Projects added to the EAR project in this way will end up as part of a single **E**nterprise **AR**chive (EAR) file, which can be deployed to WebLogic Server. An EAR project is also a way to designate libraries that are to be shared across projects.

1. Click **File > New > Project.**
2. In the **New Project** dialog, expand **J2EE**, select **Enterprise Application Project**, then click **Next**.



3. In the **New EAR Application Project** dialog, in the **Project name** box, enter `EJBTutorial_EARProject`, then click **Next**.
4. Under **Select Project Facets**, leave the **Presets** dropdown as it is. It should say **EAR**. Selecting this preset group of facets ensures that you'll have the JARs you need to support your EAR project.

   Click **Finish**.


New EAR Application Project
Select Project Facets
Facets are units of functionality that can be added to a project.
Presets: WebLogic EAR Project Facets
Save
Delete
Project Facet
Version
EAR 1.4 ...
WebLogic EAR Extensions 9.2.0
WebLogic Portal
WebLogic Portal (Optional)
WebLogic Portal Collaboration
<< Show Runtimes
< Back
Next >
Finish
Cancel

When Workshop for WebLogic has finished creating your EAR project, you'll see that the new project has been marked with an error (indicated by the red X mark in the lower left-hand corner of the EAR project folder). That's as it should be — an EAR project *must* contain references to other projects. You'll remedy it in just a moment.

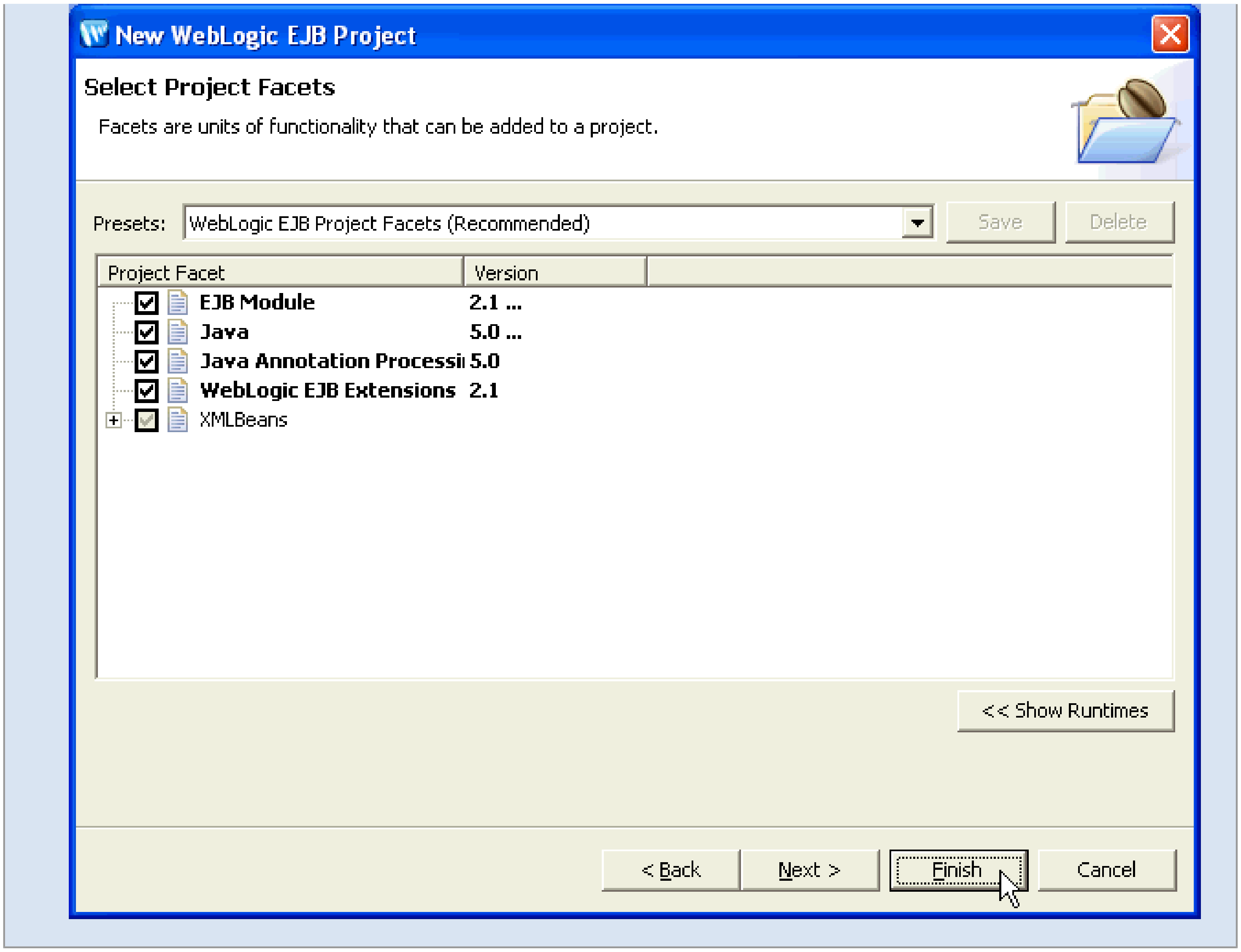
To Create an EJB Project

Now that you've got an EAR project, you can create an EJB project within which to build your EJBs. The EJB project will contain your EJB source code.

1. Click **File > New > Project.**
2. In the **New Project** dialog, expand **EJB**, click **WebLogic EJB Project**, then click **Next**.
3. In the **New EJB Project** dialog, in the **Project name** box, enter `VisitEJBProject`.
4. Place a check next to **Add project to EAR** and select **EJBTutorial_EARProject** from the dropdown menu. (This is the EAR project you created earlier.)

   Click **Next**.
5. Under **Select Project Facets**, note that you're provided support for creating EJB source code that includes annotations for faster development.

   Click **Finish**.



Presets: WebLogic EJB Project Facets (Recommended)

Project Facet	Version
☑ EJB Module	2.1 ...
☑ Java	5.0 ...
☑ Java Annotation Processi	5.0
☑ WebLogic EJB Extensions	2.1
☑ XMLBeans	

You should now see both your EAR project and EJB project in the Navigator view. Also, after you've created the EJB project, you'll see that the error flag on the EAR project has gone away.

Note that the new project you created, VisitEJBProject, has an error flag next to it. The error flag is present because no EJB code exists yet in the project. In the next step you will remove this error flag by adding EJB code to the project.

Related Topics

Enterprise JavaBeans in Workshop for WebLogic

Click one of the following arrows to navigate through the tutorial:

Step 2: Create the VisitBean Entity Bean

In this step you'll create the source code for your entity bean. As described in the WebLogic Server topic, How Do Applications Use EJBs?, an entity bean represents a set of persistent data. In the case of this application, the data is stored in a database. An entity bean representing database data will typically represent a single row in the database.

You can also think of entity beans as representing *things* your application interacts with. In this case, the thing is a visit.

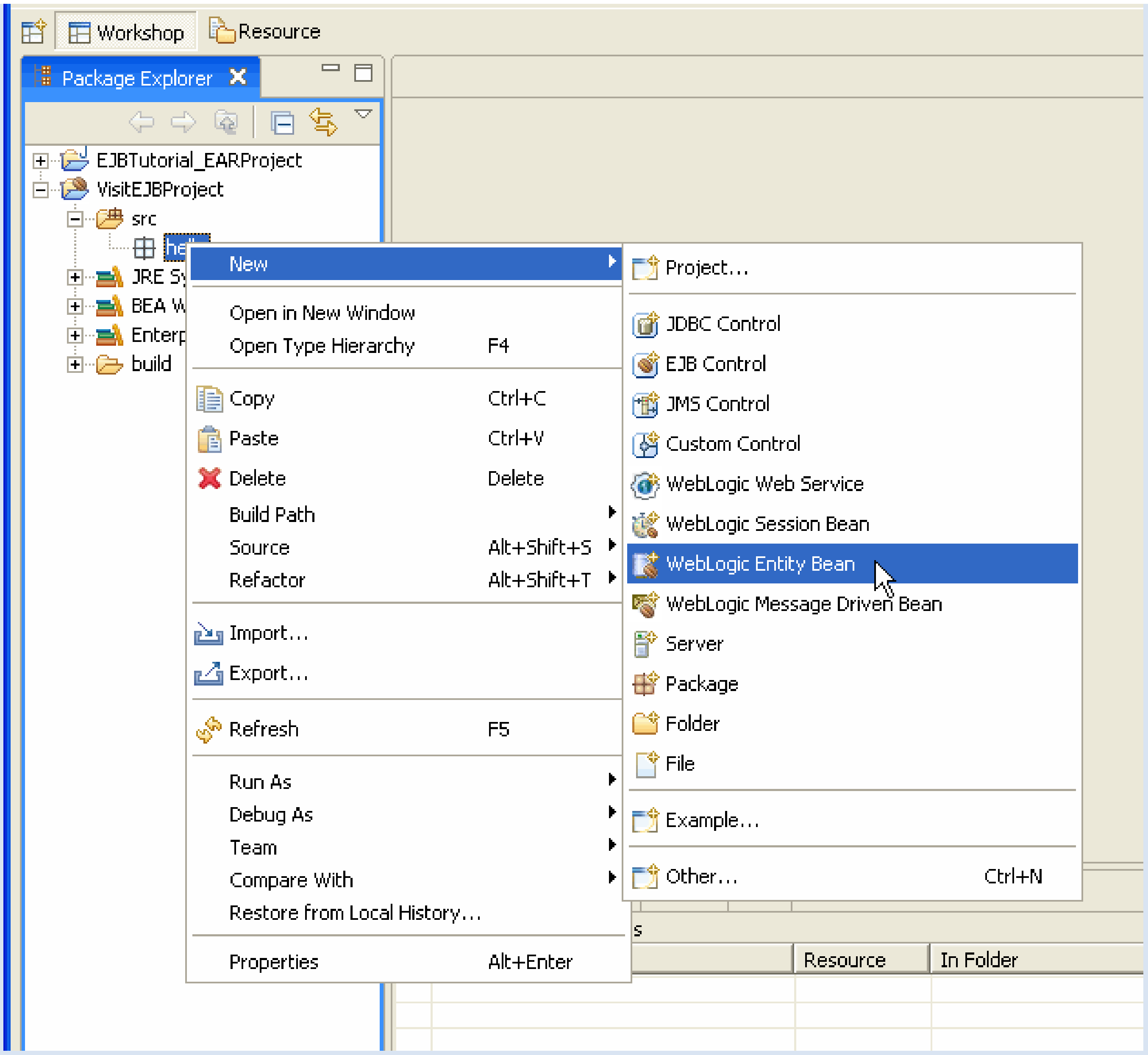
In this section, you will:

- Create VisitBean source files
- Set useful annotation values
- Define CMP fields for entity data
- Add ejbCreate and ejbPostCreate methods

To Create VisitBean Source Files

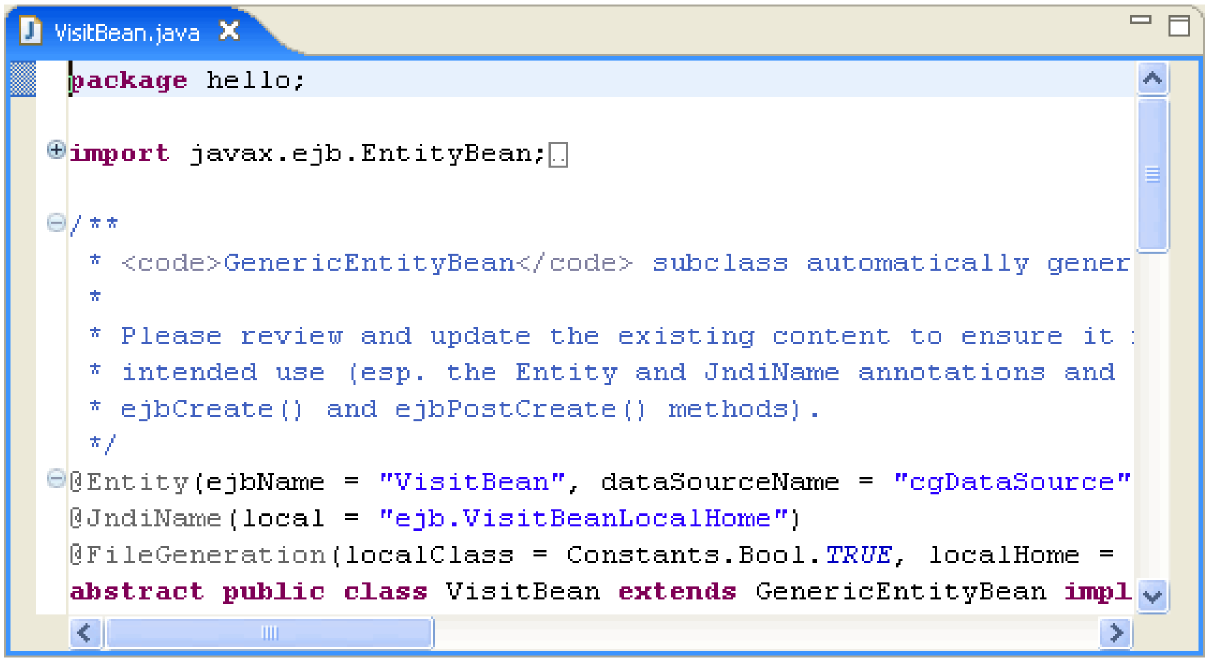
Here, you'll create the package and source file to contain the code for your entity bean.

1. In the **Package Explorer**, expand **VisitEJBProject**.
2. Right-click **src** and select **New** > **Package**.
3. In the **New Java Package** dialog, in the **Name** box, enter `hello`, then click **Finish**.
4. In the **Package Explorer**, right-click the **hello** package you just created and select **New** > **WebLogic Entity Bean** .



5. In the **File name** box, enter `VisitBean`, then click **Finish**.

In the generated source code, you'll see a few annotations — code that begins with an @ sign.



VisitBean.java

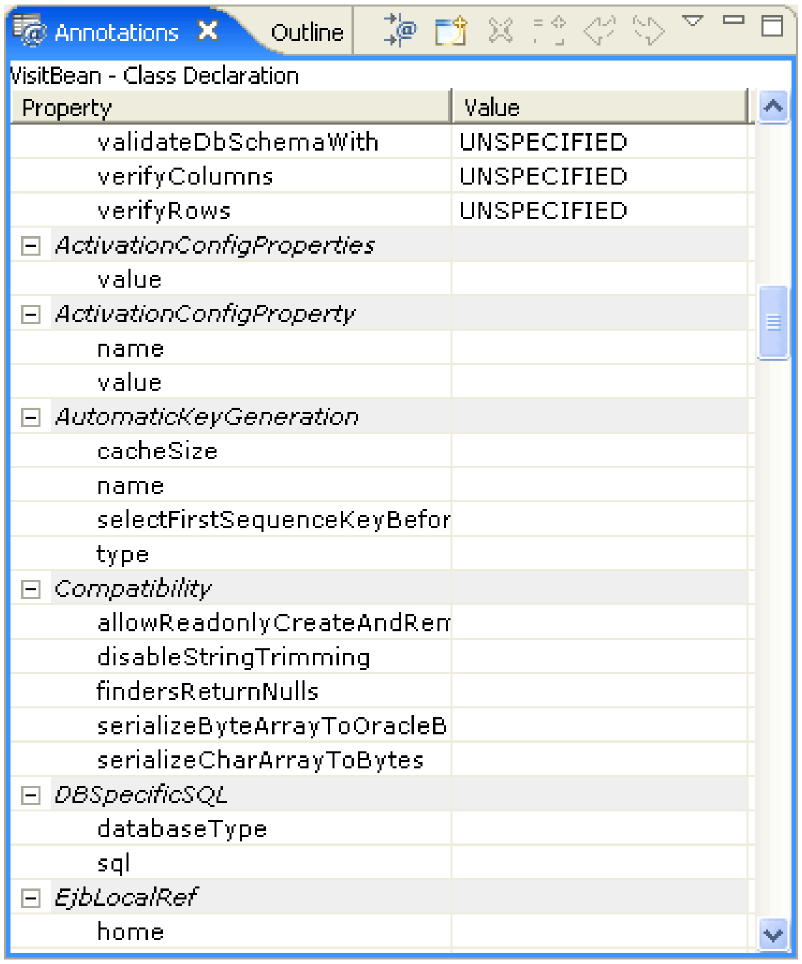
```
package hello;

import javax.ejb.EntityBean;

/**
 * <code>GenericEntityBean</code> subclass automatically gener
 *
 * Please review and update the existing content to ensure it
 * intended use (esp. the Entity and JndiName annotations and
 * ejbCreate() and ejbPostCreate() methods).
 */
@Entity(ejbName = "VisitBean", dataSourceName = "cgDataSource"
@JndiName(local = "ejb.VisitBeanLocalHome")
@FileGeneration(localClass = Constants.Bool.TRUE, localHome =
abstract public class VisitBean extends GenericEntityBean impl
```

These annotations are used at compile time by the WebLogic Server EJBGen tool to generate Remote and Home classes, as well as the deployment descriptor for the entity bean. All of the generated files are needed for developing entity beans, but the annotations you'll add remove the need for you to create the files yourself — you only need to create one.

> **Note:** For general information about the classes and interfaces that make up Enterprise JavaBeans, see EJB Anatomy and Environment in the WebLogic Server documentation on eDocs. For other WebLogic-specific information, see Create EJB Classes and Interfaces, also on eDocs.

Throughout code you write in Workshop for WebLogic, you can use annotations as an efficient way to automate these and other development tasks. There are many annotations and attributes that can be applied to an entity bean class such as `VisitBean`. To see all of the applicable annotations, use the Annotations view.

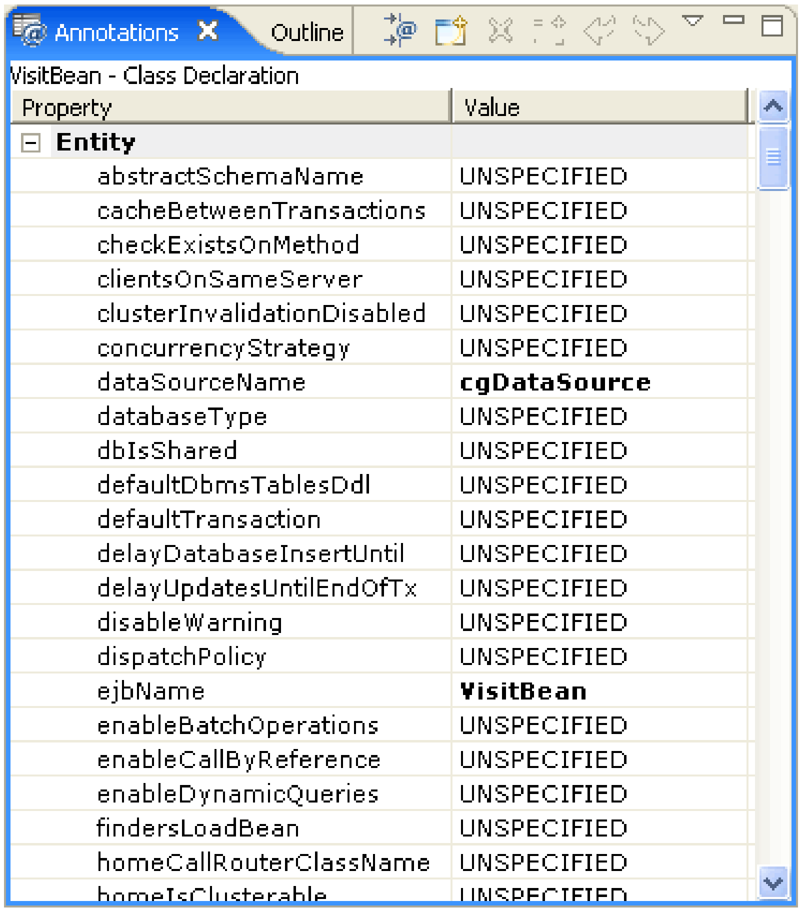
By default in the Workshop perspective, the Annotations view should be visible at the right side of the IDE. With the Annotations view docked at the right side of the IDE and your cursor in the source code on the `VisitBean` class name, you should see something like the following illustration:



VisitBean - Class Declaration

Property	Value
validateDbSchemaWith	UNSPECIFIED
verifyColumns	UNSPECIFIED
verifyRows	UNSPECIFIED
ActivationConfigProperties	
value	
ActivationConfigProperty	
name	
value	
AutomaticKeyGeneration	
cacheSize	
name	
selectFirstSequenceKeyBefor	
type	
Compatibility	
allowReadonlyCreateAndRem	
disableStringTrimming	
findersReturnNulls	
serializeByteArrayToOracleB	
serializeCharArrayToBytes	
DBSpecificSQL	
databaseType	
sql	
EjbLocalRef	
home	

The annotations listed here apply to the class, but most are set to their default values. When you edit annotations or attributes to non-default values, the annotations and attributes are written into the source code, as shown in the source. You can scroll in the Annotations view to find the annotations whose attributes have been set to default values. They're shown in bold, as in the following illustration:



VisitBean - Class Declaration

Property	Value
Entity	
abstractSchemaName	UNSPECIFIED
cacheBetweenTransactions	UNSPECIFIED
checkExistsOnMethod	UNSPECIFIED
clientsOnSameServer	UNSPECIFIED
clusterInvalidationDisabled	UNSPECIFIED
concurrencyStrategy	UNSPECIFIED
dataSourceName	**cgDataSource**
databaseType	UNSPECIFIED
dbIsShared	UNSPECIFIED
defaultDbmsTablesDdl	UNSPECIFIED
defaultTransaction	UNSPECIFIED
delayDatabaseInsertUntil	UNSPECIFIED
delayUpdatesUntilEndOfTx	UNSPECIFIED
disableWarning	UNSPECIFIED
dispatchPolicy	UNSPECIFIED
ejbName	**VisitBean**
enableBatchOperations	UNSPECIFIED
enableCallByReference	UNSPECIFIED
enableDynamicQueries	UNSPECIFIED
findersLoadBean	UNSPECIFIED
homeCallRouterClassName	UNSPECIFIED
homeIsClusterable	UNSPECIFIED

The annotations with default values include:

- `@Entity` — Specifies values for properties scoped to the entity bean class. Many of this annotation's attributes correspond to elements in an EJB deployment descriptor; the descriptor is generated in part from annotation values. The attributes shown here in code include:

Attribute	Description
`ejbName`	Descriptive name of the entity bean.
`dataSourceName`	Data source that holds the database table(s) the entity bean uses.
`primKeyClass`	Name of the Java class of the primary key. In case of a compound primary key, this class will be generated by EJBGen.
`tableName`	Table to which this entity bean is mapped.

- `@JndiName` — Specifies the local or remote JNDI names of an EJB; that is, the JNDI name associated with its local or remote interface. The attributes shown here in code include:

Attribute	Description
`local`	Local JNDI name of the EJB.

- `@FileGeneration` — Specifies the interface, compound primary key, and value classes that are to be auto-generated during build.

Attribute	Description
`localClass`	Specifies whether to generate the local interface for this EJB. The default value is TRUE.
`localHome`	Specifies whether to generate the local home interface for this EJB. The default value is TRUE.
`remoteClass`	Specifies whether to generate the remote interface for this EJB. The default value is FALSE.
`remoteHome`	Specifies whether to generate the remote home interface for this EJB. The default value is FALSE.
`valueClass`	Specifies whether to generate the value class for this EJB. The default value is TRUE

Note: If you're experienced with EJBGen annotations in platform versions prior to 9.0, you will notice that the syntax has changed. As of version 9.0, EJBGen uses Java 5 annotations based on JSR 175, and supported in JDK version 1.5.

To Set Annotation Values

The annotations provided here are a starting place for your development work. You'll need to edit some of the values so that they are useful for your specific EJB. You can edit annotation values directly in source code or in the Annotations view. Procedures in this tutorial will usually use the Annotations view so that you can become acquainted with it, but these topics will also show the updated annotation code.

1. In the the source code, put your cursor in the `@Entity` annotation.

   Notice that Annotations view displays values for this annotation only.

2. In the **Annotations** view, ensure that the **Entity** property is expanded, then locate the **tableName** attribute.
3. Change the **tableName** attribute value to `EJB_Visits`.
4. Locate the **primKeyClass** attribute. Change the value to `java.lang.String`.

   This change is in preparation for the method changes you will make below. The updated annotation code should appear something like this:

   ```
   @Entity(ejbName = "VisitBean", dataSourceName = "cgDataSource",
       tableName = "EJB_Visits", primKeyClass = "java.lang.String")
   ```

To Define CMP Fields for Entity Data

Now you'll add code that defines two container-managed persistence (CMP) fields. CMP fields are "virtual" fields represented by columns in a data source, rather than variable definitions in this class. Using the accessors you'll add, `VisitBean` will get its data from the cgDataSource specified above in the `dataSourceName` attribute. The annotations specify the details of the entity bean's correspondence to the data source row.

1. Delete the methods getKey() and setKey(String key) from the class.

   ~~/**~~
   ~~* IMPORTANT: Automatically generated primary key field getter method.~~
   ~~* Please change name and class as appropriate.~~
   ~~*/~~
   ~~@CmpField(column = "key", primkeyField = Constants.Bool.TRUE)~~
   ~~@LocalMethod()~~
   ~~public abstract java.lang.String getKey();~~

   ~~/**~~
   ~~* IMPORTANT: Automatically generated primary key field setter method.~~
   ~~* Please change name and class as appropriate.~~

```
             */
            @LocalMethod()
            public abstract void setKey(java.lang.String key);
```

2. Add the following methods to the body of the VisitBean class to define two container-managed persistence (CMP) fields.

```
            @CmpField(column = "visitorName", primkeyField = Constants.Bool.TRUE)
            @LocalMethod()
            public abstract java.lang.String getVisitorName();

            @LocalMethod()
            public abstract void setVisitorName(java.lang.String visitorName);

            @CmpField(column = "visitNumber")
            @LocalMethod()
            public abstract int getVisitNumber();

            @LocalMethod()
            public abstract void setVisitNumber(int number);
```

After you delete the setKey method, you'll have an error where the code calls it elsewhere. You'll fix that in a moment.

Note: If you're copying and pasting code into your source window, you can format it in Workshop for WebLogic by right-clicking and selecting **Source > Format**.

This code includes the following annotations:

- `@CmpField` — Defines a virtual CMP field.

Attribute	Description
column	Column in a database table to which this CMP field will be mapped.
primKeyField	Whether this field is the primary key field, or part of the compound primary key.

- `@LocalMethod` — Defines a method for an entity or session bean's local (business) interface. No attribute values need to be set for this annotation.

To Add ejbCreate and ejbPostCreate Methods

Now you'll add `ejbCreate` and `ejbPostCreate` methods. EJBs live out their life in a container, which is a server feature that creates entity beans, manages their relationship with the data source with which they're associated, and so on.

The EJB container will use these new methods to create new `VisitBean` entity beans. It will call the `ejbCreate` method before writing the bean's state to the database. After this method has finished executing, a new database record based on the CMP fields will have been created. The container calls the `ejbPostCreate` method after the bean has been written to the database and its data has been assigned to an EJB object.

1. Edit the body of the `ejbCreate` method so it appears as follows. Code to add appears in bold. Make sure to remove the line `setKey(key)` from the method.

```
public String ejbCreate(String visitorName)
        throws CreateException{
    setVisitorName(visitorName);
    setVisitNumber(1);
    return null;
}
```

This method will be called to create a new `VisitBean`. When it's called the visitor name and visit number will be added to the database that stores values for the `VisitBean`.

2. Beneath the `ejbCreate` method code, edit the `ejbPostCreate` method so it appears as follows. Code to edit appears in bold

   ```
   public void ejbPostCreate(String visitorName){}
   ```

3. Press **Ctrl+Shift+S** to save your work.

You've written all the code you'll need for your entity bean. In the next step, you'll create a session bean that will keep track of visits.

Related Topics

Developing Entity Beans

Click one of the following arrows to navigate through the tutorial:

Step 3: Create the VisitTrackerBean Session Bean

In the preceding step you created an entity bean that models a visit. That bean represents a row of visit data in the database. In this step, you'll create the session bean (another kind of EJB) that models visit tracking.

As described in the WebLogic Server topic, How Do Applications Use EJBs?, a session bean implements business logic and acts on behalf of the client. In the case of the application you're building with this tutorial, the client will be a test web page (and, by extension, you, the page's user). In order to meet its client's requests, the session bean you're about to create will know how to find a visitor by name, know how to increment the visit number, and so on.

In this section, you will create the `VisitTrackerBean` source files.

To Create VisitTrackerBean Source Files

1. In the **Package Explorer**, expand **VisitEJBProject> src**, right-click **hello**, then click **New > WebLogic Session Bean**.
2. In the **New Session Bean** dialog, in the **File name** box, enter `VisitTrackerBean`, then click **Finish**.

   As with the `VisitBean` entity bean, this new session bean comes with a few annotations already entered for you. These are:

   - `@Session`— Defines the class-scope properties of a session bean.

Attribute	Description
`ejbName`	Descriptive name of the session bean.

3. `@JndiName` — Specifies the local and remote JNDI name of an EJB; that is, the JNDI name associated with its local and remote interface. The attributes shown here in code include:

Attribute	Description
`remote`	Remote JNDI name of the EJB.

4. `@FileGeneration` — Specifies the interface, compound primary key, and value classes that are to be auto-generated during build.

Attribute	Description
`localClass`	Whether to generate the local interface of the EJB.
`localHome`	Whether to generate the local home interface of the EJB.
`remoteClass`	Whether to generate the remote interface of the EJB.
`remoteHome`	Whether to generate the remote home interface of the EJB.
`valueClass`	Specifies whether to generate the value class for this EJB.

5. Add the following `import` statements to support code you're about to add.

```
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ejb.EJBException;
import javax.ejb.FinderException;
import javax.ejb.CreateException;
import weblogic.ejbgen.EjbLocalRefs;
import weblogic.ejbgen.EjbLocalRef;
```

6. Above the `@Session` annotation, paste the following `@EjbLocalRefs` annotation, along with the nested `@EjbLocalRef` annotation.

```
@EjbLocalRefs({
    @EjbLocalRef(home = "hello.VisitBeanLocalHome",
        jndiName = "ejb.VisitBeanLocalHome",
        local = "hello.VisitBeanLocal",
        name = "ejb/VisitBean",
        type = Constants.RefType.ENTITY)
})
```

The following table describes the attributes you're setting.

Attribute	Description
home	Local home interface of the referenced EJB.
jndiName	Local JNDI name of the referenced EJB.
local	Local (business) interface of the referenced EJB.
name	Name used to reference the other bean.
type	EJB type of the referenced bean.

7. Add the following visitHome field above the ejbCreate method that was added for you. Your code will use this variable later to create a VisitBean instance for a particular visitor.

```
private VisitBeanLocalHome visitHome;
```

8. Edit the ejbCreate method so that it looks like the following. This code will retrieve a VisitBean instance.

```
public void ejbCreate()
{
    try
    {
        javax.naming.Context initialContext = new InitialContext();
        visitHome = (VisitBeanLocalHome) initialContext.lookup("java:comp/env/ejb/VisitBean");
    } catch (NamingException ne)
    {
        throw new EJBException(ne);
    }
}
```

9. Beneath the ejbCreate method code, add the following greetVisitor method code.

```
public String greetVisitor(String visitorName)
{
    VisitBeanLocal theVisit;
    int visitNumber;

    try
    {
        // Try to find the visitor in the database.
        theVisit = visitHome.findByPrimaryKey(visitorName);
    } catch (FinderException fe)
    {
        try
        {
            // If the visitor isn't in the database,
            // create a new visitor with the name given.
            visitHome.create(visitorName);
            // Greet the new visitor.
            return "Hello, " + visitorName + "!";
        } catch (CreateException ce)
        {
            throw new EJBException(ce);
        }
    }
```

```
        // Get the returning visitor's visit number and
        // increment it from this visit.
        visitNumber = theVisit.getVisitNumber();
        theVisit.setVisitNumber(visitNumber + 1);
        // Greet the returning visitor.
        if (visitNumber == 1)
        {
            return "Hello again, " + visitorName + "!";
        } else
        {
            return "Hello, " + visitorName + "! This is visit number "
                + theVisit.getVisitNumber() + ".";
        }
    }
```

10. With a cursor in the `greetVisitor` method code, locate the **RemoteMethod** property in **Annotations** view.
11. Right-click the **RemoteMethod** property, then click **Add Annotation**. The `@RemoteMethod()` annotation should be added immediately preceding the `greetVisitor` method.
12. Press **Ctrl+S** to save your work.

With code for your entity and session beans, you're ready to start testing them. You'll get set up to do this in the next step.

Related Topics

Developing Session Beans

Click one of the following arrows to navigate through the tutorial:

Step 4: Create and Start a Server on Which to Test

In this step you'll configure WebLogic server to use a domain on which you can test your EJBs. Then you'll start up the server before creating a project to test with.

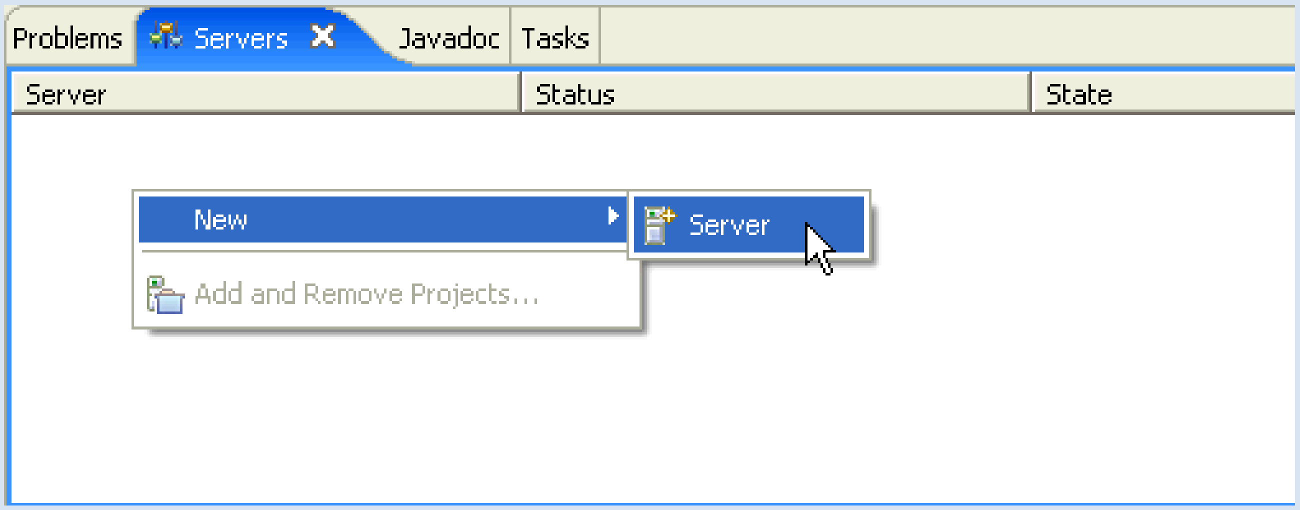
Note: if you have executed the EJB tutorial before your server may already contain previous deployments of the EJB projects. Before proceeding, it is recommended that you either (1) remove previous EJB tutorial code from your server or (2) create a new server domain.

In this section, you will:

- Create the server
- Assign the server to the EAR project

To Create the Server

1. In the Workshop perspective, the following views should be visible as tabs at the bottom (by default) of the IDE: Problems, Servers, Javadoc, Tasks.
2. Click the **Servers** view tab, right-click in its window, then click **New** > **Server**.



3. In the **New Server** dialog, under **Select the server type**, confirm that **BEA Systems, Inc.**, > **BEA WebLogic v9.2 Server** is selected, then click **Next**.
4. Under **New BEA WebLogic Server**, ensure that the dropdown **Domain home** shows the domain directory as the following, where <BEA_HOME> is the location of your BEA installation:

   <BEA_HOME>/weblogic92/samples/domains/workshop

5. Click **Finish**.

To Assign the Server to the EAR Project

Now you'll associate the server you just created with the EAR project. The EAR project, which contains your EJB project, will be deployed to it.

1. In the **Package Explorer**, right-click **EJBTutorial_EARProject**, then click **Properties**.
2. In the **Properties** dialog, in the left pane, select **Server**. In the right pane, in the **Default server** box, select **BEA WebLogic v9.2 Server**.
3. Click **OK**.

To Start the Server

- In the **Servers** view tab, right-click **BEA WebLogic v9.2 Server**, then click **Start**.

You've written the code for both of your EJBs, you've created a server on which to test, and you've started the server. Now it's time to create a test client. In the next step, you'll create a web application for that purpose.

Related Topics

None.

Click one of the following arrows to navigate through the tutorial:

Step 5: Create a Test Project

You'll use the steps in this topic to create a web project with which to test your EJBs. As with the EJB project you created earlier, this project is a framework within which to write your source code. You'll add the code in the next step.

In this section, you will:

- Create a web project for testing the EJB
- Set properties to build and run

To Create a Web Project for Testing the EJB

Begin by creating a web project that will contain the user interface code you'll use to make sure your EJB is doing what it should.

When you create a WebLogic web project, you get the beginnings of a Java Page Flow (JPF, or page flow). A page flow makes it easier for you to keep client logic and presentation separate, as you'll see in the next step.

1. Click **File > New > Project.**
2. In the **New Project** dialog, expand **Web**, click **Dynamic Web Project**, then click **Next**.
3. In the **New Dynamic Web Project** dialog, in the **Project name** box, enter `VisitWebTest`.
4. Place a check next to **Add project to an EAR**. From the dropdown menu, select **EJBTutorial_EARProject**.
5. Click **Finish**.

To Set Properties to Build and Run

With these steps, you'll set web project properties to useful values for testing the EJBs.

1. In the **Package Explorer**, right-click **VisitWebTest**, then click **Properties**.
2. In the properties dialog, in the left pane, click **Server**.
3. In the right pane, in the **Default server** box, click **BEA WebLogic v9.2**.

   This ensures that your test web project uses the server domain you created earlier.
4. Click **OK**.

With your test project created, it's time to move on to writing test code.

Related Topics

None.

Click one of the following arrows to navigate through the tutorial:

Step 6: Write Test Source Code and Test the EJBs

You're almost ready to test. In this step you'll write page flow source code that invokes your session EJB — which, in turn, invokes your entity EJB. Specifically, rather than writing typical EJB invocation code, you'll use an EJB control that handles that for you.

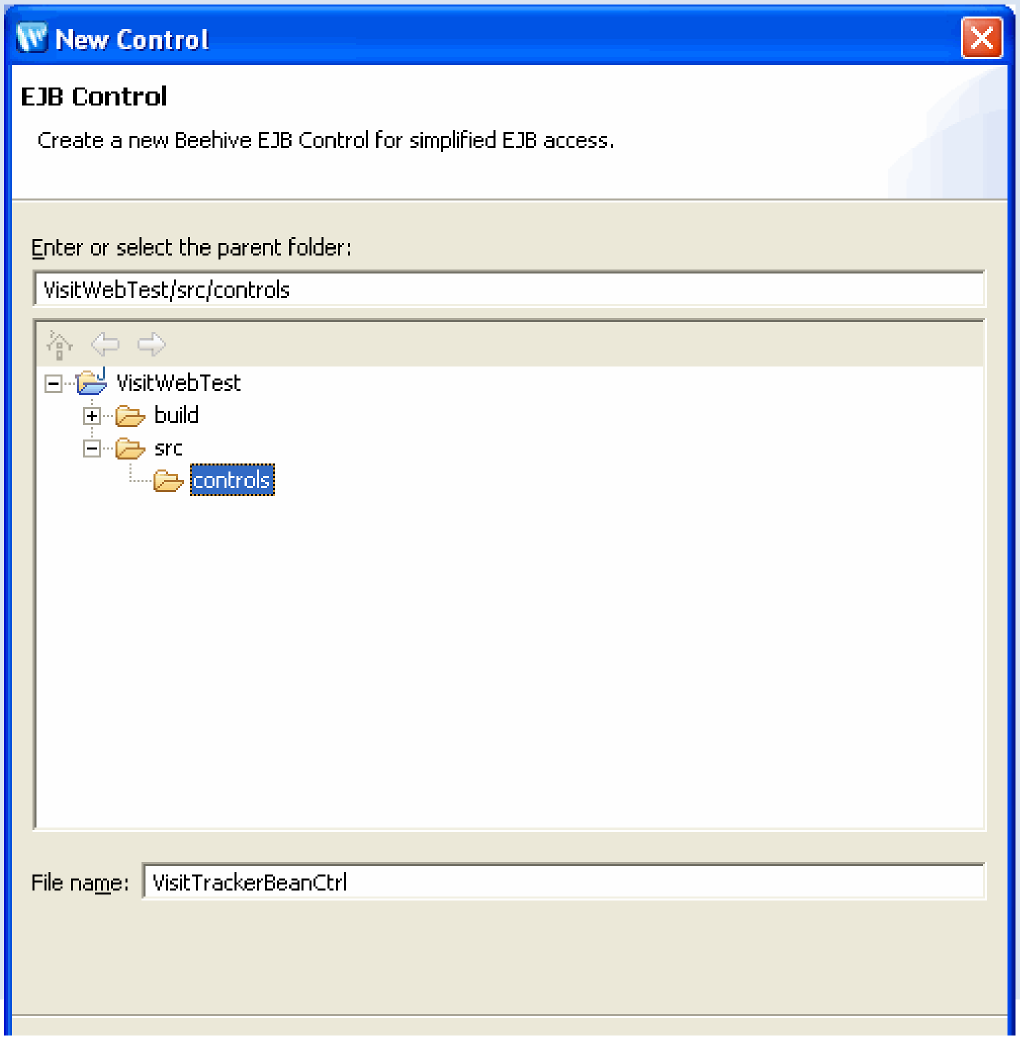
In this section, you will:

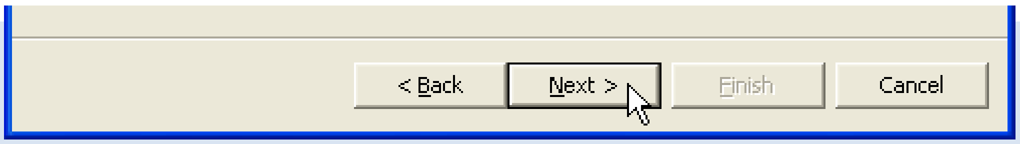
- Add an EJB control to the test project
- Add EJB control reference and method invocations to the page flow
- Create a user interface and test the EJB

To Add an EJB Control to the Test Project

In these steps you create an EJB control that will represent your entity bean.

1. In the **Package Explorer**, expand **VisitWebTest**, right-click **src**, then click **New > Package**.
2. In the **New Java Package** dialog, in the **Name** box, enter `controls`, then click **Finish**.
3. Switch to the Page Flow perspective by selecting **Window > Open Perspective > Page Flow**.
4. In the **Page Flow Explorer**, right-click the **Referenced Controls** node and select **Add Control**.
5. In the **Select Control** dialog, under **New System Control**, select **EJB Control**. Click **Ok**.
6. In the **New Control** dialog, in the **parent folder** box, expand the tree to **VisitWebTest > src > controls**, then select the **controls** node.
7. In the **Control name** box, enter `VisitTrackerBeanCtrl`, then click **Next**.

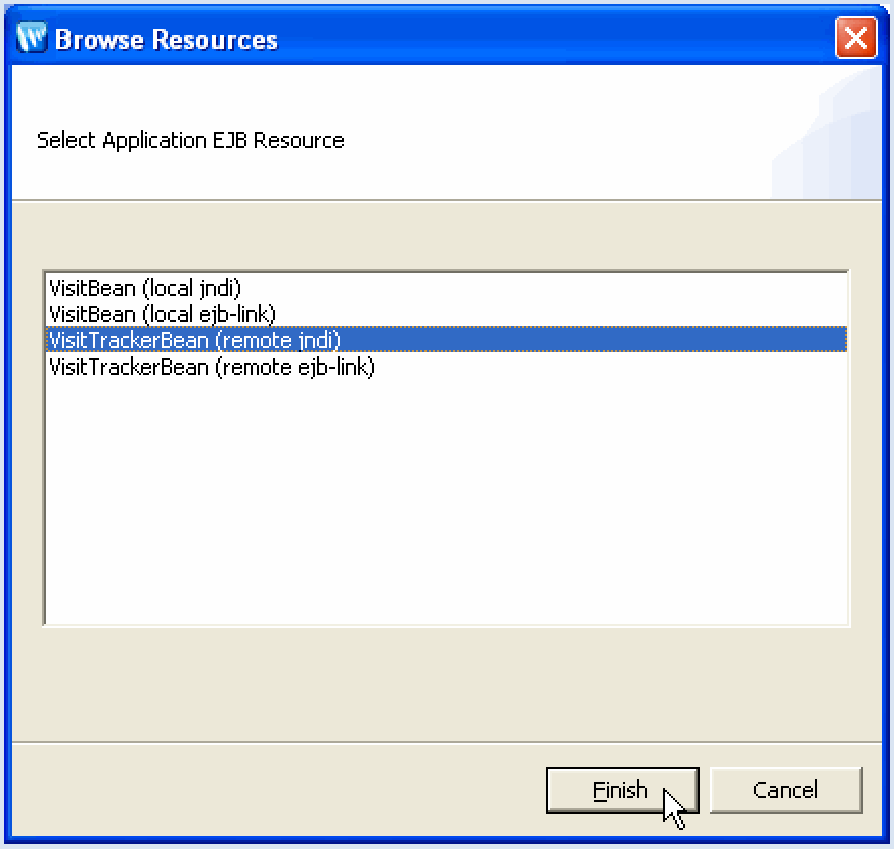

New Control
EJB Control
Create a new Beehive EJB Control for simplified EJB access.
Enter or select the parent folder:
VisitWebTest/src/controls
VisitWebTest
build
src
controls
File name:
VisitTrackerBeanCtrl



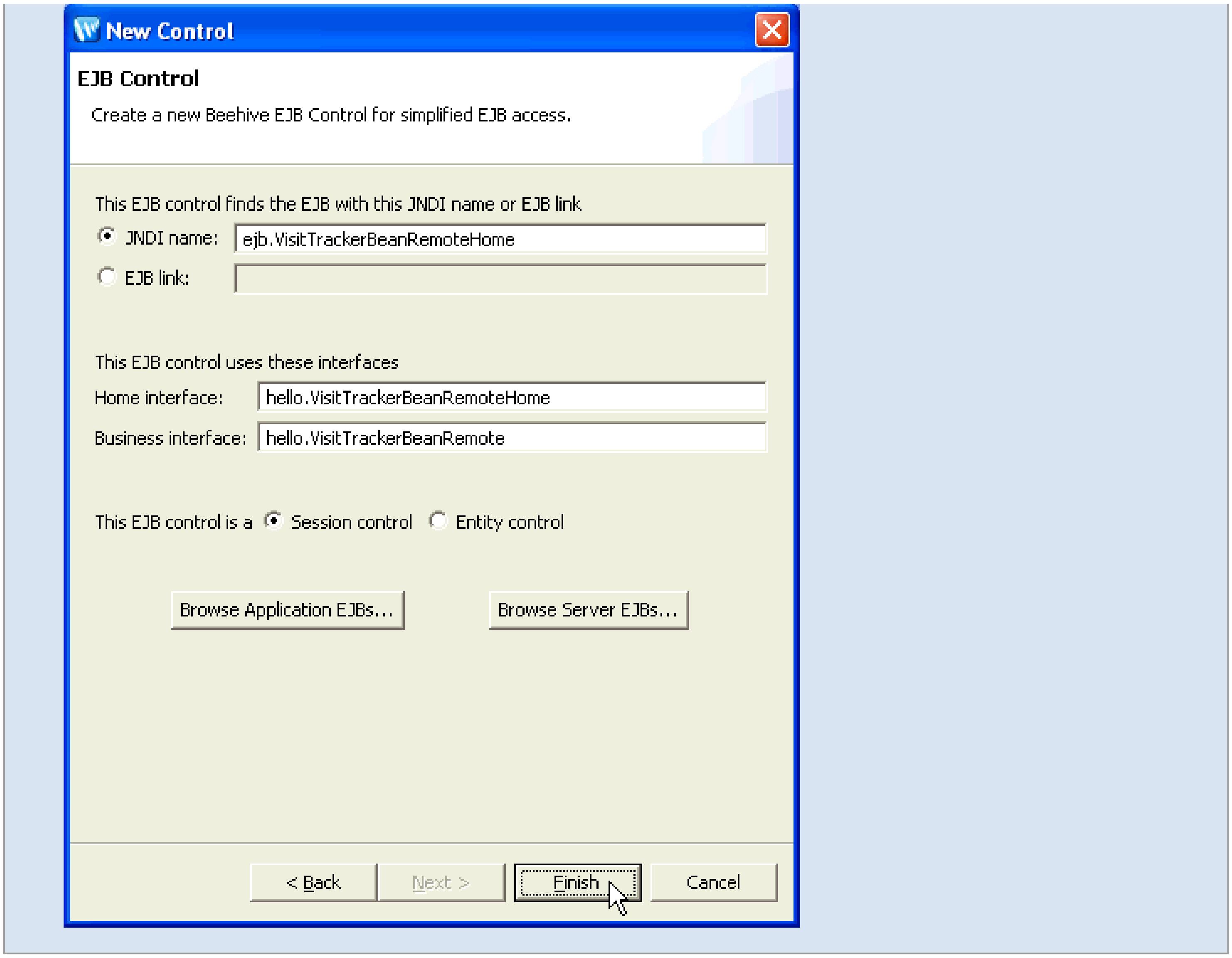
8. Under **EJB Control**, click **Browse Application EJBs**.

   In the **Browse Resources** dialog, note that the two EJBs you've created are listed.

9. Select **VisitTrackerBean (remote jndi)**, then click **Finish**.


Browse Resources
Select Application EJB Resource
VisitBean (local jndi)
VisitBean (local ejb-link)
VisitTrackerBean (remote jndi)
VisitTrackerBean (remote ejb-link)
Finish
Cancel

10. Confirm that the settings displayed are as follows, then click **Finish**.


New Control
EJB Control
Create a new Beehive EJB Control for simplified EJB access.
Browse Application EJBs...
Browse Server EJBs...
< Back
Next >
Finish
Cancel

This EJB control finds the EJB with this JNDI name or EJB link
(●) JNDI name: ejb.VisitTrackerBeanRemoteHome
( ) EJB link:

This EJB control uses these interfaces
Home interface: hello.VisitTrackerBeanRemoteHome
Business interface: hello.VisitTrackerBeanRemote

This EJB control is a (●) Session control ( ) Entity control

You'll get EJB control source code such as the following:

```
@ControlExtension
@EJBHome(jndiName="ejb.VisitTrackerBeanRemoteHome")
public interface VisitTrackerBeanCtrl extends
        hello.VisitTrackerBeanRemoteHome, // home interface
        hello.VisitTrackerBeanRemote, // business interface
        SessionEJBControl // control interface
{
    static final long serialVersionUID = 1L;
}
```

The code's simplicity masks its capability. For example, by extending the `SessionEJBControl` interface, the control supports implicit retrieval of the target EJB's home and business interface instances as well as a check to discover whether a target instance is available for method invocation. In other words, this is plumbing common to using an EJB for which you don't need to write code — the control takes care of it.

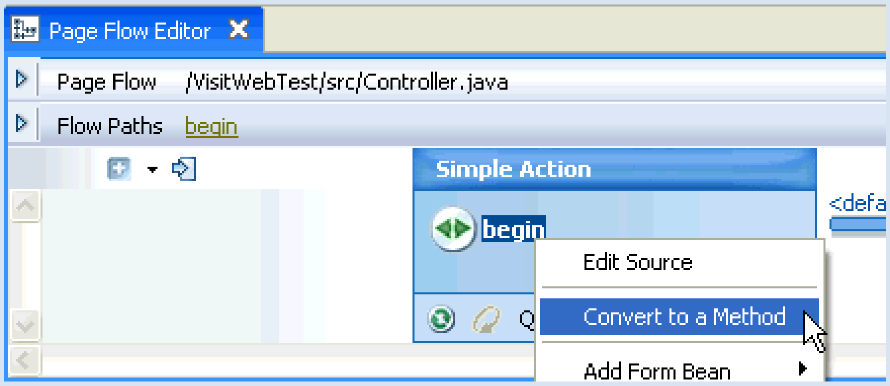
The `@ControlExtension` annotation designates to the compiler that this interface is a control. The `@EJBHome` annotation specifies the JNDI name of the home interface for the target EJB that this control provides access to.

To Add Method Invocations to the Page Flow

The preceding step inserted the control you created into your page flow. In particular, when you added a control, you added a control variable to the page flow's *controller*, represented here by code in the Controller.java file. In page flows, a controller is a central place for client logic — a place to contain state variables and data that would otherwise need to be contained in separate files (such as in JSPs).

In the case of VisitWebTest, your controller will be responsible not only for invoking your session bean via the EJB control you insert, but also for passing the bean's responses along to the JSP files that make up the presentation components in your application.

1. On the **Page Flow Explorer** tab, locate **Controller.java** and double-click it.
2. In the **Page Flow Editor** tab, locate the **begin icon**, right-click it and select **Convert to a Method**.



3. In the source code for Controller.java, add the following import statements to support code you'll be adding.

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import org.apache.beehive.netui.pageflow.Forward;
```

4. Edit the begin method so it appears as follows. This code looks up and stores the home interface for the entity bean VisitTrackerBean and forwards processing to index.jsp.

```
@Jpf.Action(forwards =
    {
        @Jpf.Forward(name = "success", path = "index.jsp")
    }
)
protected Forward begin()
{
    try
    {
        visitTrackerBeanCtrl.create();
    } catch(CreateException ce) {}
    catch(RemoteException re) {}
    return new Forward("success");
}
```

5. Beneath the begin method code, add the following code. This code provides an action that can be called from index.jsp; that action will invoke the session bean. If the action is successful, the result will be forwarded to showForm.jsp, which you'll create in a moment.

```
@Jpf.Action(forwards =
    {
        @Jpf.Forward(name = "success", path = "showForm.jsp")
    }
)
protected Forward invokeEJB(InvokeEJBForm form)
{
    String response;

    try
    {
        response = visitTrackerBeanCtrl.greetVisitor(form.getName());
    } catch(RemoteException re)
    {
        response = "An error has occurred";
    }
    getRequest().setAttribute("returnvalue", response);
    return new Forward("success");
}

/**
 * Get and set methods may be overwritten by the Form Bean editor.
 */
public static class InvokeEJBForm implements java.io.Serializable
{
    private static final long serialVersionUID = 1L;

    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return this.name;
    }
}
```

6. Press **Ctrl+Shift+S** to save your work.

To Create a User Interface and Test the EJB

Here you'll update files in your application's user interface so that the page flow controller can use them to present the results of its interaction with your EJBs.

1. In the **Page Flow Explorer** tab, open the node **Pages > index.jsp**. Double-click **index.jsp** to open the file.
2. Modify the <netui:html> tag so that it appears as follows. This code calls the invokeEJB action you coded in the preceding steps.

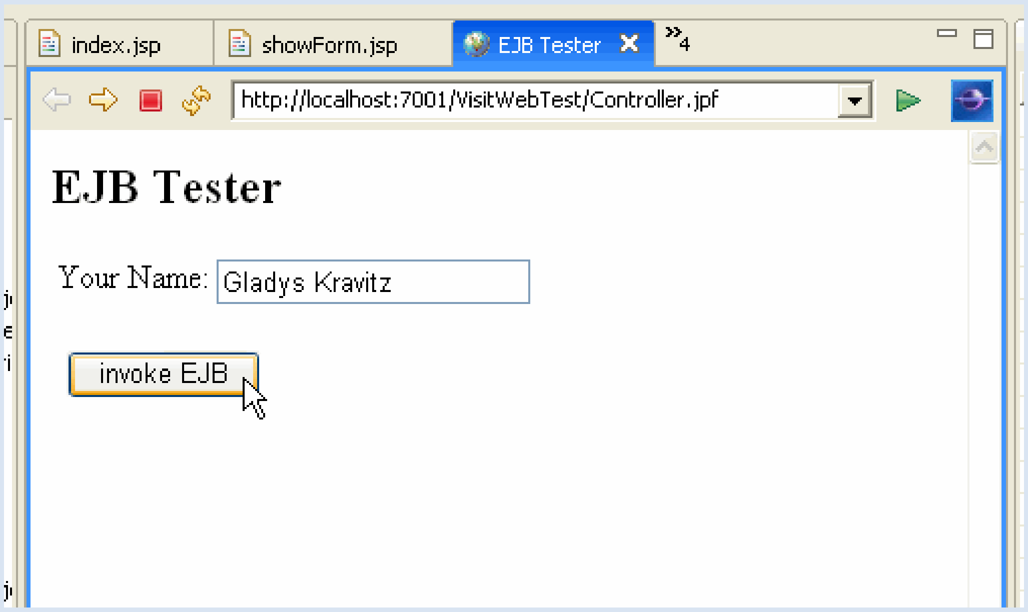
```
<netui:html>
    <head>
        <title>
            EJB Tester
        </title>
    </head>
    <body>
        <h2> EJB Tester </h2>
        <netui:form action="invokeEJB">
            <table>
                <tr valign="top">
                    <td>Your Name:</td>
                    <td>
                        <netui:textBox dataSource="actionForm.name"/>
                    </td>
                </tr>
            </table>
            <br/>&nbsp;
            <netui:button value="invoke EJB" type="submit"/>
        </netui:form>
    </body>
</netui:html>
```

3. On the **Page Flow Explorer** tab, right-click the **Pages > showForm.jsp** node and select **Create**.
4. Double-click the **Pages > showForm.jsp** node to open its source code.
5. Replace its source code with the following. This code receives results from a successful invocation of the session bean and displays a response.

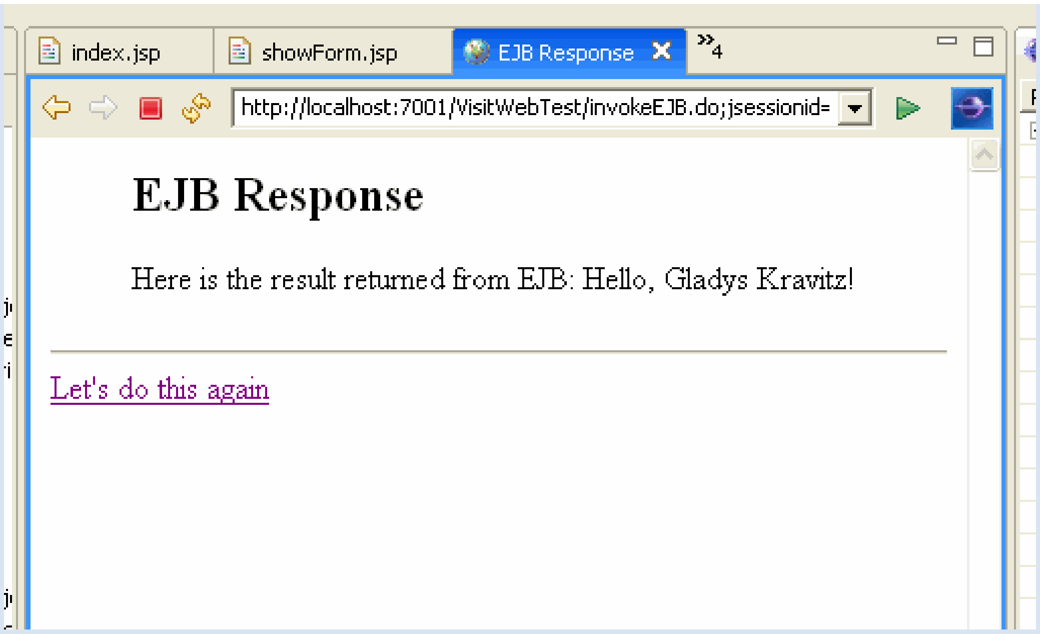
```
<%@ page language="java" contentType="text/html;charset=UTF-8"%>
<%@taglib uri="http://beehive.apache.org/netui/tags-html-1.0" prefix="netui"%>
<%@taglib uri="http://beehive.apache.org/netui/tags-databinding-1.0" prefix="netui-data"%>
<%@taglib uri="http://beehive.apache.org/netui/tags-template-1.0" prefix="netui-template"%>
```

```
<netui:html>
    <head>
        <title>
            EJB Response
        </title>
    </head>
    <body>
        <blockquote>
        <h2>EJB Response</h2>
        <p>
        Here is the result returned from EJB: <netui:label value="${requestScope.returnvalue}"></netui:
label>
        </blockquote>
        <hr>
        <netui:anchor href="index.jsp">Let's do this again</netui:anchor>
    </body>
 </netui:html>
```

6. Press **Ctrl+Shift+S** to save your work.
7. On the **Page Flow Explorer** tab, right-click **Controller.java**, then click **Run As > Run on Server**.
8. If the **Select Tasks** dialog is displayed, click **Finish**.
9. After Workshop for WebLogic has finished building your application's components, it will display the index.jsp.
10. Enter a name in the box provided, then click **invoke EJB**.



11. Note that the response page includes the name you entered echoed back to you. Remember that the logic that generates this message is contained in the `VisitTrackerBean` session bean that you created in Step 3.



EJB Response

Here is the result returned from EJB: Hello, Gladys Kravitz!

Let's do this again

12. To keep testing, click **Let's do this again**, enter the same name as before, then click **invoke EJB** again. With each test you'll see an updated response.

That's it! You've built and tested two Enterprise JavaBeans with Workshop for WebLogic. The next "step" in this tutorial is a summary of what the tutorial covered, along with links to information you might find helpful.

Related Topics

None

Click one of the following arrows to navigate through the tutorial:

Summary: Building Enterprise JavaBeans

Through this tutorial, you used Workshop for WebLogic to build a very simple application that uses Enterprise JavaBeans to model the parts of a visitor tracking application. You also created a page flow to use as a test client.

Concepts and Tasks Introduced in This Tutorial

- In Workshop for WebLogic, EJB source code is contained in a single JAVA file. Supporting files, such as those for implemented EJB interfaces, are generated for you at build time based on the annotations you use in source code.
- When developing EJBs you annotate source code to specify the details for automatically generated source artifacts. For example, the `@FileGeneration` annotation's attributes specify whether to generate files, as well as the names of generated files.
- Some source code annotations are used at build time to generate values for deployment descriptors. For example, the `@Entity` annotation's `primKeyClass` attribute value is used in the `<prim-key-class>` element of the bean's deployment descriptor.
- Most annotations in EJB source code are defined by the WebLogic Server EJBGen tool. This tool is invoked at build time to process your annotations. Note that you do not need to separately run EJBGen on your Workshop for WebLogic source code. For more information on EJBGen, see EJBGen Reference on eDocs.
- An EAR project is a special kind of project through which you can generate a single Enterprise ARchive file that contains the outputs of multiple projects — such as projects whose outputs make up a single application. Use an EAR project to represent your application as a whole.
- You can use EAR project *library modules* to collect classpath dependencies that are shared across development artifacts. In this way, you needn't copy the same JAR files into the project hierarchy of each project.
- The EJB control is a handy way to invoke EJBs from within client code. The EJB control handles typical EJB client plumbing by virtue of your specifying annotation values that indicate which EJB the control represents.
- A good way to test EJBs as you develop is to use a page flow as a test client. You can create an EJB control that represents your EJB, then call control methods within your page flow code. For more information about page flows, see Page Flow Overview at the Apache Beehive web site.
- The "Workshop" IDE perspective provides an easy way to display the IDE views you'll likely want when developing application components.

Related Topics

Developing Enterprise JavaBeans

Enterprise JavaBeans in Workshop for WebLogic

Click the following arrow to navigate through the tutorial:

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.

Topics Included in This Section

Related Topics

Tutorial: Building Enterprise JavaBeans
Provides a step-by-step guide to developing Enterprise JavaBeans.

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?

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.

Developing Entity Beans with Workshop for WebLogic

In Workshop for WebLogic, you develop an entity bean by creating a class that extends weblogic.ejb.GenericEntityDrivenBean and implements javax.ejb.EntityBean. You annotate this class with @Entity, @JndiName, and @FileGeneration annotations (and others, as needed) that specify EJB characteristics. You also add annotations to designate virtual fields, select and find methods, and relations that the entity has with other entities.

You can get started easily in the IDE by using the WebLogic Entity Bean template. When you use the template, the IDE generates code such as the following:

```
/**
 * GenericEntityBean subclass automatically generated by Workshop.
 *
 * Please review and update the existing content to ensure it matches your
 * intended use (esp. the Entity and JndiName annotations and the primary key
 * field, ejbCreate() and ejbPostCreate() methods).
 */
@Entity(ejbName = "MyEntityBean",
        dataSourceName = "samplesDataSource",
        tableName = "MyEntityBean",
        primKeyClass = "java.lang.Integer")
@JndiName(local = "ejb.MyEntityBeanLocalHome")
@FileGeneration(localClass = Constants.Bool.TRUE,
        localHome = Constants.Bool.TRUE,
        remoteClass = Constants.Bool.FALSE,
        remoteHome = Constants.Bool.FALSE,
        valueClass = Constants.Bool.TRUE)
abstract public class MyEntityBean
    extends GenericEntityBean
    implements EntityBean {

    /**
     * IMPORTANT: Automatically generated ejbCreate() method.
     * Please change as appropriate.
     */
    public java.lang.Integer ejbCreate(java.lang.Integer key)
            throws CreateException {
        setKey(key);
        return null;
    }

    /**
     * IMPORTANT: Automatically generated ejbPostCreate() method.
     * Please change as appropriate.
     */
    public void ejbPostCreate(java.lang.Integer key) {
    }

    /**
     * IMPORTANT: Automatically generated primary key field getter method.
     * Please change name and class as appropriate.
     */
    @CmpField(column = "key", primkeyField = Constants.Bool.TRUE)
    @LocalMethod()
    public abstract java.lang.Integer getKey();

    /**
     * IMPORTANT: Automatically generated primary key field setter method.
     * Please change name and class as appropriate.
     */
    @LocalMethod()
    public abstract void setKey(java.lang.Integer key);
}
```

> **Note:** To use the WebLogic Entity Bean template, in Workshop perspective right-click the package that will contain the bean, point to New, then click WebLogic Entity Bean.

The code includes typical values for the commonly used class-level annotation attributes. In addition, the template-generated code includes the container-managed persistence (CMP) fields for a primary key. The idea is to provide a starting place for your own code -- for you to rewrite it with your code for the create methods, for virtual field declarations, methods for business logic, and so on.

Workshop for WebLogic uses these annotations to generate the interfaces and descriptor files that are required for EJB entity beans. The following sections describe these entity bean pieces and characteristics.

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.

In Workshop for WebLogic, you can use the Annotations view to set or view the interface names for a given entity bean. To do this, open the bean's source code and place your cursor in the bean class code (in other words, not a method or other member or annotation). In the Annotations view, scroll to where the FileGeneration annotation's attributes are listed. There, you'll see attributes such as localHomeName and localClassName (the business interface), as well as remoteHomeName and remoteClassName (the remote interface). These correspond to attributes for the @FileGeneration Annotation in your source code.

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.

In entity bean code, you designate CMP fields with the @CmpField annotation.

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` *`PrimaryKeyClass`* `ejbCreate(`*`parameters`*`)`. The *`PrimaryKeyClass`* can be a primitive type, such as `Integer`, when a single primary key is defined, or it can be a separate primary keytes a primary key class with the name provided as an attribute in the @FileGeneration 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 @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.

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`*`)`. This method will in turn invoke the bean's `setFirstName`, `setLastName` and `setAge` methods to update the CMP fields holding this information.

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 an `ejbHome`*`MethodName`* method in the bean, and is exposed as a *`MethodName`* 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.

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` (*`PrimaryKeyClass`*) 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 uniquely defined by the method's parameter. As before, the *`PrimaryKeyClass`* can be a primitive type, such as `Integer`, when a single primary key is defined, or it can be a separate primary key class.

In Workshop for WebLogic, you can use the Annotations view to find out which primary key class is used for an entity bean. To do this, open the bean's source code and place your cursor in the class name. In the Annotations view, scroll to where the Entity annotation's attributes are listed, then look for the primKeyClass attribute. This attribute might also be given in the bean's `@Entity` annotation in source code.

In entity code, you specify finder and select methods with the @Finder and @Select annotations. For more information on these and the EJB QL (query language), see Query Methods and EJB QL.

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.

In entity code, you designate relations with the @Relation annotation. For more information, see Entity Relationships.

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 Life Cycle of an Entity Bean.

Related Topics

@FileGeneration Annotation

@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

The WebLogic Entity Bean wizard makes it easy to start creating an entity bean from scratch. If you are designing a new entity bean in a WebLogic EJB project, using the wizard will create basic entity bean code from a template. To use the wizard, in Workshop for WebLogic, right-click your WebLogic EJB project folder, point to New, then click WebLogic Entity Bean.

For more information on creating new beans, see Tutorial: Building Enterprise JavaBeans. When you create a new entity bean, by default the local interfaces and various other defaults are defined. For more details, see @FileGeneration Annotation.

> **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.

Automatic Table Creation

When you are developing a new entity bean, you can make iterative development easier by enabling automatic table creation. You can have the server create the table when it is not present, or 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 in the Properties dialog for the WebLogic EJB project. In that dialog, in the left pane, click WebLogic EJB; in the right pane choose a value from th "create tables" dropdown. The default setting is `CREATE_ONLY`. To recreate a table if the definition of the entity bean does not match the table specification, use `DROP_AND_CREATE`. For more information regarding the possible settings, see @JarSettings Annotation.

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

Defining a Basic Entity Bean

The following 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. This entity bean allows you to add a new product, find a product using its primary key, and get and set its CMP field values.

```
package myBeans;

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

@Entity(ejbName = "Product",
        dataSourceName = "samplesDataSource",
        tableName = "product",
        abstractSchemaName = "Product",
        primKeyClass = "String")
@AutomaticKeyGeneration(cacheSize = "1",
        name = "NamedSequence",
        type = AutomaticKeyGeneration.AutomaticKeyGenerationType.SEQUENCE_TABLE)
@JndiName(local = "ejb.ProductLocalHome")
@FileGeneration(localClass = Constants.Bool.TRUE,
        localClassName = "Product",
        localHome = Constants.Bool.TRUE,
        localHomeName = "ProductHome",
        remoteClass = Constants.Bool.FALSE,
        remoteHome = Constants.Bool.FALSE,
        remoteHomeName = "ProductRemoteHome",
        remoteClassName = "ProductRemote",
        valueClass = Constants.Bool.FALSE,
        valueClassName = "ProductValue",
        pkClass = Constants.Bool.TRUE)
abstract public class ProductBean
    extends GenericEntityBean
    implements EntityBean {
    @CmpField(column = "Name",
            primkeyField = Constants.Bool.TRUE)
    @LocalMethod()
    public abstract String getName();

    @LocalMethod()
    public abstract void setName(String arg);

    @CmpField(column = "Price")
    @LocalMethod()
    public abstract double getPrice();

    @LocalMethod()
    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 Workshop for 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. Workshop for WebLogic interprets **`ejbgen`** annotations in your source code to generate these files. For example, the @FileGeneration annotation specifies the names of the local home and business interface for the ProductBean. The @LocalMethod annotations on the accessor methods specify that these methods are defined in the local business interface.

You can view the generated JAVA files in Resource view. In that view, expand the .apt_src folder to view folders corresponding to your source packages. You'll find the generated files in these folders. For the `ProductBean`, you'd find a `Product` interface that extends `EJBLocalObject` and a `ProductHome` interface that extends `EJBLocalHome`. You can view the CLASS files compiled from the generated files (again, in Resource view) by expanding the project's build folder.

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.

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. These interfaces include:

- `javax.ejb.EJBLocalObject`, extended by the local interface
- `javax.ejb.EJBLocalHome`, extended by the home interface
- `javax.ejb.EJBObject`, extended by the remote interface
- `javax.ejb.EJBHome`, extended by the remote home interface

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 interfaces is a `remove` method you can use to remove an entity bean instance. In other words, when you invoke the `remove` method on an entity bean, you remove the bean and its 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 invoke 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:

```
package myBeans;

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

import weblogic.ejb.*;
import weblogic.ejbgen.*;

@EjbLocalRefs( { @EjbLocalRef(link = "Product") })
@Session(ejbName = "SomeSession")
@JndiName(remote = "ejb.SomeSessionRemoteHome")
@FileGeneration(remoteClass = Constants.Bool.TRUE,
        remoteHome = Constants.Bool.TRUE,
        localClass = Constants.Bool.TRUE,
        localHome = Constants.Bool.TRUE,
        localClassName = "SomeSessionLocal",
        localHomeName = "SomeSessionLocalHome")
public class SomeSession extends GenericSessionBean
    implements SessionBean {
    private static final long serialVersionUID = 1L;

    @LocalMethod()
    public void deleteViaHome(String thePk) {
        try {
            javax.naming.Context ic = new InitialContext();
            ProductHome productHome = (ProductHome) ic
                    .lookup("java:comp/env/ejb/Product");
            productHome.remove(thePk);
        } catch (NamingException ne) {
            // Exception handling code.
        } catch (RemoveException re) {
            // Exception handling code.
        }
    }

    @LocalMethod()
    public void deleteViaBusiness(String 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) {
            // Exception handling code.
        } catch (FinderException ne) {
            // Exception handling code.
        } catch (RemoveException re) {
            // Exception handling code.
        }
    }

    public void ejbCreate() {
        // Bean initialization code here.
    }

}
```

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 source 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

@FileGeneration Annotation

@LocalMethod Annotation

Automatic Primary Key Generation

With Workshop for WebLogic you can specify in your bean code that a primary key should automatically be generated when creating a new CMP entity bean. This eliminates the need for you to provide 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

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 `@AutomaticKeyGeneration` annotation to point to this table.

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

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

This creates a sequences of primary key values, 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 `cacheSize` attribute of the `@AutomaticKeyGeneration` annotation:

```
@AutomaticKeyGeneration(cacheSize = "3",
        name = "myOracleSequence",
        type = AutomaticKeyGeneration.AutomaticKeyGenerationType.SEQUENCE)
```

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 @JarSettings 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 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(30))
```

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:

```
@AutomaticKeyGeneration(cacheSize = "3",
        name = "mySqlServerID",
        type = AutomaticKeyGeneration.AutomaticKeyGenerationType.IDENTITY)
```

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 @JarSettings Annotation. For more information on the definition of a CMP entity bean, see below.

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:

```
@AutomaticKeyGeneration(cacheSize = "100",
        name = "MyNamedSequence",
        type = AutomaticKeyGeneration.AutomaticKeyGenerationType.SEQUENCE_TABLE)
```

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 @JarSettings Annotation. For more information on the definition of a CMP entity bean, see the next section.

> **Note**. When you specify a `cacheSize` value 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 use the the `@AutomaticKeyGeneration` annotation to point to the name of the primary key generator table to obtain primary keys. 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 described above, and that `ejbCreate` method does not take a primary key:

```
@AutomaticKeyGeneration(cacheSize = "1",
        name = "NamedSequence",
        type = AutomaticKeyGeneration.AutomaticKeyGenerationType.SEQUENCE_TABLE)
@Entity(defaultTransaction = Constants.TransactionAttribute.SUPPORTS,
        primKeyClass = "Integer",
        ejbName = "Customer_APK",
        dataSourceName = "samplesDataSource",
        tableName = "ejb_customer",
        abstractSchemaName = "Customer")
@FileGeneration(localClass = Constants.Bool.TRUE,
        localClassName = "AutomaticPK_CustomerLocal",
        localHome = Constants.Bool.TRUE,
        localHomeName = "AutomaticPK_CustomerHome",
        remoteClass = Constants.Bool.FALSE,
        remoteHome = Constants.Bool.FALSE,
        remoteHomeName = "CustomerRemoteHome",
        remoteClassName = "CustomerRemote",
        valueClass = Constants.Bool.FALSE,
        valueClassName = "CustomerValue",
        pkClass = Constants.Bool.TRUE)
@JndiName(local = "ejb.AutomaticPK_CustomerLocalHome")
public abstract class Customer_APK
    extends GenericEntityBean implements EntityBean {

    @CmpField(column = "FirstName")
    @LocalMethod()
    public abstract String getFirstName();

    @LocalMethod()
    public abstract void setFirstName(String arg);

    @CmpField(column = "LastName")
    @LocalMethod()
    public abstract String getLastName();

    @LocalMethod()
    public abstract void setLastName(String arg);

    @CmpField(primkeyField = Constants.Bool.TRUE, column = "Customer_ID")
    @LocalMethod()
    public abstract Integer getCustomer_ID();

    @LocalMethod()
    public abstract void setCustomer_ID(Integer arg);

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

        return null;
    }

    public void ejbPostCreate(java.lang.String FirstName,
            java.lang.String LastName) {
    }
}
```

Related Topics

@AutomaticKeyGeneration Annotation

@JarSettings Annotation

Relationships in Entity Beans

Entity relationships in CMP entity beans are used to model real-world dependencies between business concepts. This topic 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 the WebLogic Server documentation on Using Container-Managed Relationships and Defining Container-Managed Relationships. Note that much of the WebLogic Server documentation describes these concepts in the context of EJB descriptors, which are automatically generated by Workshop for WebLogic.

Each of the sections in this topic focuses on a relationship type:

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 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.

Here's how the @Relation annotation on the Concertgoer bean might look:

```
@Relation(roleName = "ConcertgoerHasTicket",
    cmrField = "ticket",
    targetEjb = "Ticket",
    multiplicity = Relation.Multiplicity.ONE,
    name = "Concertgoer-Ticket")
abstract public class Concertgoer
    extends GenericEntityBean
    implements EntityBean
```

Here's how the @Relation annotation on the Ticket bean might look:

```
@Relation(roleName = "ConcertgoerHasTicket",
    targetEjb = "Concertgoer",
    multiplicity = Relation.Multiplicity.ONE,
    name = "Concertgoer-Ticket")
abstract public class Ticket
    extends GenericEntityBean
    implements EntityBean
```

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 herself 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 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) throws
CreateException
   {
       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 @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 a "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.

Here's how the @Relation annotation on the Concertgoer bean might look:

```
@Relation(roleName = "ConcertgoerHasCreditCard",
    cmrField = "creditCard",
    targetEjb = "CreditCard",
    multiplicity = Relation.Multiplicity.ONE,
    name = "Concertgoer-CreditCard")
abstract public class Concertgoer
    extends GenericEntityBean
    implements EntityBean
```

Here's how the @Relation annotation on the CreditCard bean might look:

```
@Relation(roleName = "CreditCardNamesConcertgoer",
    cmrField = "concergoer",
    targetEjb = "Concertgoer",
    multiplicity = Relation.Multiplicity.ONE,
    name = "Concertgoer-CreditCard")
abstract public class CreditCard
    extends GenericEntityBean
    implements EntityBean
```

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.

Here's how the @Relation annotation on the Concertgoer bean might look:

```
@Relation(roleName = "ConcertgoerBoughtCDs",
    cmrField = "compactDisc",
    targetEjb = "CompactDisc",
    multiplicity = Relation.Multiplicity.ONE,
    name = "Concertgoer-CDs")
abstract public class Concertgoer
    extends GenericEntityBean
    implements EntityBean
```

Here's how the @Relation annotation on the CompactDisc bean might look:

```
@Relation(roleName = "CDsBoughtByConcertgoer",
    targetEjb = "Concertgoer",
    multiplicity = Relation.Multiplicity.MANY,
    name = "Concertgoer-CDs")
```

```
abstract public class CompactDisc
    extends GenericEntityBean
    implements EntityBean
```

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 Server 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.

Here's how the @Relation annotation on the Concertgoer bean might look:

```
@Relation(roleName = "ConcertgoerHasCreditCards",
    cmrField = "creditCards",
    targetEjb = "CreditCard",
    multiplicity = Relation.Multiplicity.ONE,
    name = "Concertgoer-CreditCards")
```

```
abstract public class Concertgoer
    extends GenericEntityBean
    implements EntityBean
```

Here's how the @Relation annotation on the CreditCard bean might look:

```
@Relation(roleName = "CreditCardsBelongToConcertgoer",
    cmrField = "concergoer",
    targetEjb = "Concertgoer",
    multiplicity = Relation.Multiplicity.MANY,
    name = "Concertgoer-CreditCards")
abstract public class CreditCard
    extends GenericEntityBean
    implements EntityBean
```

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.

Here's how the @Relation annotation on the Venue bean might look:

```
@Relation(roleName = "VenueHostsConcerts",
    targetEjb = "Concert",
    multiplicity = Relation.Multiplicity.ONE,
    name = "Venue-Concerts")
abstract public class Venue
    extends GenericEntityBean
    implements EntityBean
```

Here's how the @Relation annotation on the Concert bean might look:

```
@Relation(roleName = "ConcertsOccurAtVenue",
    cmrField = "venue",
    targetEjb = "Venue",
    multiplicity = Relation.Multiplicity.MANY,
    name = "Venue-Concerts")
abstract public class Concert
    extends GenericEntityBean
    implements EntityBean
```

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.

Here's how the @Relation annotation on the Concertgoer bean might look:

```
@Relation(roleName = "ConcertgoerAttendsConcerts",
    cmrField = "concerts",
    targetEjb = "Concert",
    multiplicity = Relation.Multiplicity.MANY,
    name = "Concertgoer-Concerts")
abstract public class Concertgoer
    extends GenericEntityBean
    implements EntityBean
```

Here's how the @Relation annotation on the Concert bean might look:

```
@Relation(roleName = "ConcertsHaveManyConcertgoers",
    targetEjb = "Concertgoer",
    multiplicity = Relation.Multiplicity.MANY,
    name = "Concertgoer-Concerts")
abstract public class Concert
    extends GenericEntityBean
    implements EntityBean
```

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.

Here's how the @Relation annotation on the Passenger bean might look:

```
@Relation(roleName = "PassengersTakeFlights",
    cmrField = "flights",
    targetEjb = "Flight",
    multiplicity = Relation.Multiplicity.MANY,
    name = "Paesengers-Flights")
abstract public class Passenger
    extends GenericEntityBean
    implements EntityBean
```

Here's how the @Relation annotation on the Flight bean might look:

```
@Relation(roleName = "FlightsHaveManyPassengers",
    cmrField = "passengers",
    targetEjb = "Passenger",
    multiplicity = Relation.Multiplicity.MANY,
    name = "Passengers-Flights")
abstract public class Flight
    extends GenericEntityBean
    implements EntityBean
```

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

@Relation Annotation

Query Methods and EJB QL

Workshop for WebLogic supports annotations through which you can more easily add find and select methods to your entity beans. You may already have used these annotations through the EJBGen tool provided with WebLogic Server. This topic provides an introduction to find and select methods and EJB QL. For more complete documentation on the annotations, see the EJBGen Reference.

You define find and select methods for CMP (2.0) entity beans 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.

Without the EJBGen annotations, you would typically specify these query methods through a combination of method declarations and the bean's deployment descriptor. Through values you specify in the annotations, Workshop for WebLogic updates the descriptor for you and generates the needed method declarations in bean interfaces.

> **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:

Specifying Find Methods

A find 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`. Find methods must start with the prefix `find`.

In Workshop for WebLogic, you specify a find method by using the @Finder annotation on the class declaration. You collect multiple find methods within an @Finders annotation. Through the @Finder annotation's attributes you indicate query method features. Basic features include the query language to use (EJB QL or WebLogic QL), which interfaces the method should be declared in (local or remote home), and the method's Java signature.

The following example illustrates two find methods declared within an `ItemsBean_F` entity bean. Both use EJB QL and both will appear within the bean's local home interface.

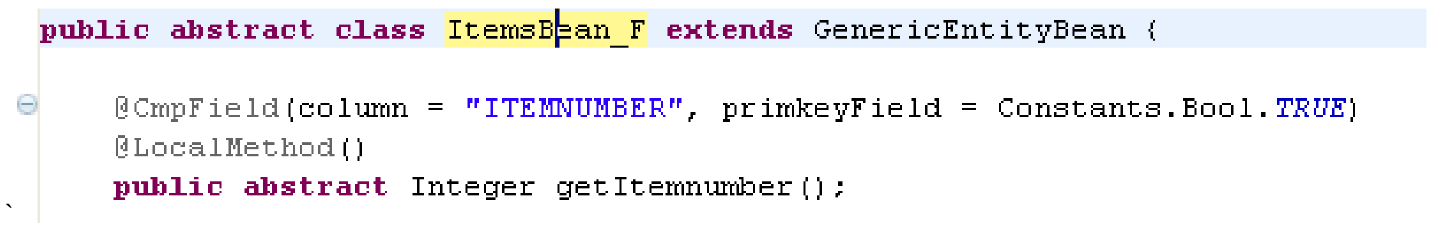
```
@Finders( {
        @Finder(ejbQl = "SELECT OBJECT(o) from ItemsBean_F as o " +
                "WHERE o.itemname = ?1",
                generateOn = Finder.GenerateOn.LOCAL,
                signature = "Collection findByItemName(java.lang.String itemname)"),
        @Finder(ejbQl = "SELECT OBJECT(i) from ManufacturerBean_F as o, " +
                "IN(o.items) AS i WHERE o.usManufacturer = 1",
                generateOn = Finder.GenerateOn.LOCAL,
                signature = "Collection findByUSManufacturer()")
})
```

> **Note:** For more information on the @Finder annotation, see weblogic.ejbgen.Finder in the EJBGen

Reference.

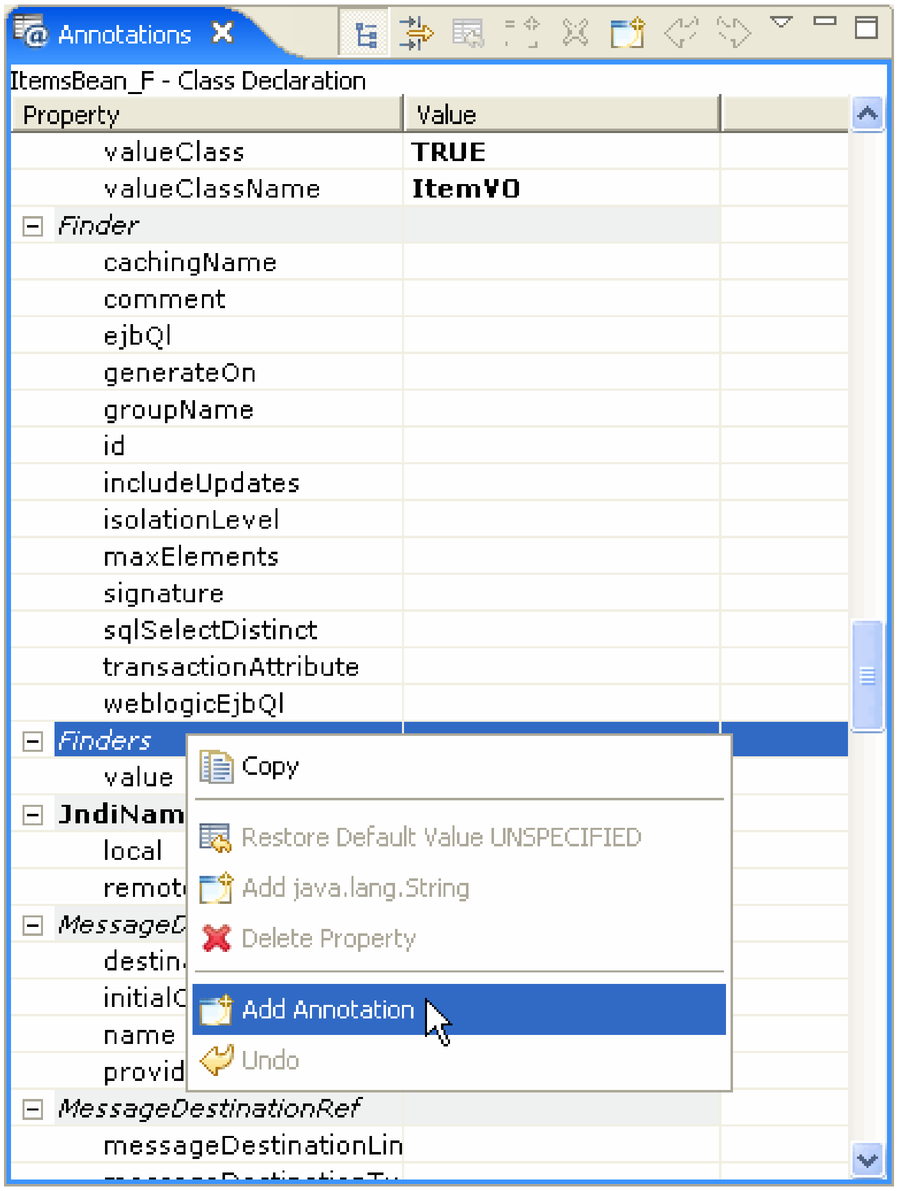
In the IDE, you can view and set annotation values using the Annotations view. To do this, follow these basic steps:

1. With your bean source code visible, place your cursor in the bean class declaration.



```
public abstract class ItemsBean_F extends GenericEntityBean {

    @CmpField(column = "ITEMNUMBER", primkeyField = Constants.Bool.TRUE)
    @LocalMethod()
    public abstract Integer getItemnumber();
```

2. In the **Annotations** view, scroll to where the **Finders** annotation is listed, right-click it, then click **Add Annotation**.

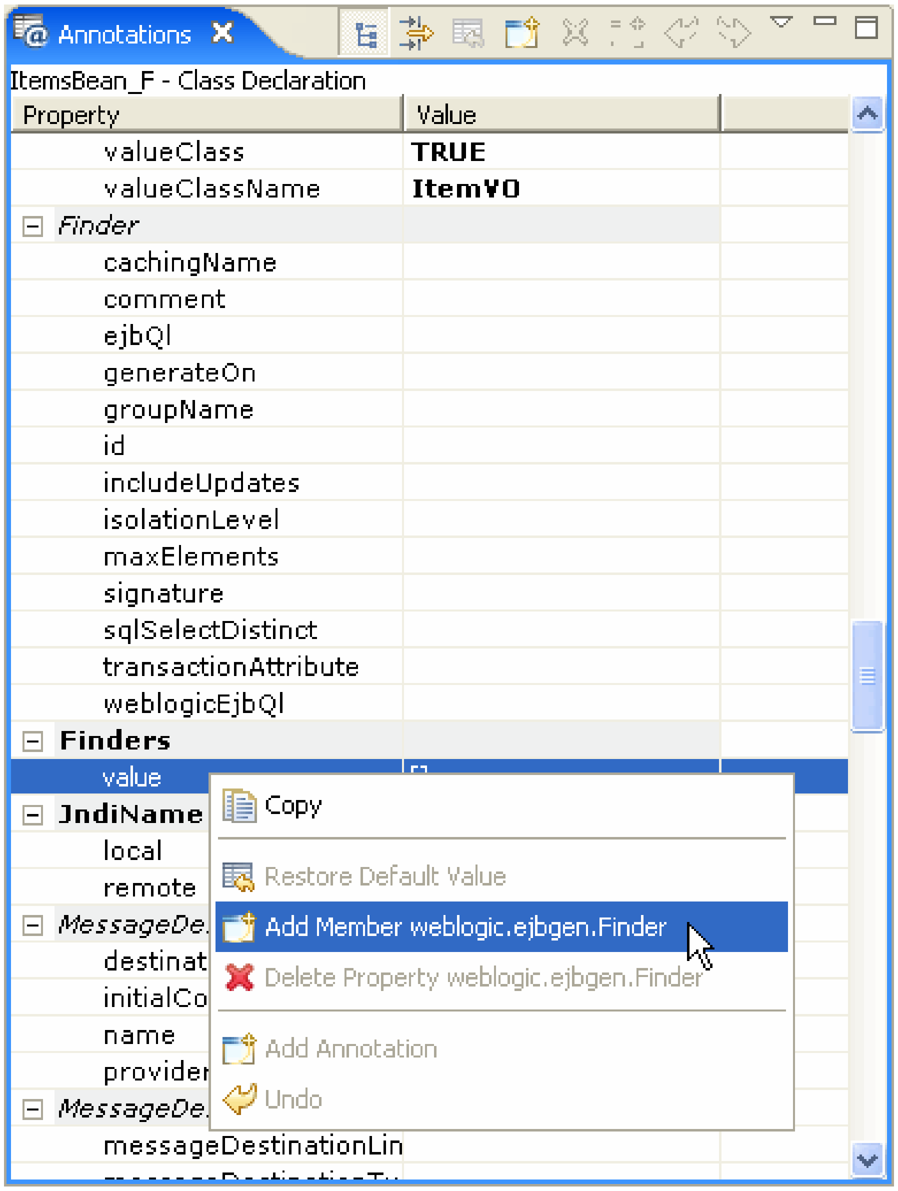


ItemsBean_F - Class Declaration

Property	Value
valueClass	TRUE
valueClassName	ItemVO
Finder	
cachingName	
comment	
ejbQl	
generateOn	
groupName	
id	
includeUpdates	
isolationLevel	
maxElements	
signature	
sqlSelectDistinct	
transactionAttribute	
weblogicEjbQl	
Finders	
value	
JndiNam	
local	
remot	
MessageD	
destin	
initialC	
name	
provid	
MessageDestinationRef	
messageDestinationLin	

Note that the @Finders annotation is written to your source code.

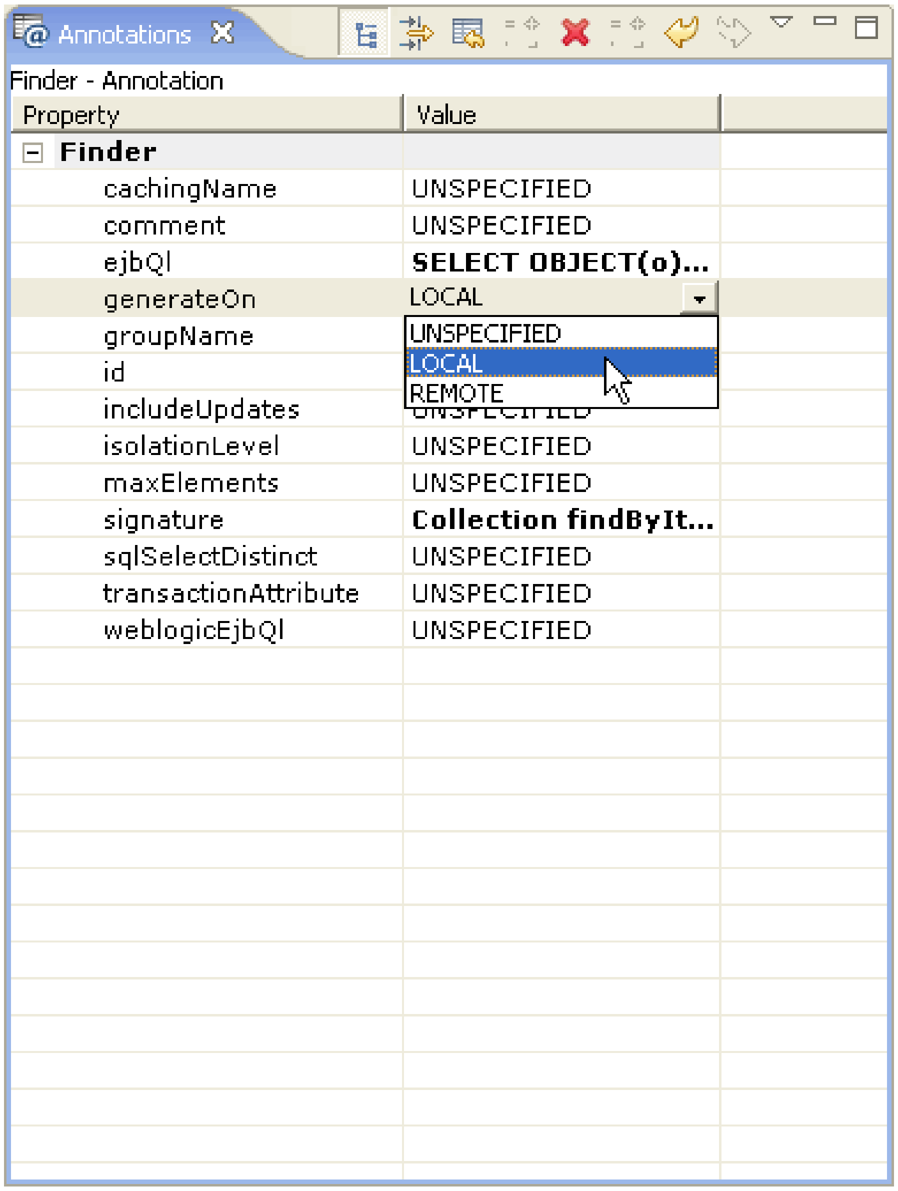
3. With your cursor in the class declaration or the new @Finders annotation, in **Annotations** view right-click **Finders**, then click **Add Member weblogic.ejbgen.Finder**.


Annotations
ItemsBean_F - Class Declaration
Property
Value
Copy
Restore Default Value
Add Member weblogic.ejbgen.Finder
Delete Property weblogic.ejbgen.Finder
Add Annotation
Undo

Property	Value
valueClass	TRUE
valueClassName	ItemVO
⊟ *Finder*	
cachingName	
comment	
ejbQl	
generateOn	
groupName	
id	
includeUpdates	
isolationLevel	
maxElements	
signature	
sqlSelectDistinct	
transactionAttribute	
weblogicEjbQl	
⊟ **Finders**	
value	[]
⊟ **JndiName**	
local	
remote	
⊟ *MessageDe*	
destinat	
initialCo	
name	
provider	
⊟ *MessageDe*	
messageDestinationLin	

The @Finder annotation is written to your source code. Note in **Annotations** view that all **Finder** attribute values are set to their default **UNSPECIFIED** values.

4. In **Annotations** view, set **Finder** annotation values in the value column.



Annotations

Finder - Annotation

Property	Value
Finder	
cachingName	UNSPECIFIED
comment	UNSPECIFIED
ejbQl	SELECT OBJECT(o)...
generateOn	LOCAL
groupName	UNSPECIFIED
id	LOCAL
includeUpdates	REMOTE
isolationLevel	UNSPECIFIED
maxElements	UNSPECIFIED
signature	Collection findByIt...
sqlSelectDistinct	UNSPECIFIED
transactionAttribute	UNSPECIFIED
weblogicEjbQl	UNSPECIFIED

Find Method Examples

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

- **Selecting all**. The method `Collection findAll()`, defined in the home interface of the `ItemsBean_F`, returns all records in the database table:

  ```
  @Finder(generateOn = Finder.GenerateOn.LOCAL,
      ejbQl = "SELECT OBJECT(o) from ItemsBean_F as o",
      signature = "Collection findAll()")
  ```

  Notice that an EJB QL query always uses the EJB's abstract schema name `ItemsBean_F` to reference it. This name is typically the same as the EJB's descriptive name. For more details on these names, see the @Entity Annotation. When the @Entity annotation's abstractSchemaName attribute isn't specified, the its ejbName attribute value will be used.

- **Usiing input parameters**. The method `Collection findByItemName(java.lang.String itemname)`, defined in the home interface of the `ItemsBean_F`, returns all references to bean instances matching the item name:

  ```
  @Finder(ejbQl = "SELECT OBJECT(o) from ItemsBean_F as o " +
  ```

```
    "WHERE o.itemname = ?1",
    generateOn = Finder.GenerateOn.LOCAL,
    signature = "Collection findByItemName(java.lang.String itemname)")
```

Notice that the method argument `itemname` is matched to input parameter `?1`.

- **Specifying literal values**. The method `Collection findUSManufacturers()`, defined in the home interface of the `ManufacturerBean_F`, returns all references to `ManufacturerBean_F` instances who are US manufacturers:

```
@Finder(ejbQl = "SELECT OBJECT(o) from ManufacturerBean_F as o WHERE o.usManufacturer = 1",
    generateOn = Finder.GenerateOn.LOCAL,
    signature = "Collection findUSManufacturer()")
```

- **Making relationship queries**. A `ManufacturerBean_F` and an `ItemsBean_F` 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_F`'s CMR field `manufacturer` stores a unique index to a manufacturer. The method `Collection findAllManufacturers()`, defined in the home interface of the `ManufacturerBean_F`, queries the `ItemsBean_F` 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:

```
@Finder(ejbQl = "SELECT OBJECT(m) from ItemsBean_F as o, IN(o.manufacturer) AS m",
    generateOn = Finder.GenerateOn.LOCAL,
    signature = "Collection findAllManufacturers()")
```

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

- **Getting unique records**. A `ManufacturerBean_F` and an `ItemsBean_F` 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_F`'s CMR field `manufacturer` stores a unique index to a manufacturer. The method `Collection findDistinctManufacturer()`, defined in the home interface of the `ManufacturerBean_F`, queries the `ItemsBean_F` 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:

```
@Finder(ejbQl = "SELECT DISTINCT OBJECT(m) from ItemsBean_F as o, IN(o.manufacturer) AS m",
    generateOn = Finder.GenerateOn.LOCAL,
    signature = "Collection findDistinctManufacturer()")
```

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`.

In Workshop for WebLogic, you specify a select method by adding the method's declaration to your bean class, then annotating method with the @Select annotation. As with the @Finder annotation, you use the @Select annotation's attributes to indicate query method features, such as the language to use.

The following example shows a select method declared within an `ItemsBean_S` entity bean. It uses EJB QL.

```
@Select(ejbQl = "SELECT OBJECT(o) from ItemsBean_S as o")
public abstract java.util.Collection ejbSelectAll() throws FinderException;
```

> **Note:** For more information on the @Select annotation, see weblogic.ejbgen.Select in the EJBGen Reference.

In the IDE, you can view and set @Select annotation values using the Annotations view. To do this, do as described above for adding a find method, except that your cursor should be in a method declaration.

Select Method Examples

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_S` class (but not its interfaces), returns all records in the database table:

  ```
  @Select(ejbQl = "SELECT OBJECT(o) from ItemsBean_S as o")
  public abstract java.util.Collection ejbSelectAll() throws FinderException;
  ```

- **CMP Fields**. The method `java.util.Collection ejbSelectItemNames()`, defined in the `ItemsBean_S` class, returns only the item names of all items:

  ```
  @Select(ejbQl = "SELECT o.itemname from ItemsBean_S as o")
  public abstract java.util.Collection ejbSelectItemNames()
      throws FinderException;
  ```

  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_S` class, returns only the item names of the items that match the argument `itemname`:

  ```
  @Select(ejbQl = "SELECT o.itemname from ItemsBean_S as o WHERE o.itemname = ?1")
  public abstract java.util.Collection ejbSelectNameByItemName(String itemname)
      throws FinderException;
  ```

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_S 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.

Related Topics

@Finder Annotation

@Select Annotation

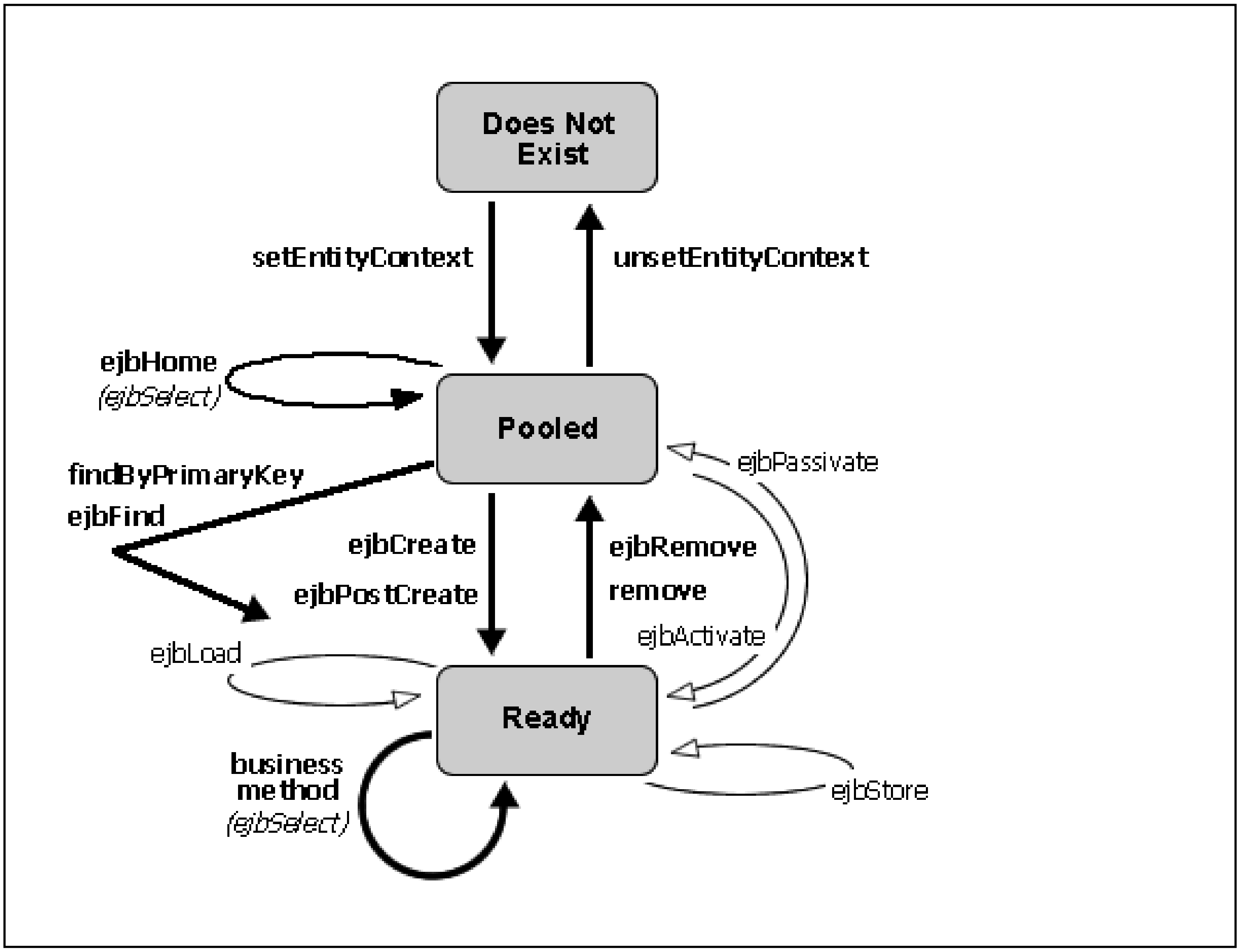
Life Cycle of an Entity Bean

When developing an entity bean you can take advantage of the bean's relationship with the container to execute logic and optimizations outside the context of the bean's core logic. As the container creates and pools an instance, assigns data to it, executes bean methods, and eventually removes the instance, the container provides opportunities for your code to execute. This topic provides an overview of an entity bean's life cycle, pointing out some of these opportunities.

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 state** . 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 @Entity Annotation).
- **Ready state**. 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:

```
@LocalRefs({
    @LocalRef(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());
    }
}
```

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.

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 by the container. 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.

- Find methods. When a find 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.

Ready State

When a bean instance is in the ready state, it represents 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.

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 becomes 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 methods 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

Entity Relationships

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
: Provides an overview of session beans.

Defining a Session Bean
: 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.

Life Cycle of a Session Bean
: Discusses the life cycle of stateful and stateless session beans.

Related Topics

Tutorial: Enterprise JavaBeans
: Provides a step-by-step guide to developing Enterprise JavaBeans.

Getting Started with Session Beans

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

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.

Developing Session Beans with Workshop for WebLogic

In Workshop for WebLogic, you develop a session bean by creating a class that extends `weblogic.ejb.GenericSessionBean` and implements `javax.ejb.SessionBean`. You annotate this class with @Session, @JndiName, and @FileGeneration annotations (and others, as needed) that specify EJB characteristics.

You can get started easily in the IDE by using the WebLogic Session Bean template. When you use the template, the IDE generates code such as the following:

```
/**
 * GenericSessionBean subclass automatically generated by
 * Workshop.
 *
 * Please complete the ejbCreate() method, add all desired business
 * methods and review the Session, JndiName and FileGeneration annotations to
 * ensure the settings match your intended use.
 */
@Session(ejbName = "MySessionBean")
@JndiName(remote = "ejb.MySessionBeanRemoteHome")
@FileGeneration(remoteClass = Constants.Bool.TRUE,
        remoteHome = Constants.Bool.TRUE,
        localClass = Constants.Bool.FALSE,
        localHome = Constants.Bool.FALSE)
public class MySessionBean
    extends GenericSessionBean
    implements SessionBean {

    private static final long serialVersionUID = 1L;

    /*
     * (non-Javadoc)
     *
     * @see weblogic.ejb.GenericSessionBean#ejbCreate()
     */
    public void ejbCreate() {
        // IMPORTANT: Add your code here
    }

    // IMPORTANT: Add business methods
}
```

Note: To use the WebLogic Session Bean template, in Workshop perspective right-click the package that will contain the bean, point to New, then click WebLogic Session Bean.

The code includes typical values for the commonly used class-level annotation attributes. The idea is to provide a starting place for your own code — for you to rewrite it with your code for methods for business logic, and so on.

Workshop for WebLogic uses these annotations to generate the interfaces and descriptor files that are required for EJB session beans. The following sections describe these session bean pieces and characteristics.

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.

Note: When creating a new EJB, a stateless bean is created by default.

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.

In Workshop for WebLogic, you specify whether a session bean is stateful or stateless by using the @Session annotation's type attribute.

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.

You can view and set the interfaces defined for a session bean through the Annotations view. With your cursor in the bean class's declaration, scroll to where the FileGeneration annotation is listed. The localHomeName and remoteHomeName attributes specify the local and remote interface names. As these are set, corresponding @FileGeneration Annotation values should be visible in your source code.

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.

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 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 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 would be used to reserve tickets for a movie.

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 each time, 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 Life Cycle of a Session Bean.

Related Topics

None.

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

The WebLogic Session Bean wizard makes it easy to start creating an session bean from scratch. If you are designing a new session bean in a WebLogic EJB project, using the wizard will create basic session bean code from a template. To use the wizard, in Workshop for WebLogic right-click your WebLogic EJB project folder, point to New, then click WebLogic Session Bean.

For step-by-step information on creating new beans, see Tutorial: Building Enterprise JavaBeans.

> **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.

Defining a Basic Session Bean

The following 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.

```
package mypackage;

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

@Session(ejbName = "PriceChecker")
@JndiName(local = "ejb.PriceCheckerLocalHome")
@FileGeneration(remoteClass = Constants.Bool.FALSE,
        remoteHome = Constants.Bool.FALSE,
        localClass = Constants.Bool.TRUE,
        localClassName = "PriceCheckerLocal",
        localHome = Constants.Bool.TRUE,
        localHomeName = "PriceCheckerLocalHome")
@EjbLocalRefs( {
    @EjbLocalRef(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);
        }
    }

    @LocalMethod()
    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 @Session annotation contains the actual name of the session bean. For stateful session beans this annotation will contain the attribute `type = Session.SessionType.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 @EjbLocalRef 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 Workshop for WebLogic, all the information needed to make a session bean is 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 @FileGeneration annotation specifies the names of the local home and business interface for the `PriceChecker` bean, and the @LocalMethod annotation on the component method specifies that the method should be defined in the local business interface.

You can view the generated JAVA files in Resource view. In that view, expand the .apt_src folder to view folders corresponding to your source packages. You'll find the generated files in these folders. You can view the CLASS files compiled from the generated files (again, in Resource view) by expanding the project's build folder.

Predefined and Callback Methods

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

- `javax.ejb.EJBLocalObject`, extended by the local interface
- `javax.ejb.EJBLocalHome`, extended by the home interface
- `javax.ejb.EJBObject`, extended by the remote interface
- `javax.ejb.EJBHome`, extended by the remote home interface

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 Life Cycle of a Session Bean.

Related Topics

Life Cycle of a Session Bean

@FileGeneration Annotation

@LocalMethod Annotation

@Session Annotation

Life Cycle of a Session Bean

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

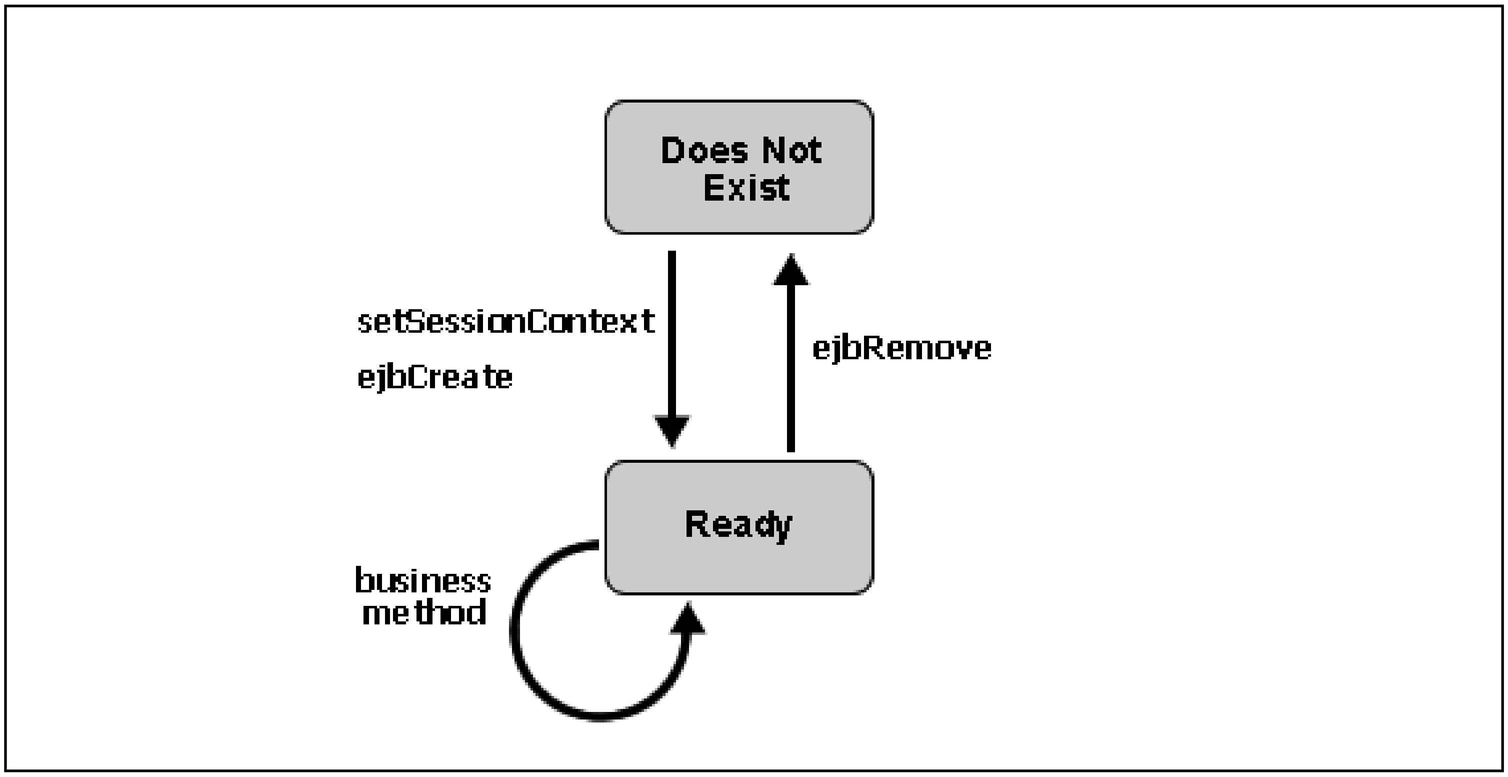
- Life Cycle of a Stateless Session Bean
- Life Cycle of a Stateful Session Bean

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 state**. 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.

Ready State

When a bean instance is in the ready state, it can service client requests; that is, execute component methods. When a client invokes a business method, the EJB container assigns an available bean instance to execute the business method. Once execution has finished, the session bean instance is ready to execute another business method.

Moving from the Ready to the Does Not Exist State

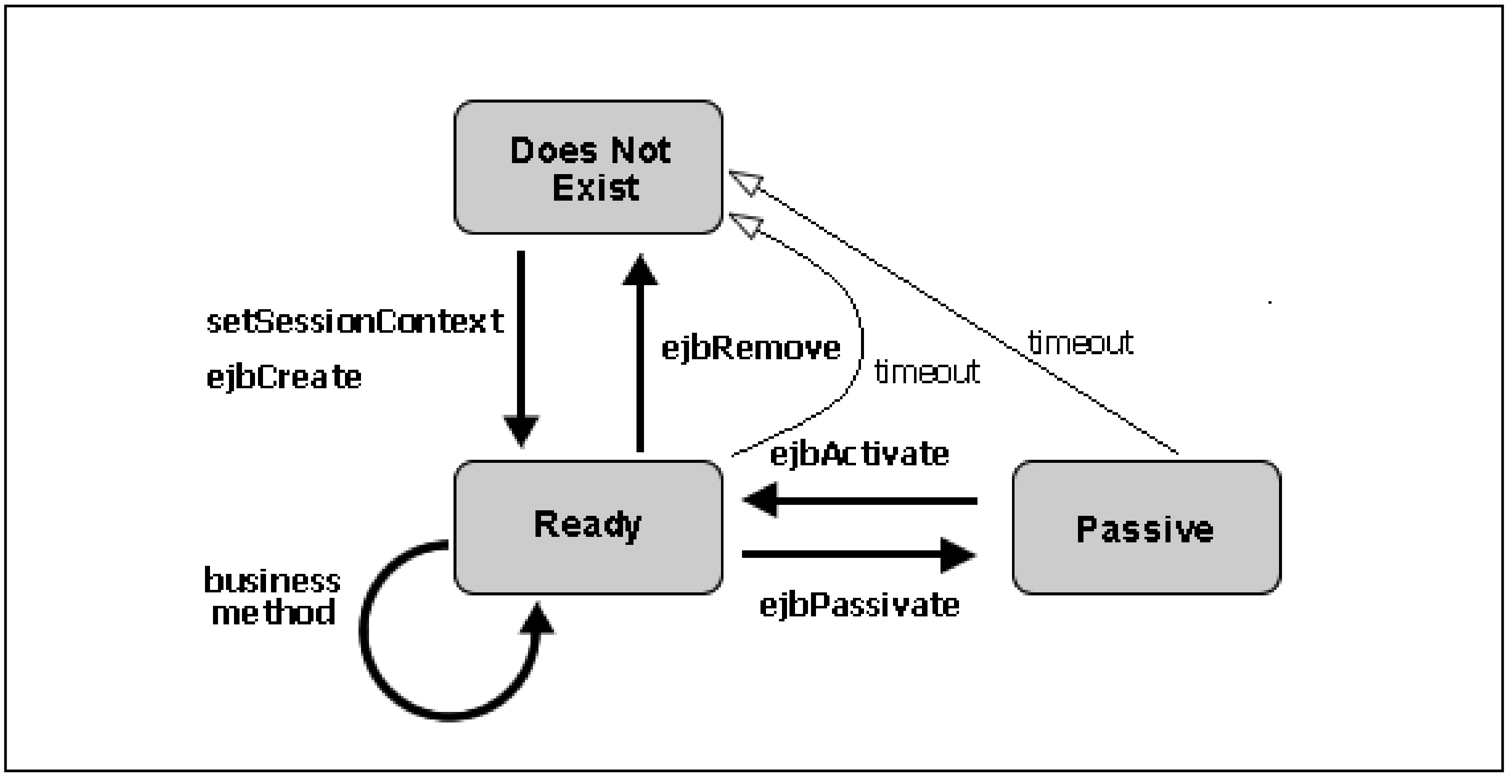
When the EJB container decides to reduce the number of session bean instances in the ready pool, it makes the bean instance ready for garbage collection. Just prior to doing this, it calls the callback method `ejbRemove`. If your session bean needs to execute some cleanup action prior to garbage collection, 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.

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 state**. A bean instance in the ready state is tied to particular client and engaged in a conversation.
- **Passive state**. 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 `idleTimeoutSeconds` attribute on the @Session annotation. The cache-type algorithm can be set using the cacheType 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 `idleTimeoutSeconds` attribute on the @Session annotation. The cache-type algorithm can be set using the cacheType attribute on the same annotation.

Related Topics

Life Cycle of an Entity Bean

@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

Related Topics

None.

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?

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 bean environment, including making enough instances available for processing and message-acknowledgement.

Developing Message-Driven Beans in Workshop for WebLogic

In Workshop for WebLogic, you develop a message-driven bean by creating a class that extends weblogic.ejb.GenericMessageDrivenBean and implements javax.ejb.MessageDrivenBean and javax.jms.MessageListener. You annotate this class with an @MessageDriven annotation that specifies EJB characteristics.

You can get started easily in the IDE by using the WebLogic Message-Driven Bean template. When you use the template, the IDE generates code such as the following:

```
/**
 * GenericMessageDrivenBean subclass automatically generated by
 * Workshop.
 *
 * Please complete the onMessage() method and review the MessageDriven
 * annotation to ensure it matches your intended use.
 */
@MessageDriven(ejbName = "MyMessageDrivenBean",
    destinationJndiName = "MyMessageDrivenBeanJndiName",
    destinationType = "javax.jms.Queue")
```

```
public class MyMessageDrivenBean
    extends GenericMessageDrivenBean
    implements MessageDrivenBean, MessageListener {

    private static final long serialVersionUID = 1L;

    /*
     * (non-Javadoc)
     *
     * @see javax.jms.MessageListener#onMessage(javax.jms.Message)
     */
    public void onMessage(Message msg) {
        // IMPORTANT: Add your code here
    }
}
```

Note: To use the WebLogic Message Driven Bean template, in Workshop perspective right-click the package that will contain the bean, point to New, then click WebLogic Message Driven Bean.

Through the @MessageDriven annotation, you specify the type and JNDI name for the JMS destination with which the message-driven bean interacts. For more on destinations, see Topics and Queues below.

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.

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 @MessageDriven 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.

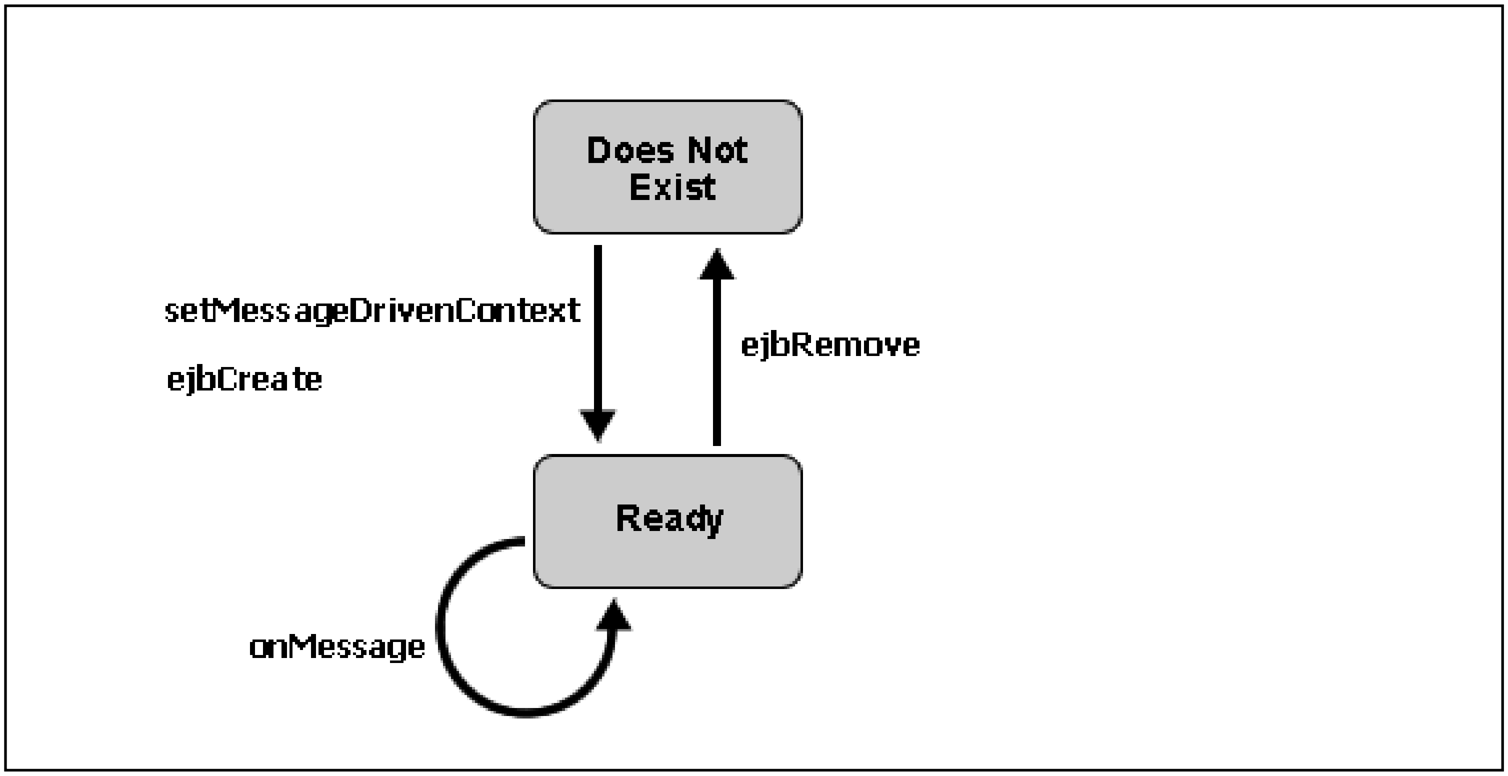
Topics and Queues

A message-driven bean listens to a particular channel, or *destination*, 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.

You specify the destination type (topic or queue) and JndiName with attributes of the @MessageDriven annotation.

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:

```
@EjbLocalRefs( {
    @EjbLocalRef(link = "Recording") })
@MessageDriven(ejbName = "Statistics",
        destinationJndiName = "jms.EJBTutorialSampleJmsQ",
        destinationType = "javax.jms.Queue")
public class StatisticsBean
    extends GenericMessageDrivenBean
    implements MessageDrivenBean, MessageListener {

    private static final long serialVersionUID = 1L;
    private RecordingHome recordingHome;

    public void ejbCreate() {
        try {
            javax.naming.Context ic = new InitialContext();
            recordingHome = (RecordingHome) ic
                    .lookup("java:/comp/env/ejb/Recording");
        } 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");

            // Placeholder logic for the rating
            Random randomGenerator = new Random();
            String rating = new Integer(randomGenerator.nextInt(5)).toString();

            // Save the rating with the recording
            Recording album = recordingHome
                    .findByPrimaryKey(new RecordingBeanPK(bandName,
                            recordingTitle));
            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

none.

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.

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. 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_M`, and subsequently deletes all records corresponding to `SimpleToken_M` from the underlying database.

```
@EjbLocalRefs( {
    @EjbLocalRef(link = "SimpleToken_M") })
@MessageDriven(defaultTransaction = MessageDriven.DefaultTransaction.
NOT_SUPPORTED,
        messageSelector = "Command = 'Delete'",
        ejbName = "DeleteViaQMD",
        destinationJndiName = "jms/SamplesAppMDBQ",
        destinationType = "javax.jms.Queue")
public class DeleteViaQMDBean
    extends GenericMessageDrivenBean
    implements MessageDrivenBean, MessageListener {

    private static final long serialVersionUID = 1L;

    private SimpleTokenHome_M tokenHome;

    public void ejbCreate() {
        try {
            javax.naming.Context ic = new InitialContext();
            tokenHome = (SimpleTokenHome_M) ic
                    .lookup("java:/comp/env/ejb/SimpleToken_M");
        } 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_M) 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; 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 @MessageDriven annotation and use the Annotations view to set the transactionType attribute to CONTAINER and the defaultTransaction attribute 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.

Related Topics

@MessageDriven Annotation