BEA WebLogic Server®

Programming WebLogic Enterprise JavaBeans, Version 3.0

Version: 10.0
Revised: May 30, 2008

Contents

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*Programming WebLogic Enterprise Java Beans, Version 3.0*.

- "Document Scope and Audience" on page 1-1
- "Guide to this Document" on page 1-2
- "Related Documentation" on page 1-2
- "Comprehensive Example for the EJB 3.0 Developer" on page 1-3
- "Release-Specific EJB 3.0 Information" on page 1-4

Document Scope and Audience

This document is a resource for software developers who develop applications that include WebLogic Server® Enterprise Java Beans (EJBs), Version 3.0.

The document mostly discusses the new EJB 3.0 programming model, in particular the use of metadata annotations to simplify development. The document briefly discusses the main differences between EJB 3.0 and 2.X for users who are familiar with programming EJB 2.X and want to know why they might want to use the new 3.0 programming model.

This document does not address EJB topics that the same between versions 2.X and 3.0, such as design considerations, EJB container architecture, deployment descriptor use, and so on. This document also does not address production phase administration, monitoring, or performance tuning. For links to WebLogic Server documentation and resources for these topics, see "Related Documentation" on page 1-2.

It is assumed that the reader is familiar with Java Platform, Enterprise Edition (Java EE) Version 5 and EJB 2.X concepts.

Guide to this Document

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide.
- Chapter 2, "Understanding Enterprise JavaBeans 3.0," provides an overview of the new EJB 3.0 features and programming model, as well as a brief description of the differences between EJB 3.0 and 2.X.
- Chapter 3, "Simple Enterprise JavaBeans 3.0 Examples," provides simple examples of programming EJBs using the new metadata annotations specified by EJB 3.0.
- Chapter 4, "Iterative Development of Enterprise JavaBeans 3.0," describes the EJB implementation process, and provides guidance for how to get an EJB up and running in WebLogic Server.
- Chapter 5, "Programming the Annotated EJB 3.0 Class," describes the requirements and typical steps when programming the EJB bean class that contains the metadata annotations.
- Chapter 6, "Using Kodo with WebLogic Server," describes how to use BEA Kodo to create entity beans. BEA Kodo is a product that provides the implementation of the Java Persistence API section of the EJB 3.0 specification, as well as other persistence-related technologies such as Java Data Objects (JDO).
- Appendix A, "EJB 3.0 Metadata Annotations Reference," provides reference information for the EJB 3.0 metadata annotations, as well as information about standard metadata annotations that are used by EJB.
- Appendix B, "Persistence Configuration Schema Reference," provides reference information about the `persistence-configuration.xml` file, which is the WebLogic-specific file used to configure BEA Kodo entities.

Related Documentation

This document contains EJB 3.0-specific development information. Additionally, it provides information only for session and message-driven beans. For completed information on general

EJB design and architecture, the EJB 2.X programming model (which is fully supported in EJB 3.0), and programming 3.0 entities, see the following documents:

- Programming Weblogic Enterprise JavaBeans (Version 2.X)
- An Introduction to the Enterprise JavaBeans 3.0 Specification (dev2dev Article)
- Enterprise JavaBeans 3.0 Specification (JSR-220)

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- *Developing Applications with WebLogic Server* is a guide to developing WebLogic Server applications.
- *Deploying Applications to WebLogic Server* is the primary source of information about deploying WebLogic Server applications in development and production environments.

Comprehensive Example for the EJB 3.0 Developer

In addition to this document and the basic examples described in Chapter 3, "Simple Enterprise JavaBeans 3.0 Examples," BEA Systems also provides a comprehensive example in the WebLogic Server distribution kit. The example illustrates EJB 3.0 in action and provides practical instructions on how to perform key EJB 3.0 development tasks. In particular, the example demonstrates usage of EJB 3.0 with:

- Java Persistence API
- Stateless Session Bean
- Message Driven Bean
- Asynchronous JavaScript based browser application.

The example uses a persistent domain model for entity EJBs.

WebLogic Server optionally install this comprehensive example in `WL_HOME\samples\server\examples\src\examples\ejb\ejb30`, where `WL_HOME` is the top-level directory of your WebLogic Server installation. On Windows, you can start the examples server, and obtain information about the samples and how to run them from the WebLogic Server Start menu.

BEA recommends that you run this example before programming your own application that uses EJB 3.0.

Release-Specific EJB 3.0 Information

For release-specific information, see these sections in *WebLogic Server Release Notes*:

- WebLogic Server Features and Changes lists new, changed, and deprecated features.
- WebLogic Server Known and Resolved Issues lists known problems by general release, as well as service pack, for all WebLogic Server APIs, including Web Services.

CHAPTER 2

Understanding Enterprise JavaBeans 3.0

These sections describe the new features and programming model of EJB 3.0.

It is assumed the reader is familiar with Java programming, Java Platform, Enterprise Edition (Java EE) Version 5, and EJB 2.x concepts and features.

Understanding EJB 3.0: New Features and Changes From EJB 2.X

Enterprise JavaBeans (EJB) is a Java Platform, Enterprise Edition (Java EE) Version 5 technology for the development and deployment of component-based business applications. Although EJB is a powerful and useful technology, the programming model in version 2.X and previous was complex and confusing, requiring the creation of multiple Java files and deployment descriptors for even the simplest of EJB. This complexity hindered the wide adoption of EJBs.

As a consequence, one of the central goals of Version 3.0 of the EJB specification is to make it much easier to program an EJB, in particular by reducing the number of required programming artifacts and introducing a set of EJB-specific metadata annotations that make programming the bean file easier and more intuitive.

Another goal of the EJB 3.0 specification was to standardize the persistence framework and reduce the complexity of the entity bean programming model and object-relational (O/R) mapping model.

Note: This document does not discuss programming 3.0 entity beans; that information is provided in the Enterprise JavaBeans 3 Persistence guide of the BEA Kodo documentation set.

The remainder of this section describes, at a high-level, how the programming model and requirements changed in EJB 3.0, as compared to version 2.X, and lists a brief description of the the new features of EJB 3.0.

For a more detailed article that discusses these issues in depth, see An Introduction to the Enterprise JavaBeans 3.0 Specification (dev2dev Article).

Changes in the EJB Programming Model and Requirements Between Versions 2.X and 3.0

The changes between EJB 2.X and 3.0 are:

- You are no longer required to create the EJB deployment descriptor files (such as `ejb-jar.xml`). You can now use metadata annotations in the bean file itself to configure metadata. You are still allowed, however, to use XML deployment descriptors if you want; in the case of conflicts, the deployment descriptor value overrides the annotation value.

  Note: In this release, WebLogic-specific features can only be configured in the WebLogic deployment descriptors, such as `weblogic-ejb-jar.xml`.

- The bean file can be a plain Java object (or POJO); it is no longer required to implement `javax.ejb.SessionBean` or `javax.ejb.MessageDrivenBean`.
- As a result of not having to implement `javax.ejb.SessionBean` or `javax.ejb.MessageDrivenBean`, the bean file no longer has to implement the lifecycle callback methods, such as `ejbCreate`, `ejbPassivate`, and so on. If, however, you want to implement these callback methods, you can name them anything you want and then annotate them with the appropriate annotation, such as `@javax.ejb.PostActivate`.
- The bean file is required to use a business interface. The bean file can either explicitly implement the business interface or it can specify it using the `@javax.ejb.Remote` or `@javax.ejb.Local` annotations.)

- The business interface is a plain Java interface (or POJI); it should not extend `javax.ejb.EJBObject` or `javax.ejb.EJBLocalObject`.
- The business interface methods may not throw `java.rmi.RemoteException` unless the business interface extends `java.rmi.Remote`.

Because the EJB 3.0 programming model is so simple, BEA no longer supports using the EJBGen tags and code-generating tool on EJB 3.0 beans. Rather, you can use this tool *only* on 2.X beans. For information, see EJBGen Reference.

New EJB 3.0 Features

- Bean files can now use metadata annotations to configure metadata, thus eliminating the need for deployment descriptors.
- The only required metadata annotation in your bean file is the one that specifies the type of EJB you are writing (`@javax.ejb.Stateless`, `@javax.ejb.Stateful`, `@javax.ejb.MessageDriven`, or `@javax.persistence.Entity`). The default value for all other annotations reflect typical and standard use of EJBs. This reduces the amount of code in your bean file in the case where you are programming a typical EJB; you only need to use additional annotations if the default values do not suit your needs.
- Bean files supports dependency injection. *Dependency injection* is when the EJB container automatically supplies (or *injects*) a variable or setter method in the bean file with a reference to another EJB or resource or another environment entry in the bean's context.
- Bean files support interceptors, which is a standard way of using aspect-oriented programming with EJB.
- You can configure two types of interceptor methods: those that intercept business methods and those that intercept lifecycle callbacks. Y
- You can configure multiple interceptor methods that execute in a chain in a particular order.
- You can configure default interceptor methods that execute for all EJBs contained in a JAR file.

WebLogic Server Value-Added EJB 3.0 Features

The following features are not part of the Enterprise JavaBeans 3.0 specification, but rather, are value-added features to further simplify the EJB 3.0 programming model:

- Automatic persistence unit deployment based on variable name.

  If you want to query and update an entity in your session bean, you annotate a variable with either the `@javax.persistence.PersistenceContext` or `@javax.persistence.PersistenceUnit` annotation; the variable is then injected with persistence unit information. You can specify the `unitName` attribute to reference a particular persistence unit in the `persistence.xml` file of the EJB JAR file; in this case, the EJB container automatically deploys the persistence unit and sets its JNDI name to the persistence unit name, as listed in the `persistence.xml` file. You can also simply name the injected variable exactly the same as the persistence unit you want injected into the variable; in this case, you no longer need to specify the `unitName` attribute, but the EJB container still deploys the persistence unit and sets its JNDI name to the persistence unit name.

  See "Invoking a 3.0 Entity" on page 5-8 for general information about invoking an entity from a session bean and "Annotations Used to Interact With Entity Beans" on page A-20 for reference information about the annotations.

- Support for vendor-specific subinterfaces when injecting an `EntityManager` from a particular persistence provider.

  When you inject a persistence context into a variable using either `@javax.persisetence.PersistenceContext` or `@javax.persistence.PersistenceUnit`, the standard data type of the injected variable is `EntityManager`. If your persistence provider provides a subinterface of the `EntityManager` (such as `KodoEntityManager` in the case of BEA Kodo) then as a convenience you can simply set the data type of the injected variable to that subinterface, rather than use the more complicated lookup mechanisms as described in the EJB 3.0 specification. For example:

  ```
  @PersistenceContext private KodoEntityManager em;
  ```

  See "Invoking a 3.0 Entity" on page 5-8 and `KodoEntityManager` for general information about `EntityManager` and the BEA Kodo-provided `KodoEntityManager`.

EJB 3.0 Examples

See Chapter 3, "Simple Enterprise JavaBeans 3.0 Examples," for simple examples of stateless and stateful session beans, interceptor classes, and how to invoke an entity. The sections in this guide reference these examples extensively.   These examples are meant to simply show how to use the new EJB 3.0 metadata annotations and programming model, rather than how to program the business code of your EJB.

For a more complex example that includes actual business code, see samples directory of the WebLogic Server installation, in particular `WL_HOME/samples/server/examples/src/examples/ejb/ejb30`, where `WL_HOME` refers to the installation directory of WebLogic Server, such as `/bea/wlserver_10.0`.

Programming 3.0 Entities

This guide describes how to program 3.0 session and message-driven EJBs, and how to invoke 3.0 entities from a session EJB. It does not describe how to actually program and configure a 3.0 entity. For details instructions on that topic, see Enterprise JavaBeans 3 Persistence in the BEA Kodo documentation.

CHAPTER 3

Simple Enterprise JavaBeans 3.0 Examples

The following sections describe simple Java examples of EJBs that use the new metadata annotation programming model:

Later procedural sections of this guide that describe how to program an EJB make reference to these examples.

Example of a Simple Stateless EJB

The following code shows a simple business interface for the `ServiceBean` stateless session EJB:

```
package examples;

/**
* Business interface of the Service stateless session EJB
*/

public interface Service {

  public void sayHelloFromServiceBean();
}
```

The code shows that the `Service` business interface has one method, `sayHelloFromServiceBean()`, that takes no parameters and returns void.

The following code shows the bean file that implements the preceding `Service` interface; the code in bold is described after the example:

```
package examples;

import static javax.ejb.TransactionAttributeType.*;

import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;

import javax.interceptor.ExcludeDefaultInterceptors;

/**
 * Bean file that implements the Service business interface.
 * Class uses following EJB 3.0 annotations:
 *   - @Stateless - specifies that the EJB is of type stateless session
 *   - @TransactionAttribute - specifies that the EJB never runs in a
 *       transaction
 *   - @ExcludeDefaultInterceptors - specifies any configured default
 *      interceptors should not be invoked for this class
 */

@Stateless
@TransactionAttribute(NEVER)
@ExcludeDefaultInterceptors
public class ServiceBean
  implements Service
{

  public void sayHelloFromServiceBean() {
    System.out.println("Hello From Service Bean!");
  }

}
```

The main points to note about the preceding code are:

- Use standard `import` statements to import the metadata annotations you use in the bean file:

```
import static javax.ejb.TransactionAttributeType.*;
import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.interceptor.ExcludeDefaultInterceptors
```

The annotations that apply only to EJB 3.0 are in the `javax.ejb` package. Annotations that can be used by other Java Platform, Enterprise Edition (Java EE) Version 5 components are in more generic packages, such `javax.interceptor` or `javax.annotation`.

- The `ServiceBean` bean file is a plain Java file that implements the `Service` business interface; it is not required to implement any EJB-specific interface. This means that the bean file does not need to implement the lifecycle methods, such as `ejbCreate` and `ejbPassivate`, that were required in the 2.X programming model.
- The class-level `@Stateless` metadata annotation specifies that the EJB is of type stateless session.
- The class-level `@TransactionAttribute(NEVER)` annotation specifies that the EJB never runs inside of a transaction.
- The class-level `@ExcludeDefaultInterceptors` annotation specifies that default interceptors, if any are defined in the `ejb-jar.xml` deployment descriptor file, should never be invoked for any method invocation of this particular EJB.

Example of a Simple Stateful EJB

The following code shows a simple business interface for the `AccountBean` stateful session EJB:

```
package examples;

/**
 * Business interface for the Account stateful session EJB.
 */

public interface Account {

  public void deposit(int amount);
  public void withdraw(int amount);
  public void sayHelloFromAccountBean();

}
```

The code shows that the `Account` business interface has three methods, `deposit`, `withdraw`, and `sayHelloFromAccountBean`.

The following code shows the bean file that implements the preceding `Account` interface; the code in bold is described after the example:

```
package examples;

import static javax.ejb.TransactionAttributeType.*;

import javax.ejb.Stateful;
import javax.ejb.TransactionAttribute;
import javax.ejb.Remote;
import javax.ejb.EJB;

import javax.annotation.PreDestroy;

import javax.interceptor.Interceptors;
import javax.interceptor.ExcludeClassInterceptors;
import javax.interceptor.InvocationContext;

/**
 * Bean file that implements the Account business interface.
 * Uses the following EJB annotations:
 *    -  @Stateful: specifies that this is a stateful session EJB
 *    -  @TransactionAttribute - specifies that this EJB never runs
 *         in a transaction
 *    -  @Remote - specifies the Remote interface for this EJB
 *    -  @EJB - specifies a dependency on the ServiceBean stateless
 *         session ejb
 *    -  @Interceptors - Specifies that the bean file is associated with an
 *         Interceptor class; by default all business methods invoke the
 *         method in the interceptor class annotated with @AroundInvoke.
 *    -  @ExcludeClassInterceptors - Specifies that the interceptor methods
 *         defined for the bean class should NOT fire for the annotated
 *         method.
 *    -  @PreDestroy - Specifies lifecycle method that is invoked when the
 *         bean is about to be destoryed by EJB container.
 *
 */

@Stateful
@TransactionAttribute(NEVER)
@Remote({examples.Account.class})
@Interceptors({examples.AuditInterceptor.class})
```

```
public class AccountBean
 implements Account
{
  private int balance = 0;

  @EJB(beanName="ServiceBean")
  private Service service;

  public void deposit(int amount) {
    balance += amount;
    System.out.println("deposited: "+amount+" balance: "+balance);
  }

  public void withdraw(int amount) {
    balance -= amount;
    System.out.println("withdrew: "+amount+" balance: "+balance);
  }

  @ExcludeClassInterceptors
  public void sayHelloFromAccountBean() {
    service.sayHelloFromServiceBean();
  }

  @PreDestroy
  public void preDestroy() {
   System.out.println("Invoking method: preDestroy()");
  }

}
```

The main points to note about the preceding code are:

- Use standard `import` statements to import the metadata annotations you use in the bean file:

  ```
  import static javax.ejb.TransactionAttributeType.*;
  import javax.ejb.Stateful;
  import javax.ejb.TransactionAttribute;
  import javax.ejb.Remote;
  import javax.ejb.EJB;

  import javax.annotation.PreDestroy;

  import javax.interceptor.Interceptors;
  import javax.interceptor.ExcludeClassInterceptors;
  ```

The annotations that apply only to EJB 3.0 are in the `javax.ejb` package. Annotations that can be used by other Java Platform, Enterprise Edition (Java EE) Version 5 components are in more generic packages, such `javax.interceptor` or `javax.annotation`.

- Import the InvocationContext class, used to maintain state between interceptors:

```
import javax.interceptor.InvocationContext;
```

- The `AccountBean` bean file is a plain Java file that implements the `Account` business interface; it is not required to implement any EJB-specific interface. This means that the bean file does not need to implement the lifecycle methods, such as `ejbCreate` and `ejbPassivate`, that were required in the 2.X programming model.
- The class-level `@Stateful` metadata annotation specifies that the EJB is of type stateful session.
- The class-level `@TransactionAttribute(NEVER)` annotation specifies that the EJB never runs inside of a transaction.
- The class-level `@Remote` annotation specifies the name of the remote interface of the EJB; in this case it is the same as the business interface, `Account`.
- The class-level `@Interceptors({examples.AuditInterceptor.class})` annotation specifies the interceptor class that is associated with the bean file. This class typically includes a business method interceptor method, as well as lifecycle callback interceptor methods. See "Example of an Interceptor Class" on page 3-7 for details about this class.
- The field-level `@EJB` annotation specifies that the annotated variable, `service`, is injected with the dependent `ServiceBean` stateless session bean context. The data type of the injected field, `Service`, is the business interface of the `ServiceBean` EJB. The following code in the `sayHelloFromAccountBean` method shows how to invoke the `sayHelloFromServiceBean` method of the dependent `ServiceBean`:

```
   service.sayHelloFromServiceBean();
```

- The method-level `@ExcludeClassInterceptors` annotation specifies that the `@AroundInvoke` method specified in the associated interceptor class (`AuditInterceptor`) should `not` be invoked for the `sayHelloFromAccountBean` method.
- The method-level `@PreDestroy` annotation specifies that the EJB container should invoke the `preDestroy` method before the container destroys an instance of the `AccountBean`. This shows how you can specify interceptor methods (for both business methods and lifecycle callbacks) in the bean file itself, in addition to using an associated interceptor class.

Example of an Interceptor Class

The following code shows an example of an interceptor class, specifically the `AuditInterceptor` class that is referenced by the preceding `AccountBean` stateful session bean with the `@Interceptors({examples.AuditInterceptor.class})` annotation; the code in bold is described after the example:

```
package examples;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

import javax.ejb.PostActivate;
import javax.ejb.PrePassivate;

/**
 * Interceptor class.  The interceptor method is annotated with the
 *  @AroundInvoke annotation.
 */

public class AuditInterceptor {

  public AuditInterceptor() {}

  @AroundInvoke
  public Object audit(InvocationContext ic) throws Exception {
    System.out.println("Invoking method: "+ic.getMethod());
    return ic.proceed();
  }

  @PostActivate
  public void postActivate(InvocationContext ic) {
    System.out.println("Invoking method: "+ic.getMethod());
  }

  @PrePassivate
  public void prePassivate(InvocationContext ic) {
    System.out.println("Invoking method: "+ic.getMethod());
  }

}
```

The main points to notice about the preceding example are:

- As usual, import the metadata annotations used in the file:

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;
import javax.ejb.PostActivate;
import javax.ejb.PrePassivate;
```

- The interceptor class is plain Java class.
- The class has an empty constructor:

  ```
  public AuditInterceptor() {}
  ```
- The method-level `@AroundInvoke` specifies the business method interceptor method. You can use this annotation only once in an interceptor class.
- The method-level `@PostActivate` and `@PrePassivate` annotations specify the methods that the EJB container should call after reactivating and before passivating the bean, respectively.

  Note: These lifecycle callback interceptor methods apply only to stateful session beans.

Example of Invoking a 3.0 Entity From A Session Bean

For an example of invoking an entity from a session bean, see the EJB 3.0 example in the distribution kit. After you have installed WebLogic Server, the example is in the following directory:

`WL_HOME``/samples/server/examples/src/examples/ejb/ejb30`

where *`WL_HOME`* refers to the directory in which you installed WebLogic Server, such as `/bea/wlserver_10.0`.

CHAPTER 4

Iterative Development of Enterprise JavaBeans 3.0

The sections that follow describe the general EJB 3.0 implementation process, and provide guidance for how to get an EJB 3.0 up and running in WebLogic Server.

For a review of WebLogic Server EJB features, see "WebLogic Server Value-Added EJB 3.0 Features" on page 2-3.

Overview of the EJB 3.0 Development Process

This section is a brief overview of the EJB 3.0 development process. It describes the key implementation tasks and associated results.

The following section mostly discusses the EJB 3.0 programming model and points out the differences between the 3.0 and 2.X programming model in only a few places. If you are an experienced EJB 2.X programmer and want the full list of differences between the two models, see "Understanding EJB 3.0: New Features and Changes From EJB 2.X" on page 2-1.

Table 4-1 EJB Development Tasks and Results

Step	Description	Result
1) Create a Source Directory	Create the directory structure for your Java source files, and optional deployment descriptors.	A directory structure on your local drive.
2) Program the EJB 3.0 Business Interface	Create the required business interface that describes your EJB.	`.java` file
3) Program the Annotated EJB Class	Create the Java file that implements the business interface and includes the EJB 3.0 metadata annotations that describe how your EJB behaves.	`.java` file
4) Optionally Program Interceptors	Optionally create the interceptor classes that describe the interceptors that intercept a business method invocation or a lifecycle callback event.	`.java` file
5) Compile Java Source	Compile source code.	`.class` file for each class and interface
6) Optionally Create and Edit Deployment Descriptors	Optionally create the EJB-specific deployment descriptors, although this step is no longer required when using the EJB 3.0 programming model.	• `ejb-jar.xml`, • `weblogic-ejb-jar.xml`, which contains elements that control WebLogic Server-specific features, and • `weblogic-cmp-jar.xml` if the bean is a container-managed persistence entity bean.

Table 4-1 EJB Development Tasks and Results

Step	Description	Result
7) Package	Package compiled classes and optional deployment descriptors for deployment. If appropriate, you can leave your files unarchived in an exploded directory.	Archive file (either an EJB JAR or Enterprise Application EAR) or equivalent exploded directory.
8) Deploy	Target the archive or application directory to desired Managed Server, or a WebLogic Server cluster, in accordance with selected staging mode.	The deployment settings for the bean are written to `EJBComponent` element in `config.xml`.

Create a Source Directory

Create a source directory where you will assemble the EJB 3.0.

BEA recommends a *split development directory structure*, which segregates source and output files in parallel directory structures. For instructions on how to set up a split directory structure and package your EJB 3.0 as an enterprise application archive (EAR), see "Overview of the Split Development Directory Environment" in *Developing Applications with WebLogic Server*.

If you prefer to package and deploy your EJB 3.0 in a JAR file, create a directory for your class files. If you are also creating deployment descriptors (which is optional but supported in the EJB 3.0 programming model) put them in a subdirectory named `META-INF`.

Listing 4-1 Directory Structure for Packaging JAR

```
myEJB/
  META-INF/
    ejb-jar.xml
    weblogic-ejb-jar.xml
    weblogic-cmp-jar.xml
  foo.class
  fooHome.class
  fooBean.class
```

Program the EJB 3.0 Business Interface

The EJB 3.0 business interface is a plain Java interface that describes the full signature of all the business methods of the EJB. For example, assume an `Account` EJB represents a client's checking account; its business interface might include three methods (`withdraw`, `deposit`, and `balance`) that clients can use to manage their bank accounts.

The EJB 3.0 business interface can extend other interfaces. In the case of message-driven beans, the business interface is typically the message-listener interface that is determined by the messaging type used by the bean, such as `javax.jms.MessageListener` in the case of JMS. The interface for a session bean has not such defining type; it can be anything that suits your business needs.

WARNING: The only requirement for an EJB 3.0 business interface is that it must *not* extend `javax.ejb.EJBObject` or `javax.ejb.EJBLocalObject`, as required in EJB 2.X.

See "Example of a Simple Stateless EJB" on page 3-1 and "Example of a Simple Stateful EJB" on page 3-3 for examples of business interfaces implemented by stateless and stateful session beans.

Business Interface Application Exceptions

When you design the business methods of your EJB, you can define an application exception in the `throws` clause of a method of the EJB's business interface. An *application exception* is an exception that you program in your bean class to alert a client of abnormal application-level conditions. For example, a `withdraw()` method in an `Account` EJB that represents a bank checking account might throw an application exception if the client tries to withdraw more money than is available in their account.

Application exceptions are different from *system exceptions*, which are thrown by the EJB container to alert the client of a system-level exception, such as the unavailability of a database management system. You should not report system-level errors in your application exceptions.

Finally, your business methods should not throw the `java.rmi.RemoteException`, even if the interface is a remote business interface, the bean class is annotated with the `@WebService` JWS annotation, or the method is annotated with `@WebMethod`. The only exception is if the business interface extends `java.rmi.Remote`. If the EJB container encounters problems at the protocol level, the container throws an `EJBException` which wraps the underlying `RemoteException`.

Note: The `@WebService` and `@WebMethod` annotations are in the `javax.jws` package; you use them to specify that your EJB implements a Web Service and that the EJB business will be exposed as public Web Service operations. For details about these annotations and programming Web Services in general, see Programming WebLogic Web Services.

Using Generics in EJBs

The EJB 3.0 programming model supports the use of generics in the business interface at the class level.

BEA recommends as a best practice that you first define a super-interface that uses the generics, and then have the actual business interface extend this super-interface with a specific data type.

The following example shows how to do this. First program the super-interface that uses generics:

```
public interface RootI<T> {

  public T getObject();
  public void updateObject(T object);
}
```

Then program the actual business interface to extend `Root*<T>` for a particular data type:

```
@Remote
public interface StatelessI extends RootI<String> { }
```

Finally, program the actual stateless session bean to implement the business interface; use the specified data type, in this case `String`, in the implementation of the methods:

```
@Stateless
public class StatelessSample implements StatelessI {
  public String getObject() {
    return null;
  }
  public void updateObject(String object) {
  }
}
```

Program the Annotated EJB Class

The EJB bean class is the main EJB programming artifact. It implements the EJB business interface and contains the EJB metadata annotations that specify semantics and requirements to the EJB container, request container services, and provide structural and configuration information to the application deployer or the container runtime.

In the 3.0 programming model, there is only one required annotation: either `@javax.ejb.Stateful`, `@javax.ejb.Stateless`, or `@javax.ejb.MessageDriven` to specify the type of EJB. Although there are many other annotations you can use to further configure your EJB, these annotations have typical default values so that you are not required to explicitly use the annotation in your bean class unless you want it to behave other than in the default manner. This programming model makes it very easy to program an EJB that exhibits typical behavior.

For additional details and examples of programming the bean class, see Chapter 5, "Programming the Annotated EJB 3.0 Class."

Optionally Program Interceptors

An interceptor is a method that intercepts the invocation of a business method or a lifecycle callback event.

You can define an interceptor method within the actual bean class, or you can program an interceptor class (distinct from the bean class itself) and associate it with the bean class using the `@javax.ejb.Interceptor` annotation.

See "Specifying Interceptors for Business Methods or Lifecycle Callback Events" on page 5-12 for information on programming the bean class to use interceptors.

Compile Java Source

Once you have written the Java source code for your EJB bean class and optional interceptor class, you must compile it into class files. Typical tools to compile include:

- `weblogic.appc`–The `weblogic.appc` Java class (or its equivalent Ant task `wlappc`) generates and compiles the classes needed to deploy EJBs and JSPs to WebLogic Server. It validates the optional deployment descriptors for compliance with the current specifications at both the individual module level and the application level. The application-level checks include checks between the application-level deployment descriptors and the individual modules as well as validation checks across the modules.

  See Building Modules and Applications Using wlappc.

- wlcompile Ant task—Invokes the `javac` compiler to compile your application's Java components in a split development directory structure.

  See Compiling Applications Using wlcompile

- `javac` —The `javac` compiler provided with the Sun Java J2SE SDK provides java compilation capabilities.

  See http://java.sun.com/docs/.

Optionally Create and Edit Deployment Descriptors

A very important aspect of the new EJB 3.0 programming model is the introduction of metadata annotations. Annotations simplify the EJB development process by allowing a developer to specify within the Java class itself how the bean behaves in the container, requests for dependency injection, and so on. Annotations are an alternative to deployment descriptors that were required by older versions (2.X and earlier) of EJB.

However, EJB 3.0 still fully supports the use of deployment descriptors, even though the standard Java Platform, Enterprise Edition (Java EE) Version 5 ones are not required. For example, you may prefer to use the old 2.X programming model, or might want to allow further customizing of the EJB at a later development or deployment stage; in these cases you can create the standard deployment descriptors in addition to, or instead of, the metadata annotations.

Deployment descriptor elements always override their annotation counterparts. For example, if you specify the `@javax.ejb.TransactionManagement(BEAN)` annotation in your bean class, but then create an `ejb-jar.xml` deployment descriptor for the EJB and set the `<transaction-type>` element to `container`, then the deployment descriptor value takes precedence and the EJB uses container-managed transaction demarcation.

Note: This version of EJB 3.0 also supports all 2.X WebLogic-specific EJB features. However, the features that are configured in the `weblogic-ejb-jar.xml` or `weblogic-cmp-rdbms-jar.xml` deployment descriptor files must continue to be configured that way for this release of EJB 3.0 because currently they do not have any annotation equivalent.

The 2.X version of Programming WebLogic Enterprise JavaBeans provides detailed information about creating and editing EJB deployment descriptors, both the Java EE standard and WebLogic-specific ones. In particular, see the following sections:

- EJB Deployment Descriptors (Overview Information)
- Editing Deployment Descriptors
- Deployment Descriptor Schema and Document Type Definitions Reference
- weblogic-ejb-jar.xml Deployment Descriptor Reference

- weblogic-cmp-jar.xml Deployment Descriptor Reference

Package

BEA recommends that you package EJBs as part of an enterprise application. For more information, see Deploying and Packaging from a Split Development Directory in *Developing Applications with WebLogic Server*.

See Packaging Considerations for EJBs with Clients in Other Applications for additional EJB-specific packaging information.

Deploy

Deploying an EJB enables WebLogic Server to serve the components of an EJB to clients. You can deploy an EJB using one of several procedures, depending on your environment and whether or not your EJB is in production. Deploying an EJB created with the 3.0 programming model is the same as deploying an EJB created with the 2.X programming model.

For general instructions on deploying WebLogic Server applications and modules, including EJBs, see *Deploying Applications to WebLogic Server*. For EJB-specific deployment issues and procedures, see Deployment Guidelines For Enterprise Java Beans in the *Programming WebLogic Enterprise JavaBeans* guide, which concentrates on the 2.X programming model.

CHAPTER 5

Programming the Annotated EJB 3.0 Class

The sections that follow describe how to program the annotated EJB 3.0 class file:

Overview of Metadata Annotations and EJB 3.0 Bean Files

The new EJB 3.0 programming model uses the JDK 5.0 metadata annotations feature in which you create an annotated EJB 3.0 bean file and then use the WebLogic compile tool `weblogic.appc` (or its Ant equivalent `wlappc`) to compile the bean file into a Java class file and generate the associated EJB artifacts, such as the required EJB interfaces and deployment descriptors.

The annotated 3.0 bean file is the core of your EJB. It contains the Java code that determines how your EJB behaves. The 3.0 bean file is an ordinary Java class file that implements an EJB business interface that outlines the business methods of your EJB. You then annotate the bean file with JDK 5.0 metadata annotations to specify the shape and characteristics of the EJB, document your EJB, and provide special services such as enhanced business-level security or special business logic during runtime.

See "Complete List of Metadata Annotations By Function" on page 5-22 for a breakdown of the annotations you can specify in a bean file, by function. These annotations include those described by the Enterprise JavaBeans 3.0 specification (JSR-220), as well as some described by the Common Annotations for the Java Platform (JSR-250). See Appendix A, "EJB 3.0 Metadata Annotations Reference," for reference information about the annotations, listed in alphabetical order.

This topic is part of the iterative development procedure for creating an EJB 3.0, described in Chapter 4, "Iterative Development of Enterprise JavaBeans 3.0."

Programming the Bean File: Requirements and Changes From 2.X

The requirements for programming the 3.0 bean class file are essentially the same as the 2.X requirements. This section briefly describes the basic mandatory requirements of the bean class, mostly for overview purposes, as well as changes in requirements between 2.X and 3.0.

See Programming WebLogic Enterprise JavaBeans for detailed information about the mandatory and optional requirements for programming the bean class.

Bean Class Requirements and Changes From 2.X

The following bullets list the new 3.0 requirements for programming a bean class, as well as the 2.X requirements that no longer apply:

- The class must specify its bean type, typically using one of the following metadata annotations, although you can also override this using a deployment descriptor:
  - `@javax.ejb.Stateless`
  - `@javax.ejb.Stateful`
  - `@javax.ejb.MessageDriven`
  - `@javax.ejb.Entity`

  Note: Programming a 3.0 entity is discussed in a separate document. See Enterprise JavaBeans 3 Persistence in the BEA Kodo documentation.

- If the bean is a session bean, the bean class must implement the bean's business interface(s) or the methods of the bean's business interface(s), if any.
- Session beans no longer needs to implement `javax.ejb.SessionBean`, which means the bean no longer needs to implement the `ejbXXX()` methods, such as `ejbCreate()`, `ejbPassivate()`, and so on.
- Stateful session beans no loner need to implement `java.io.Serializable`.

- Message-driven beans no longer need to implement `javax.ejb.MessageDrivenBean`.

The following requirements are the same as in EJB 2.X and are provided only as a brief overview:

- The class must be defined as `public`, must not be `final`, and must not be `abstract`. The class must be a top level class.
- The class must have a `public` constructor that takes no parameters.
- The class must not define the `finalize()` method.
- If the bean is message-driven, the bean class must implement, directly or indirectly, the message listener interface required by the messaging type that it supports or the methods of the message listener interface. In the case of JMS, this is the `javax.jms.MessageListener` interface.

Bean Class Method Requirements

The method requirements have not changed since EJB 2.X and are provided in this section for a brief overview only.

The requirements for programming the session bean class' methods (that implement the business interface methods) are as follows:

- The method names can be arbitrary.
- The business method must be declared as `public` and must not be `final` or `static`.
- The argument and return value types for a method must be legal types for RMI/IIOP if the method corresponds to a business method on the session bean's remote business interface or remote interface.
- The `throws` clause may define arbitrary application exceptions.

The requirements for programming the message-driven bean class' methods are as follows:

- The methods must implement the listener methods of the message listener interface.
- The methods must be declared as `public` and must not be `final` or `static`.

Programming the Bean File: Typical Steps

The following procedure describes the typical basic steps when programming the 3.0 bean file for a EJB.   The steps you follow depends, of course, on what your EJB does.

Refer to Chapter 3, "Simple Enterprise JavaBeans 3.0 Examples," for code examples of the topics discussed in the remaining sections.

1. Import the EJB 3.0 and other common annotations that will be used in your bean file. The general EJB annotations are in the `javax.ejb` package, the interceptor annotations are in the `javax.interceptor` package, the annotations to invoke a 3.0 entity are in the `javax.persistence` package, and the common annotations are in the `javax.common` or `javax.common.security` packages. For example:

   ```
   import javax.ejb.Stateless;
   import javax.ejb.TransactionAttribute;
   import javax.interceptor.ExcludeDefaultInterceptors;
   ```

2. Specify the business interface that your EJB is going to implement, as well as other standard interfaces. You can either explicitly implement the interface, or use an annotation to specify it.

   See "Specifying the Business and Other Interfaces" on page 5-5.

3. Use the required annotation to specify the type of bean you are programming (session or message-driven).

   See "Specifying the Bean Type (Stateless, Stateful, Message-Driven)" on page 5-6.

4. Optionally use dependency injection to use external resources, such as another EJB or other Java Platform, Enterprise Edition (Java EE) Version 5 object.

   See "Injecting Resource Dependency into a Variable or Setter Method" on page 5-7.

5. Optionally create an `EntityManager` object and use the entity annotations to inject entity information.

   See "Invoking a 3.0 Entity" on page 5-8.

6. Optionally program and configure business method or lifecycle callback method interceptor method. You can program the interceptor methods in the bean file itself, or in a separate Java file.

   See "Specifying Interceptors for Business Methods or Lifecycle Callback Events" on page 5-12.

7. If your business interface specifies that business methods throw application exceptions, you must program the exception class, the same as in EJB 2.X.

   See "Programming Application Exceptions" on page 5-18 for EJB 3.0 specific information.

8. Optionally specify the security roles that are allowed to invoke the EJB methods using the security-related metadata annotations.

   See "Securing Access to the EJB" on page 5-19.

9. Optionally change the default transaction configuration in which the EJB runs.

   See "Specifying Transaction Management and Attributes" on page 5-22.

Specifying the Business and Other Interfaces

When using the EJB 3.0 programming model to program a bean, you are required to specify a business interface.

There are two ways you can specify the business interface for the EJB bean class:

- By explicitly implementing the business interface, using the `implements` Java keyword.
- By using metadata annotations (such as `javax.ejb.Local` and `javax.ejb.Remote`) to specify the business interface. In this case, the bean class does not need to explicitly implement the business interface.

Typically, if an EJB bean class implements an interface, it is assumed to be the business interface of the EJB. Additionally, the business interface is assumed to be the local interface unless you explicitly denote it as the remote interface, either by using the `javax.ejb.Remote` annotation or updating the appropriate EJB deployment descriptor. You can specify the `javax.ejb.Remote` annotation. as well as the `javax.ejb.Local` annotation, in either the business interface itself, or the bean class that implements the interface.

A bean class can have more than one interface. In this case (excluding the interfaces listed below), you must specify the business interface of the EJB by explicitly using the `javax.ejb.Local` or `javax.ejb.Remote` annotations in either the business interface itself, the bean class that implements the business interface, or the appropriate deployment descriptor.

The following interfaces are excluded when determining whether the bean class has more than one interface:

- `java.io.Serializable`
- `java.io.Externalizable`
- any of the interfaces defined by `the javax.ejb` package

The following code snippet shows how to specify the business interface of a bean class by explicitly implementing the interface:

```
public class ServiceBean
```

```
  implements Service
```

For the full example, see "Example of a Simple Stateless EJB" on page 3-1

Specifying the Bean Type (Stateless, Stateful, Message-Driven)

There is only one required metadata annotation in a 3.0 bean class: an annotation that specifies the type of bean you are programing. You must specify one, and only one, of the following:

- `@javax.ejb.Stateless`—Specifies that you are programming a stateless session bean.
- `@javax.ejb.Stateful`—Specifies that you are programming a stateful session bean.
- `@javax.ejb.MessageDriven`—Specifies that you are programming a message-driven bean.
- `@javax.ejb.Entity`—Specifies that you are programming an entity bean.

  Note: Programming a 3.0 entity is discussed in a separate document. See Enterprise JavaBeans 3 Persistence in the BEA Kodo documentation.

Although not required, you can specify attributes of the annotations to further describe the bean type. For example, you can set the following attributes for all bean types:

- `name`—Name of the bean class; the default value is the unqualified bean class name.
- `mappedName`—Product-specific name of the bean.
- `description`—Description of what the bean does.

If you are programming a message-driven bean, then you can specify the following optional attributes:

- `messageListenerInterface`—Specifies the message listener interface, if you haven't explicitly implemented it or if the bean implements additional interfaces.
- `activationConfig`—Specifies an array of activation configuration name-value pairs that configure the bean in its operational environment.

The following code snippet shows how to specify that a bean is a stateless session bean:

```
@Stateless
public class ServiceBean
  implements Service
```

For the full example, see "Example of a Simple Stateless EJB" on page 3-1.

Injecting Resource Dependency into a Variable or Setter Method

Dependency injection is when the EJB container automatically supplies (or *injects*) a bean's variable or setter method with a reference to a resource or another environment entry in the bean's context. Dependency injection is simply an easier-to-program alternative to using the `javax.ejb.EJBContext` interface or JNDI APIs to look up resources.

You specify dependency injection by annotating a variable or setter method with one of the following annotations, depending on the type of resource you want to inject:

- `@javax.ejb.EJB`—Specifies a dependency on another EJB.
- `@javax.annotation.Resource`—Specifies a dependency on an external resource, such as a JDBC datasource or a JMS destination or connection factory.

  Note: This annotation is not specific to EJB; rather, it is part of the common set of metadata annotations used by many different types of Java EE components.

Both annotations have an equivalent grouping annotation to specify a dependency on multiple resources (`@javax.ejb.EJBs` and `@javax.annotation.Resources`).

Although not required, you can specify attributes to these dependency annotations to explicitly describe the dependent resource. The amount of information you need to specify depends upon its usage context and how much information the EJB container can infer from that context. See "javax.ejb.EJB" on page A-4 and "javax.annotation.Resource" on page A-27 for detailed information on the attributes and when you should specify them.

The following code snippet shows how to use the `@javax.ejb.EJB` annotation to inject a dependency on an EJB into a variable; only the relevant parts of the bean file are shown:

```
package examples;

import javax.ejb.EJB;

...

@Stateful
public class AccountBean
 implements Account

{
  @EJB(beanName="ServiceBean")
  private Service service;
...
  public void sayHelloFromAccountBean() {
    service.sayHelloFromServiceBean();
  }
```

In the preceding example, the private variable `service` is annotated with the `@javax.ejb.EJB` annotation, which makes reference to the EJB with a bean name of `ServiceBean`. The data type of the service variable is `Service`, which is the business interface implemented by the `ServiceBean` bean class. As soon as the EJB container creates the `AccountBean` EJB, the container injects a reference to `ServiceBean` into the `service` variable; the variable then has direct access to all the business methods of `SessionBean`, as shown in the `sayHelloFromAccountBean` method implementation in which the `sayHelloFromServiceBean` method is invoked.

Invoking a 3.0 Entity

This section describes how to invoke and update a 3.0 entity from within a session bean.

Note: It is assumed in this section that you have already programmed the entity, as well as configured the database resources that support the entity. For details on that topic, see Enterprise JavaBeans 3 Persistence in the BEA Kodo documentation.

An *entity* is a persistent object that represents datastore records; typically an instance of an entity represents a single row of a database table. Entities make it easy to query and update information in a persistent store from within another Java EE component, such as a session bean. A `Person` entity, for example, might include `name`, `address`, and age `fields`, each of which correspond to the columns of a table in a database. Using an `javax.persistence.EntityManager` object to access and manage the entities, you can easily retrieve a `Person` record, based on either their unique id or by using a SQL query, and then change the information and automatically commit the information to the underlying datastore.

The following sections describe the typical programming tasks you perform in your session bean to interact with entities:

- "Injecting Persistence Context Using Metadata Annotations" on page 5-9
- "Finding an Entity Using the EntityManager API" on page 5-10

- "Creating and Updating an Entity Using EntityManager" on page 5-11

Injecting Persistence Context Using Metadata Annotations

In your session bean, use the following metadata annotations inject entity information into a variable:

- `@javax.persistence.PersistenceContext`—Injects a persistence context into a variable of data type `javax.persistence.EntityManager`. A *persistence context* is simply a set of entities such that, for any persistent identity, there is a unique entity instance. The `persistence.xml` file defines and names the persistence contexts available to a session bean.
- `@javax.persistence.PersistenceContexts`—Specifies a set of multiple persistence contexts.
- `@javax.persistence.PersistenceUnit`—Injects a persistence context into a variable of data type `javax.persistence.EntityManagerFactory`.
- `@javax.persistence.PersistenceUnits`—Specifies a set of multiple persistence contexts.

The `@PersistenceContext` and `@PersistenceUnit` annotations perform a similar function: inject persistence context information into a variable; the main difference is the data type of the instance into which you inject the information. If you prefer to have full control over the lifecycle of the `EntityManager` in your session bean, then use `@PersistenceUnit` to inject into an `EntityManagerFactory` instance, and then write the code to manually create an `EntityManager` and later destroy when you are done, to release resources. If you prefer that the EJB container manage the lifecycle of the `EntityManager`, then use the `@PersistenceContext` annotation to inject directly into an `EntityManager`.

The following example shows how to inject a persistence context into the variable `em` of data type `EntityManager`; relevant code is shown in bold:

```
package examples;

import javax.ejb.Stateless;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;

@Stateless
public class ServiceBean
  implements Service
```

```
{
  @PersistenceContext private EntityManager em;
...
```

Finding an Entity Using the EntityManager API

Once you have instantiated an `EntityManager` object, you can use its methods to interact with the entities in the persistence context. This section discusses the methods used to identify and manage the lifecycle of an entity; see EntityManager in the BEA Kodo documentation for additional uses of the `EntityManager`, such as transaction management, caching, and so on.

Note: For clarity, this section assumes that the entities are configured such that they represent actual rows in a database table.

Use the `EntityManager.find()` method to find a row in a table based on its primary key. The `find` method takes two parameters: the entity class that you are querying, such as `Person.class`, and the primary key value for the particular row you want to retrieve. Once you retrieve the row, you can use standard `getXXX` methods to get particular properties of the entity. The following code snippet shows how to retrieve a `Person` with whose primary key value is `10`, and then get their address:

```
public List<Person> findPerson () {

   Person p = em.find(Person.class, 10);
   Address a = p.getAddress();

    Query q = em.createQuery("select p from Person p where p.name = :name");
   q.setParameter("name", "Patrick");
   List<Person> l = (List<Person>) q.getResultList();

   return l;

  }
```

The preceding example also shows how to use the `EntityManager.createQuery()` method to create a `Query` object that contains a custom SQL query; by contrast, the `EntityManager.find()` method allows you to query using only the table's primary key. In the example, the table is queried for all `Persons` whose first name is `Patrick`; the resulting set of rows populates the `List<Person>` object and is returned to the `findPerson()` invoker.

Creating and Updating an Entity Using EntityManager

To create a new entity instance (and thus add a new row to the database), use the `EntityManager.persist` method, as shown in the following code snippet

```
  @TransactionAttribute(REQUIRED)

  public Person createNewPerson(String name, int age) {

    Person p = new Person(name, age);

    em.persist(p); // register the new object with the database

    Address a = new Address();

    p.setAddress(a);

    em.persist(a); // depending on how things are configured, this may or
may not be required

    return p;

  }
```

Note: Whenever you create or update an entity, you must be in a transaction, which is why the `@TransactionAttribute` annotation in the preceding example is set to `REQUIRED`.

The preceding example shows how to create a new Person, based on parameters passed to the `createNewPerson` method, and then call the EntityManager.persist method to automatically add the row to the database table.

The preceding example also shows how to update the newly-created `Person` entity (and thus new table row) with an `Address` by using the `setAddress()` entity method. Depending on the cascade configuration of the `Person` entity, the second `persist()` call may not be necessary; this is because the call to the `setAddress()` method might have automatically triggered an update to the database. For more information about cascading operations, see Cascade Type in the BEA Kodo documentation.

If you use the `EntityManager.find()` method to find an entity instance, and then use a `setXXX` method to change a property of the entity, the database is automatically updated and you do not need to explicitly call the `EntityManager.persist()` method, as shown in the following code snippet:

```
  @TransactionAttribute(REQUIRED)

  public Person changePerson(int id, int newAge) {

    Person p = em.find(Person.class, id);
    p.setAge(newAge);
    return p;
  }
```

In the preceding example, the call to the `Person.setAge()` method automatically triggered an update to the appropriate row in the database table.

Finally, you can use the `EntityManager.merge()` method to quickly and easily update a row in the database table based on an update to an entity made by a client, as shown in the following example:

```
  @TransactionAttribute(REQUIRED)
  public Person applyOfflineChanges(Person pDTO) {
    return em.merge(pDTO);
  }
```

In the example, the `applyOfflineChanges()` method is a business method of the session bean that takes as a parameter a `Person`, which has been previously created by the session bean client. When you pass this `Person` to the `EntityManager.merge()` method, the EJB container automatically finds the existing row in the database table and automatically updates the row with the new data. The `merge()` method then returns a copy of this updated row.

Specifying Interceptors for Business Methods or Lifecycle Callback Events

An interceptor is a method that intercepts a business method invocation or a lifecycle callback event. There are two types of interceptors: those that intercept business methods and those that intercept lifecycle callback methods.

Interceptors can be specified for session and message-driven beans.

You can program an interceptor method inside the bean class itself, or in a separate interceptor class which you then associate with the bean class with the `@javax.interceptor.Interceptors` annotation. You can create multiple interceptor methods that execute as a chain in a particular order.

Interceptor instances may hold state. The lifecycle of an interceptor instance is the same as that of the bean instance with which it is associated. Interceptors can invoke JNDI, JDBC, JMS, other enterprise beans, and the EntityManager. Interceptor methods share the JNDI name space of the bean for which they are invoked. Programming restrictions that apply to enterprise bean components to apply to interceptors as well.

Interceptors are configured using metadata annotations in the `javax.interceptor` package, as described in later sections.

The following topics discuss how to actually program interceptors for your bean class:

- Specifying Business or Lifecycle Interceptors: Typical Steps
- Programming the Interceptor Class
- Programming Business Method Interceptor Methods
- Programming Lifecycle Callback Interceptor Methods
- Specifying Default Interceptor Methods
- Saving State Across Interceptors With the InvocationContext API

Specifying Business or Lifecycle Interceptors: Typical Steps

The following procedure provides the typical steps to specify and program interceptors for your bean class.

See "Example of a Simple Stateful EJB" on page 3-3 for an example of specifying interceptors and "Example of an Interceptor Class" on page 3-7 for an example of programming an interceptor class.

1. Decide whether interceptor methods are programmed in bean class or in a separate interceptor class.

2. If you decide to program the interceptor methods in a separate interceptor class

   a. Program the class, as described in "Programming the Interceptor Class" on page 5-14.

   b. In your bean class, use the `@javax.interceptor.Interceptors` annotation to associate the interceptor class with the bean class. The method in the interceptor class annotated with the `@javax.interceptor.AroundInvoke` annotation then becomes a business method interceptor method of the bean class. Similarly, the methods annotated with the lifecycle callback annotations become the lifecycle callback interceptor methods of the bean class.

      You can specify any number of interceptor classes for a given bean class—the order in which they execute is the order in which they are listed in the annotation. If you specify the interceptor class at the class-level, the interceptor methods apply to all appropriate bean class methods. If you specify the interceptor class at the method-level, the interceptor methods apply to only the annotated method.

3. In the bean class or interceptor class (wherever you are programming the interceptor methods), program business method interceptor methods, as described in "Programming Business Method Interceptor Methods" on page 5-14.
4. In the bean class or interceptor class (wherever you are programming the interceptor methods), program lifecycle callback interceptor methods, as described in "Programming Business Method Interceptor Methods" on page 5-14.
5. In the bean class, optionally annotate methods with the `@javax.interceptor.ExcludeClassInterceptors` annotation to exclude any interceptors defined at the class-level.
6. In the bean class, optionally annotate the class or methods with the `@javax.interceptor.ExcludeDefaultInterceptors` annotation to exclude any default interceptors that you might define later. Default interceptors are configured in the `ejb-jar.xml` deployment descriptor, and apply to all EJBs in the JAR file, unless you explicitly use the annotation to exclude them.
7. Optionally specify default interceptors for the entire EJB JAR file, as described in "Specifying Default Interceptor Methods" on page 5-17.

Programming the Interceptor Class

The interceptor class is a plain Java class that includes the interceptor annotations to specify which methods intercept business methods and which intercept lifecycle callback methods.

Interceptor classes support dependency injection, which is performed when the interceptor class instance is created, using the naming context of the associated enterprise bean.

You must include a public no-argument constructor.

You can have any number of methods in the interceptor class, but restrictions apply as to how many methods can be annotated with the interceptor annotations, as described in the following sections.

For an example, see "Example of an Interceptor Class" on page 3-7.

Programming Business Method Interceptor Methods

You specify business method interceptor methods by annotating them with the `@AroundInvoke` annotation.

An interceptor class or bean class can have only one method annotated with `@AroundInvoke`. To specify that multiple interceptor methods execute for a given business method, you must associate multiple interceptor classes with the bean file, in addition to optionally specifying an interceptor method in the bean file itself. The order in which the interceptor methods execute is the order in which the associated interceptor classes are listed in the `@Interceptor` annotation. Interceptor methods in the bean class itself execute after those defined in the interceptor classes.

You cannot annotate a business method itself with the `@AroundInvoke` annotation.

The signature of an `@AroundInvoke` method must be:

```
Object <METHOD>(InvocationContext) throws Exception
```

The method annotated with the `@AroundInvoke` annotation must always call `InvocationContext.proceed()` or neither the business method will be invoked nor any subsequent `@AroundInvoke` methods. See "Saving State Across Interceptors With the InvocationContext API" on page 5-17 for additional information about the `InvocationContext` API.

Business method interceptor method invocations occur within the same transaction and security context as the business method for which they are invoked. Business method interceptor methods may throw runtime exceptions or application exceptions that are allowed in the `throws` clause of the business method.

For an example, see "Example of an Interceptor Class" on page 3-7.

Programming Lifecycle Callback Interceptor Methods

You specify a method to be a lifecycle callback interceptor method so that it can receive notification of life cycle events from the EJB container. Life cycle events include creation, passivation, and destruction of the bean instance.

You can name the lifecycle callback interceptor method anything you want; this is different from the EJB 2.X programming model in which you had to name the methods `ejbCreate()`, `ejbPassivate()`, and so on.

You use the following lifecycle interceptor annotations to specify that a method is a lifecycle callback interceptor method:

- `@javax.ejb.PrePassivate`—Specifies the method that the EJB container notifies when it is about to passivate a stateful session bean.
- `@javax.ejb.PostActivate`—Specifies the method that the EJB container notifies right after it has reactivated a stateful session bean.

- `@javax.annotation.PostConstruct`—Specifies the method that the EJB container notifies before it invokes the first business method and after it has done dependency injection. You typically apply this annotation to the method that performs initialization.

  Note: This annotation is in the `javax.annotation` package, rather than `javax.ejb`.

- `@javax.annotation.PreDestroy`—Specifies the method that the EJB container notifies right before it destroys the bean instance. You typically apply this annotation to the method that release resources that the bean class has been holding.

  Note: This annotation is in the `javax.annotation` package, rather than `javax.ejb`.

You use the preceding annotations the same way, whether the annotated method is in the bean class or in a separate interceptor class. You can annotate the same method with more than one annotation.

You can also specify any subset or combination of lifecycle callback annotations in the bean class or in an associated interceptor class. However, the same callback annotation may not be specified more than once in a given class. If you do specify a callback annotation more than once in a given class, the EJB will not deploy.

To specify that multiple interceptor methods execute for a given lifecycle callback event, you must associate multiple interceptor classes with the bean file, in addition to optionally specifying the lifecycle callback interceptor method in the bean file itself. The order in which the interceptor methods execute is the order in which the associated classes are listed in the `@Interceptor` annotation. Interceptor methods in the bean class itself execute after those defined in the interceptor classes.

The signature of the annotated methods depends on where the method is defined:

- Lifecycle callback methods defined on a bean class have the following signature:

  ```
  void <METHOD>()
  ```

- Lifecycle callback methods defined on an interceptor class have the following signature:

  ```
  void <METHOD>(InvocationContext)
  ```

See "Saving State Across Interceptors With the InvocationContext API" on page 5-17 for additional information about the `InvocationContext` API.

See "javax.ejb.PostActivate" on page A-10, "javax.ejb.PrePassivate" on page A-11, "javax.annotation.PostConstruct" on page A-26, and "javax.annotation.PreDestroy" on page A-27 for additional requirements when programming the lifecycle interceptor class.

For an example, see "Example of an Interceptor Class" on page 3-7.

Specifying Default Interceptor Methods

Default interceptor methods apply to *all* components in a particular EJB JAR file or exploded directory, and thus can only be configured in the `ejb-jar.xml` deployment descriptor file and not with metadata annotations, which apply to a particular EJB.

The EJB container invokes default interceptor methods, if any, before *all* other interceptors defined for an EJB (both business and lifecycle). If you do not want the EJB container to invoke the default interceptors for a particular EJB, specify the class-level `@javax.interceptor.ExcludeDefaultInterceptors` annotation in the bean file.

In the `ejb-jar.xml` file, use the `<interceptor-binding>` child element of `<assembly-descriptor>` to specify default interceptors. In particular, set the `<ejb-name>` child element to `*`, which means the class applies to all EJBs, and then the `<interceptor-class>` child element to the name of the interceptor class.

The following snippet from an `ejb-jar.xml` file shows how to specify the default interceptor class `org.mycompany.DefaultIC`:

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar version="3.0"
     xmlns="http://java.sun.com/xml/ns/javaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">

...

   <assembly-descriptor>

...

      <interceptor-binding>
        <ejb-name>*</ejb-name>
        <interceptor-class>org.mycompany.DefaultIC</interceptor-class>
      </interceptors>

   </assembly-descriptor>

</ejb-jar>
```

Saving State Across Interceptors With the InvocationContext API

Use the `javax.interceptor.InvocationContext` API to pass state information between the interceptors that execute for a given business method or lifecycle callback. The EJB Container passes the same `InvocationContext` instance to each interceptor method, so you can, for example save information when the first business method interceptor method executes, and then retrieve this information for all subsequent interceptor methods that execute for this business method. The `InvocationContext` instance is not shared between business method or lifecycle callback invocations.

All interceptor methods must have an `InvocationContext` parameter. You can then use the methods of the `InvocationContext` interface to get and set context information. The `InvocationContext` interface is shown below:

```
public interface InvocationContext {
    public Object getBean();
    public Method getMethod();
    public Object[] getParameters();
    public void setParameters(Object[]);
    public java.util.Map getContextData();
    public Object proceed() throws Exception;
}
```

The `getBean` method returns the bean instance. The `getMethod` method returns the name of the business method for which the interceptor method was invoked; in the case of lifecycle callback interceptor methods, `getMethod` returns null.

The `proceed` method causes the invocation of the next interceptor method in the chain, or the business method itself if called from the last `@AroundInvoke` interceptor method.

For an example of using `InvocationContext`, see “Example of an Interceptor Class” on page 3-7.

Programming Application Exceptions

If you specified in the business interface that a method throws an application method, then you must program the exception as a separate class from the bean class.

Use the `@javax.ejb.ApplicationException` annotation to specify that an exception class is an application exception thrown by a business method of the EJB. The EJB container reports the exception directly to the client in the event of the application error.

Use the `rollback` Boolean attribute of the `@ApplicationException` annotation to specify whether the application error causes the current transaction to be rolled back. By default, the current transaction is not rolled back in event of the error.

You can annotate both checked and unchecked exceptions with this annotation.

The following `ProcessingException.java` file shows how to use the `@ApplicationException` annotation to specify that an exception class is an application exception thrown by one of the business methods of the EJB:

```
package examples;

import javax.ejb.ApplicationException;

/**
 * Application exception class thrown when there was a processing error
 * with a business method of the EJB.  Annotated with the
 * @ApplicationException annotation.
 */

@ApplicationException()
public class ProcessingException extends Exception {

  /**
   * Catches exceptions without a specified string
   *
   */
  public ProcessingException() {}

  /**
   * Constructs the appropriate exception with the specified string
   *
   * @param message           Exception message
   */
  public ProcessingException(String message) {super(message);}
}
```

Securing Access to the EJB

By default, any user can invoke the public methods of an EJB. If you want to restrict access to the EJB, you can use the following security-related annotations to specify the roles that are allowed to invoke all, or a subset, of the methods:

- `javax.annotation.security.DeclareRoles`—Explicitly lists the security roles that will be used to secure the EJB.
- `javax.annotation.security.RolesAllowed`—Specifies the security roles that are allowed to invoke all the methods of the EJB (when specified at the class-level) or a particular method (when specified at the method-level.)

- `javax.annotation.security.DenyAll`—Specifies that the annotated method can not be invoked by any role.
- `javax.annotation.security.PermitAll`—Specifies that the annotated method can be invoked by all roles.
- `javax.annotation.security.RunAs`—Specifies the role which runs the EJB. By default, the EJB runs as the user who actually invokes it.

The preceding annotations can be used with many Java EE components that allow metadata annotations, not just EJB 3.0.

You create security roles and map users to roles using the WebLogic Server Administration Console to update your security realm. For details, see Manage Security Roles.

The following example shows a simple stateless session EJB that uses all of the security-related annotations; the code in bold is discussed after the example:

```
package examples;

import javax.ejb.Stateless;

import javax.annotation.security.DeclareRoles;
import javax.annotation.security.PermitAll;
import javax.annotation.security.DenyAll;
import javax.annotation.security.RolesAllowed;
import javax.annotation.security.RunAs;

/**
 * Bean file that implements the Service business interface.
 */

@Stateless
@DeclareRoles( { "admin", "hr" } )
@RunAs ("admin")

public class ServiceBean
  implements Service
{

  @RolesAllowed ( {"admin", "hr"} )
  public void sayHelloRestricted() {
    System.out.println("Only some roles can invoke this method.");
  }
  @DenyAll
  public void sayHelloSecret() {
    System.out.println("No one can invoke this method.");
  }

  @PermitAll
  public void sayHelloPublic() {
    System.out.println("Everyone can invoke this method.");
  }

}
```

The main points to note about the preceding example are:

- Import the security-related metadata annotations:

  ```
  import javax.annotation.security.DeclareRoles;
  import javax.annotation.security.PermitAll;
  import javax.annotation.security.DenyAll;
  import javax.annotation.security.RolesAllowed;
  import javax.annotation.security.RunAs;
  ```

- The class-level `@DeclareRoles` annotation explicitly specifies that the `admin` and `hr` security roles will later be used to secure some or all of the methods. This annotation is not required; any security role referenced in, for example, the `@RolesReferenced` annotation is implicitly declared. However, explicitly declaring the security roles makes your code easier to read and understand.
- The class-level `@RunAs` annotation specifies that, regardless of the user who actually invokes a particular method of the EJB, the EJB container runs the method as the `admin` role, assuming, of course, that the original user is allowed to invoke the method.
- The `@RolesAllowed` annotation on the `sayHelloRestricted` method specifies that only users mapped to the `admin` and `hr` roles are allowed to invoke the method.
- The `@DenyAll` annotation on the `sayHelloSecret` method specifies that no one is allowed to invoke the method.
- The `@PermitAll` annotation on the `sayHelloPublic` method specifies that all users mapped to any roles are allowed to invoke the method.

Specifying Transaction Management and Attributes

By default, the EJB container invokes a business method within a transaction context. Additionally, the EJB container itself decides whether to commit or rollback a transaction; this is called container-managed transaction demarcation.

You can change this default behavior by using the following annotations in your bean file:

- `javax.ejb.TransactionManagement`—Specifies whether the EJB container or the bean file manages the demarcation of transactions. If you specify that the bean file manages it, then you must program transaction management in your bean file, typically using the Java Transaction API (JTA).
- `javax.ejb.TransactionAttribute`—Specifies whether the EJB container invokes methods within a transaction.

For an example of using the `javax.ejb.TransactionAttribute` annotation, see "Example of a Simple Stateful EJB" on page 3-3.

Complete List of Metadata Annotations By Function

Appendix A, "EJB 3.0 Metadata Annotations Reference," provides full reference information about the EJB 3.0 metadata annotations in alphabetical order. The tables in this section group the annotations based on what task they perform.

Annotations to Specify the Bean Type

Table 5-1 Annotations to Specify the Bean Type

Annotation	Description
`@javax.ejb.Stateless`	Specifies that the bean class is a stateless session bean.
`@javax.ejb.Stateful`	Specifies that the bean class is a stateful session bean.
`@javax.ejb.Init`	Specifies the correspondence of a stateful session bean class method with a `create<METHOD>` method for an adapted EJB 2.1 EJBHome and/or EJBLocalHome client view.
`@javax.ejb.Remove`	Specifies a `remove` method of a stateful session bean.

Table 5-1 Annotations to Specify the Bean Type

Annotation	Description
`@javax.ejb.MessageDriven`	Specifies that the bean class is a message-driven bean.
`@javax.ejb.ActivationConfigProperty`	Specifies properties used to configure a message-driven bean in its operational environment.

Annotations to Specify the Local or Remote Interfaces

Table 5-2 Annotations to Specify the Local or Remote Interfaces

Annotation	Description
`@javax.ejb.Local`	Specifies a local interface of the bean.
`@javax.ejb.Remote`	Specifies a remote interface of the bean.

Annotations to Support EJB 2.X Client View

Table 5-3 Annotations to Support EJB 2.X Client View

Annotation	Description
`@javax.ejb.LocalHome`	Specifies a local home interface of the bean.
`@javax.ejb.RemoteHome`	Specifies a remote home interface of the bean.

Annotations to Invoke a 3.0 Entity Bean

Table 5-4 Annotations to Invoke a 3.0 Entity Bean

Annotation	Description
`@javax.persistence.PersistenceContext`	Specifies a dependency on an `EntityManager` persistence context.
`@javax.persistence.PersistenceContexts`	Specifies one or more `PersistenceContext` annotations.
`@javax.persistence.PersistenceUnit`	Specifies a dependency on an `EntityManagerFactory`.
`@javax.persistence.PersistenceUnits`	Specifies one or more `PersistenceUnit` annotations.

Transaction-Related Annotations

Table 5-5 Transaction-Related Annotations

Annotation	Description
`@javax.ejb.TransactionManagement`	Specifies the transaction management demarcation type (container- or bean-managed)
`@javax.ejb.TransactionAttribute`	Specifies whether a business method is invoked within the context of a transaction.

Annotations to Specify Interceptors

Table 5-6 Annotations to Specify Interceptors

Annotation	Description
`@javax.interceptor.Interceptors`	Specifies the list of interceptor classes associated with a bean class or method.
`@javax.interceptor.AroundInvoke`	Specifies an interceptor method.

Table 5-6 Annotations to Specify Interceptors

Annotation	Description
`@javax.interceptor.ExcludeClassInterceptors`	Specifies that, when the annotated method is invoked, the class-level interceptors should *not* invoke.
`@javax.interceptor.ExcludeDefaultInterceptors`	Specifies that, when the annotated method is invoked, the default interceptors should *not* invoke.

Annotations to Specify Lifecycle Callbacks

Table 5-7 Annotations to Specify Lifecycle Callbacks

Annotation	Description
`@javax.ejb.PostActivate`	Designates a method to receive a callback after a stateful session bean has been activated.
`@javax.ejb.PrePassivate`	Designates a method to receive a callback before a stateful session bean is passivated.
`@javax.annotation.PostConstruct`	Specifies the method that needs to be executed after dependency injection is done to perform any initialization.
`@javax.annotation.PreDestroy`	Specifies a method to be a callback notification to signal that the instance is in the process of being removed by the container

Security-Related Annotations

The following metadata annotations are not specific to EJB 3.0, but rather, are general security-related annotations in the `javax.annotation.security` package.

Table 5-8 Security-Related Annotations

Annotation	Description
`@javax.annotation.security.DeclareRoles`	Specifies the references to security roles in the bean class.
`@javax.annotation.security.RolesAllowed`	Specifies the list of security roles that are allowed to invoke the bean's business methods.
`@javax.annotation.security.PermitAll`	Specifies that all security roles are allowed to invoke the method.
`@javax.annotation.security.DenyAll`	Specifies that no security roles are allowed to invoke the method.
`@javax.annotation.security.RunAs`	Specifies the security role which the method is run as.

Context Dependency Annotations

Table 5-9 Context Dependency Annotations

Annotation	Description
`@javax.ejb.EJB`	Specifies a dependency to an EJB business interface or home interface.
`@javax.ejb.EJBs`	Specifies one or more `@EJB` annotations.
`@javax.annotation.Resource`	Specifies a dependency on an external resource in the bean's environment.
`@javax.annotation.Resources`	Specifies one or more `@Resource` annotations.

Timeout and Exceptions Annotations

Table 5-10 Timeout and Exception Annotations

Annotation	Description
`@javax.ejb.Timeout`	Specifies the timeout method of the bean class.
`@javax.ejb.ApplicationException`	Specifies that an exception is an application exception and should be reported to the client directly.

CHAPTER 6

Using Kodo with WebLogic Server

This chapter provides an overview of developing, deploying, and configuring a BEA Kodo application using WebLogic Server: The following topics are covered:

Overview of Kodo

BEA Kodo an implementation of Sun Microsystem's Java Persistence API (JPA) specification and Java Data Objects (JDO) specification for transparent data objects. BEA Kodo is available as a stand-alone product and is integrated within WebLogic Server.

This chapter describes how to implement an application using JPA or JDO in WebLogic Server. Within WebLogic Server, the JPA and JDO implementations are part of WebLogic Servers overall Enterprise Java Bean 3.0 persistence implementation.

For general information on creating an application using JPA and JDO, see the *Kodo Developer's Guide*.

Creating a Kodo Application

The first step in implementing a BEA Kodo application on WebLogic Server is to write your application's code. The following resources provide general information on writing an application that uses BEA Kodo to manage persistence of your data:

- *BEA Kodo JPA Tutorials*
- *BEA Kodo JDO Tutorials*

Once you are familiar with the steps involved in creating applications using BEA Kodo and have created your application, the following sections describe how to deploy and configure your application using WebLogic Server.

Using Different Kodo Versions

If you choose to use a different version of Kodo than the one provided by default within WebLogic Server, you must use the FilteringClassLoader to exclude the Kodo and OpenJPA libraries from the component classpath.

The following example shows how to exclude these class libraries using `weblogic-application.xml`:

```
<prefer-application-packages>
  <package-name>org.apache.openjpa.*</package-name>
  <package-name>kodo.*</package-name>
</prefer-application-packages>
```

For more information on filtering classloaders, see "Understanding WebLogic Server Application Classloading" in *Developing Applications With WebLogic Server*.

Configuring Persistence

The following sections describe how to configure persistence.

- "Editing the Configuration Property Files" on page 6-3
- "Configuring a Plugin" on page 6-4

Editing the Configuration Property Files

BEA Kodo uses two XML files, listed in the following table, to define configuration properties.

Table 6-1 Persistence Configuration Files

Configuration File	Description
`persistence.xml`	Kodo configuration parameters defined by the JPA functional specifications. This file is required. The XML schema for structuring this configuration is available at: `http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd`. For more information, see "Chapter 6. Persistence" in the *Kodo Developers Guide*.
`persistence-configuration.xml`	Configuration parameters that are specific to BEA Kodo. This file is not required when deploying an application. If specified, you must still Use of persistence If you do not include `persistence-configuration.xml` in your deployment, WebLogic Server will create reasonable defaults for each configuration parameter. The XML schema for structuring this configuration is available at: `http://www.bea.com/ns/weblogic/persistence-configuration/10.0/persistence-configuration.xsd`. **Note:** The `weblogic.jar` file must be in the `CLASSPATH` when using the `persistent-configuration.xml` file in the Java SE environment.

Edit the contents of the configuration files as required to configure persistence. The files should be available as resources in the META-INF directory of the root of the persistence unit. For container environments, the root of a persistence unit may be one of the following:

- EJB-JAR file
- `WEB-INF/classes` directory of a WAR file
- JAR file in the `WEB-INF/lib` directory of a WAR file
- JAR file in the root of the EAR
- JAR file in the EAR library directory

- Application client jar file

Configuring a Plugin

Because Kodo is a highly customizable environment, many configuration properties relate to the creation and configuration of system plugins. Plugin properties have a syntax very similar to that of Java annotations. They allow you to specify both what class to use for the plugin and how to configure the public fields or bean properties of the instantiated plugin instance.

Essentially, plugins are defined via a series of properties using name/value pairs. The following example shows how a plugin is defined within `persistence.xml`:

```
<property name='myplugin.DataCache'
value='com.bea.MyDataCache(CacheSize=1000, RemoteHost='CacheServer)'>
```

Deploying a Kodo Application

You are ready to deploy your application on WebLogic Server, once you have completed the following tasks:

- Created a Kodo application
- Configured `persistence.xml` and `persistence-configuration.xml`
- Created and archive for your application (.ear or .war)

Once your application is configured, a Kodo application is deployed just like any other application. For complete information on deploying applications, see Deploying Applications on BEA WebLogic Server.

Configuring a Kodo Application

Once you have deployed your Kodo application, you can alter the configuration paramters defined in `persistence.xml` and `persistence-configuration.xml`. Many Kodo configuration parameters can be configured from the WebLogic Server Administration Console.

If your deployed application has defined a persistence unit within `persistence.xml`, you can access configuration from within the Administration Console using the following:

1. Select **Deployments**
2. Select the name of the module containing a persistence unit that you want to configure.

3. Select the **Configuration** tab.
4. Select the **Persistence** tab.
5. From the list of persistence units, select the one that you want to configure.

From here, you can access all of the Kodo persistence parameters that can be edited from the Administration Console.

Note: You cannot create a new persistence unit from the Administration Console. To create a new persistence unit, you must edit `persistence.xml` manually.

APPENDIX A

EJB 3.0 Metadata Annotations Reference

The following topics provide reference information about the EJB 3.0 metadata annotations:

Overview of EJB 3.0 Annotations

The new EJB 3.0 programming model uses the JDK 5.0 metadata annotations feature in which you create an annotated EJB 3.0 bean file and then use the WebLogic compile tool `weblogic.appc` (or its Ant equivalent `wlappc`) to compile the bean file into a Java class file and generate the associated EJB artifacts, such as the required EJB interfaces and deployment descriptors.

The following sections provide reference information for the metadata annotations you can specify in the EJB bean file. Some of the annotations are in the `javax.ejb` package, and are thus specific to EJBs; others are more common and are used by other Java Platform, Enterprise Edition (Java EE) Version 5 components, and are thus in more generic packages, such as `javax.annotation`.

Annotations for Stateless, Stateful, and Message-Driven Beans

This section provides reference information for the following annotations:

javax.ejb.ActivationConfigProperty

Description

Target: Any

Specifies properties used to configure a message-driven bean in its operational environment. This may include information about message acknowledgement modes, message selectors, expected destination or endpoint types, and so on.

This annotation is used only as a value to the `activationConfig` attribute of the `@javax.ejb.MessageDriven` annotation.

Attributes

Table A-1 Attributes of the javax.ejb.ActivationConfigProperty Annotation

Name	Description	Data Type	Required?
propertyName	Specifies the name of the activation property.	String	Yes
propertyValue	Specifies the value of the activation property.	String	Yes

javax.ejb.ApplicationException

Description

Target: Class

Specifies that an exception is an application exception and that it should be reported to the client application directly, or unwrapped.

This annotation can be applied to both checked and unchecked exceptions.

Attributes

Table A-2 Attributes of the javax.ejb.ApplicationException Annotation

Name	Description	Data Type	Required?
rollback	Specifies whether the EJB container should rollback the transaction, if the bean is currently being invoked inside of one, if the exception is thrown. Valid values for this attribute are `true` and `false`. Default value is `false`, or the transaction should *not* be rolled back.	boolean	No.

javax.ejb.EJB

Description

Target: Class, Method, Field

Specifies a dependency or reference to an EJB business or home interface.

You annotate a bean's instance variable with the `@EJB` annotation to specify a dependence on another EJB. WebLogic Server automatically initializes the annotated variable with the reference to the EJB on which it depends; this is also called *dependency injection*. This initialization occurs before any of the bean's business methods are invoked and after the bean's `EJBContext` is set.

You can also annotate a setter method in the bean class; in this case WebLogic Server uses the setter method itself when performing dependency injection. This is an alternative to instance variable dependency injection.

If you apply the annotation to a class, the annotation declares the EJB that the bean will look up at runtime.

Whether using variable or setter method injection, WebLogic Server determines the name of the referenced EJB by either the name or data type of the annotated instance variable or setter method parameter. If there is any ambiguity, you should use the `beanName` or `mappedName` attributes of the `@EJB` annotation to explicitly name the dependent EJB.

Attributes

Table A-3 Attributes of the javax.ejb.EJB Annotation

Name	Description	Data Type	Required?
name	Specifies the name by which the referenced EJB is to be looked up in the environment.	String	No
beanInterface	Specifies the interface type of the referenced EJB (either a business or home interface). Default value for this attribute is `Object.class`	Class	No

Table A-3 Attributes of the javax.ejb.EJB Annotation

Name	Description	Data Type	Required?
beanName	Specifies the name of the referenced EJB. This attribute corresponds to the `name` element of the `@Stateless` or `@Stateful` annotation in the referenced EJB, which by default is the unqualified name of the referenced bean class. This attribute is most useful when multiple session beans in an EJB JAR file implement the same interface, because the name of each bean must be unique.	String	No
mappedName	Specifies the global JNDI name of the referenced EJB. For example: `mappedName="bank.Account"` specifies that the referenced EJB has a global JNDI name of `bank.Account` and is deployed in the WebLogic Server JNDI tree. **Note:** EJBs that use mapped names may not be portable.	String	No
description	Describes the EJB reference.	String	No

javax.ejb.EJBs

Description

Target: Class

Specifies an array of `@javax.ejb.EJB` annotations.

Attribute

Table A-4 Attribute of the javax.ejb.EJBs Annotation

Name	Description	Data Type	Required?
value	Specifies the array of `@javax.ejb.EJB` annotations	EJB[]	No

javax.ejb.Init

Description

Target: Method

Specifies the correspondence of a method in the bean class with a `createMETHOD` method for an adapted EJB 2.1 `EJBHome` or `EJBLocalHome` client view.

This annotation is used only in conjunction with stateful session beans, or those that have been annotated with the `@javax.ejb.Stateful` class-level annotation,

The return type of a method annotated with the `@javax.ejb.Init` annotation must be `void`, and its parameter types must be exactly the same as those of the referenced `createMETHOD` method or methods.

The `@Init` annotation is required only for stateful session beans that provide a `Remote-Home` or `LocalHome` interface. You must specify the name of the adapted create method of the Home or LocalHome interface, using the `value` attribute, if there is any ambiguity.

Attributes

Table A-5 Attributes of the javax.ejb.Init Annotation

Name	Description	Data Type	Required?
value	Specifies the name of the corresponding `createMETHOD` method. This attribute is required only when the `@Init` annotation is used to associate an adapted `Home` interface of a stateful session bean that has more than one `create<METHOD>` method.	String	No.

javax.ejb.Local

Description

Target: Class

Specifies the local interface or interfaces of a session bean. The local interface exposes business logic to local clients—those running in the same application as the EJB. It defines the business methods a local client can call.

You are required to specify this annotation if your bean class implements more than a single interface, not including the following:

- `java.io.Serializable`
- `java.io.Externalizable`
- `javax.ejb.*`

This annotation applies only to stateless or stateful session beans.

Attributes

Table A-6 Attributes of the javax.ejb.Local Annotation

Name	Description	Data Type	Required?
value	Specifies the list of local interfaces as an array of classes. You are required to specify this attribute only if your bean class implements more than a single interface, not including the following: • `java.io.Serializable` • `java.io.Externalizable` • `javax.ejb.*`	Class[]	No.

javax.ejb.LocalHome

Description

Target: Class

Specifies the local home interface of the bean class.

The local home interface provides methods that local clients—those running in the same application as the EJB—can use to create, remove, and in the case of an entity bean, find instances of the bean. The local home interface also has *home methods*—business logic that is not specific to a particular bean instance.

This attribute applies only to stateless and stateful session beans.

You typically specify this attribute only if you are going to provide an adapted EJB 2.1 component view of the EJB 3.0 bean. You can also use this annotation with bean classes that have been written to the EJB 2.1 APIs.

Attributes

Table A-7 Attributes of the javax.ejb.LocalHome Annotation

Name	Description	Data Type	Required?
value	Specifies the local home class.	Class	Yes.

javax.ejb.MessageDriven

Description

Target: Class

Specifies that the Enterprise JavaBean is a message-driven bean.

Attributes

Table A-8 Attributes of the javax.ejb.MessageDriven Annotation

Name	Description	Data Type	Required?
name	Specifies the name of the message-driven bean. If you do not specify this attribute, the default value is the unqualified name of the bean class.	String	No.
messageListenerInterface	Specifies the message-listener interface of the bean class. You must specify this attribute if the bean class does not explicitly implement the message-listener interface, or if the bean class implements more than one interface other than `java.io.Serializable`, `java.io.Externalizable`, or any of the interfaces in the `javax.ejb` package. The default value for this attribute is `Object.class`.	Class	No.
activationConfig	Specifies the configuration of the message-driven bean in its operational environment. This may include information about message acknowledgement modes, message selectors, expected destination or endpoint types, and so on. You specify activation configuration information using an Array of `@javax.ejb.ActivationConfigProperty` annotation, specify the property name and value.	ActivationConfigProperty[]	No.

Table A-8 Attributes of the javax.ejb.MessageDriven Annotation

Name	Description	Data Type	Required?
mappedName	Specifies the product-specific name to which the message-driven bean should be mapped. You can also use this attribute to specify the JNDI name of the message destination of this message-driven bean. For example: `mappedName="my.Queue"` specifies that this message-driven bean is associated with a JMS queue, whose JNDI name is `my.Queue` and is deployed in the WebLogic Server JNDI tree. **Note:** If you specify this attribute, the message-driven bean may not be portable.	String	No
description	Specifies a description of the message-driven bean.	String	No

javax.ejb.PostActivate

Description

Target: Method

Specifies the lifecycle callback method that signals that the EJB container has just reactivated the bean instance.

This annotation applies only to stateful session beans. Because the EJB container automatically maintains the conversational state of a stateful session bean instance when it is passivated, you do not need to specify this annotation for most stateful session beans. You only need to use this annotation, along with its partner `@PrePassivate`, if you want to allow your stateful session bean to maintain the open resources that need to be closed prior to a bean instance's passivation and then reopened during the bean instance's activation.

Only one method in the bean class can be annotated with this annotation. If you annotate more than one method with this annotations, the EJB will not deploy.

The method annotated with `@PostActivate` must follow these requirements:

- The return type of the method must be `void`.
- The method must not throw a checked exception.

- The method may be `public`, `protected`, `package private` or `private`.
- The method must not be `static`.
- The method must not be `final`.

This annotation does not have any attributes.

javax.ejb.PrePassivate

Description

Target: Method

Specifies the lifecycle callback method that signals that the EJB container is about to passivate the bean instance.

This annotation applies only to stateful session beans. Because the EJB container automatically maintains the conversational state of a stateful session bean instance when it is passivated, you do not need to specify this annotation for most stateful session beans. You only need to use this annotation, along with its partner `@PostActivate`, if you want to allow your stateful session bean to maintain the open resources that need to be closed prior to a bean instance's passivation and then reopened during the bean instance's activation.

Only one method in the bean class can be annotated with this annotation. If you annotate more than one method with this annotations, the EJB will not deploy.

The method annotated with `@PrePassivate` must follow these requirements:

- The return type of the method must be `void`.
- The method must not throw a checked exception.
- The method may be `public`, `protected`, `package private` or `private`.
- The method must not be `static`.
- The method must not be `final`.

This annotation does not have any attributes.

javax.ejb.Remote

Description

Target: Class

Specifies the remote interface or interfaces of a session bean. The remote interface exposes business logic to remote clients—clients running in a separate application from the EJB. It defines the business methods a remote client can call.

This annotation applies only to stateless or stateful session beans.

Attributes

Table A-9 Attributes of the javax.ejb.Remote Annotation

Name	Description	Data Type	Required?
value	Specifies the list of remote interfaces as an array of classes. You are required to specify this attribute only if your bean class implements more than a single interface, not including the following: • `java.io.Serializable` • `java.io.Externalizable` • `javax.ejb.*`	Class[]	No.

javax.ejb.RemoteHome

Description

Target: Class

Specifies the remote home interface of the bean class.

The remote home interface provides methods that remote clients—those running in a separate application from the EJB—can use to create, remove, and find instances of the bean.

This attribute applies only to stateless and stateful session beans.

You typically specify this attribute only if you are going to provide an adapted EJB 2.1 component view of the EJB 3.0 bean. You can also use this annotation with bean classes that have been written to the EJB 2.1 APIs.

Attributes

Table A-10 Attributes of the javax.ejb.RemoteHome Annotation

Name	Description	Data Type	Required?
value	Specifies the remote home class.	Class	Yes.

javax.ejb.Remove

Description

Target: Method

Use the `@javax.ejb.Remove` annotation to denote a remove method of a stateful session bean.

When the method completes, the EJB container will invoke the method annotated with the `@javax.annotation.PreDestroy` annotation, if any, and then destroy the stateful session bean.

Attributes

Table A-11 Attributes of the javax.ejb.Remove Annotation

Name	Description	Data Type	Required?
retainIfException	Specifies that the container should not remove the stateful session bean if the annotated method terminates abnormally with an application exception. Valid values are `true` and `false`. Default value is `false`.	boolean	No.

javax.ejb.Stateful

Description

Target: Class

Specifies that the Enterprise JavaBean is a stateful session bean.

Attributes

Table A-12 Attributes of the javax.ejb.Stateful Annotation

Name	Description	Data Type	Required?
name	Specifies the name of the stateful session bean. If you do not specify this attribute, the default value is the unqualified name of the bean class.	String	No.
mappedName	Specifies the product-specific name to which the stateful session bean should be mapped. You can also use this attribute to specify the JNDI name of this stateful session bean. WebLogic Server uses the value of the `mappedName` attribute when creating the bean's global JNDI name. In particular, the JNDI name will be: *`mappedName#name_of_businessInterface`* where *`name_of_businessInterface`* is the fully qualified name of the business interface of this session bean. For example, if you specify `mappedName="bank"` and the fully qualified name of the business interface is `com.CheckingAccount`, then the JNDI of the business interface is `bank#com.CheckingAccount`. **Note:** If you specify this attribute, the stateful session bean may not be portable.	String	No.
description	Describes the stateful session bean.	String	No.

javax.ejb.Stateless

Description

Target: Class

Specifies that the Enterprise JavaBean is a stateless session bean.

Attributes

Table A-13 Attributes of the javax.ejb.Stateless Annotation

Name	Description	Data Type	Required?
name	Specifies the name of the stateless session bean. If you do not specify this attribute, the default value is the unqualified name of the bean class.	String	No.
mappedName	Specifies the product-specific name to which the stateless session bean should be mapped. You can also use this attribute to specify the JNDI name of this stateless session bean. WebLogic Server uses the value of the `mappedName` attribute when creating the bean's global JNDI name. In particular, the JNDI name will be: *`mappedName#name_of_businessInterface`* where *`name_of_businessInterface`* is the fully qualified name of the business interface of this session bean. For example, if you specify `mappedName="bank"` and the fully qualified name of the business interface is `com.CheckingAccount`, then the JNDI of the business interface is `bank#com.CheckingAccount`. **Note:** If you specify this attribute, the stateless session bean may not be portable.	String	No.
description	Describes the stateless session bean.	String	No.

javax.ejb.Timeout

Description

Target: Method

Specifies the timeout method of the bean class.

This annotation makes it easy to program an EJB timer service in your bean class. The EJB timer service is an EJB-container provided service that allows you to create timers that schedule callbacks to occur when a timer object expires.

Previous to EJB 3.0, your bean class was required to implement `javax.ejb.TimedObject` if you wanted to program the timer service. Additionally, your bean class had to include a method with the exact name `ejbTimeout`. These requirements are relaxed in Version 3.0 of EJB. You no longer are required to implement the `javax.ejb.TimedObject` interface, and you can name your timeout method anything you want, as long as you annotate it with the `@Timeout` annotation. You can, however, continue to use the pre-3.0 way of programming the timer service if you want.

For details, see Programming the EJB Timer Service.

This annotation does not have any attributes.

javax.ejb.TransactionAttribute

Description

Target: Class, Method

Specifies whether the EJB container invokes an EJB business method within a transaction context.

WARNING: If you specify this annotation, you are also required to use the `@TransactionManagement` annotation to specify container-managed transaction demarcation.

You can specify this annotation on either the bean class, or a particular method of the class that is also a method of the business interface. If specified at the bean class, the annotation applies to all applicable business interface methods of the class. If specified for a particular method, the annotation applies to that method only. If the annotation is specified at both the class and the method level, the method value overrides if the two disagree.

If you do not specify the `@TransactionAttribute` annotation in your bean class, and the bean uses container managed transaction demarcation, the semantics of the REQUIRED transaction attribute are assumed.

Attributes

Table A-14 Attributes of the javax.ejb.TransactionAttribute Annotation

Name	Description	Data Type	Required?
value	Specifies how the EJB container manages the transaction boundaries when invoking a business method. For details about these values, see the description of the trans-attribute element in the Container-Managed Transactions Elements table. Valid values for this attribute are: • `TransactionAttributeType.MANDATORY` • `TransactionAttributeType.REQUIRED` • `TransactionAttributeType.REQUIRED_NEW` • `TransactionAttributeType.SUPPORTS` • `TransactionAttributeType.NOT_SUPPORTED` • `TransactionAttributeType.NEVER` Default value is `TransactionAttributeType.REQUIRED`.	TransactionAttributeType	No.

javax.ejb.TransactionManagement

Description

Target: Class

Specifies the transaction management demarcation type of the session bean or message-driven bean.

A transaction is a unit of work that changes application state—whether on disk, in memory or in a database—that, once started, is completed entirely, or not at all. Transactions can be demarcated—started, and ended with a commit or rollback—by the EJB container, by bean code, or by client code. This annotation specifies whether the EJB container or the user-written bean code manages the demarcation of a transaction.

If you do not specify this annotation in your bean class, it is assumed that the bean has container-managed transaction demarcation.

For additional information about transactions, see Transaction Design and Management Options.

Attributes

Table A-15 Attributes of the javax.ejb.TransactionManagement Annotation

Name	Description	Data Type	Required?
value	Specifies the transaction management demarcation type used by the bean class. Valid values for this attribute are: • `TransactionManagementType.CONTAINER` • `TransactionManagementType.BEAN` Default value is `TransactionManagementType.CONTAINER`	TransacationManagementType	No.

Annotations Used to Configure Interceptors

This section provides reference information for the following annotations:

- "javax.interceptor.AroundInvoke" on page A-18
- "javax.interceptor.ExcludeClassInterceptors" on page A-19
- "javax.interceptor.ExcludeDefaultInterceptors" on page A-19
- "javax.interceptor.Interceptors" on page A-19

javax.interceptor.AroundInvoke

Description

Target: Method

Specifies the business method interceptor for either a bean class or an interceptor class.

You can annotate only *one* method in the bean class or interceptor class with the `@AroundInvoke` annotation; the method cannot be a business method of the bean class.

This annotation does not have any attributes.

javax.interceptor.ExcludeClassInterceptors

Description

Target: Method

Specifies that any class-level interceptors should not be invoked for the annotated method. This does not include default interceptors, whose invocation are excluded only with the `@ExcludeDefaultInterceptors` annotation.

This annotation does not have any attributes.

javax.interceptor.ExcludeDefaultInterceptors

Description

Target: Class, Method

Specifies that any defined default interceptors (which can be specified only in the EJB deployment descriptors, and not with annotations) should not be invoked.

If defined at the class-level, the default interceptors are never invoked for any of the bean's business methods. If defined at the method-level, the default interceptors are never invoked for the particular business method, but they are invoked for all other business methods that do not have the `@ExludeDefaultInterceptors` annotation.

This annotation does not include any attributes.

javax.interceptor.Interceptors

Description

Target: Class, Method

Specifies the interceptor classes that are associated with the bean class or method. An interceptor class is a class—distinct from the bean class itself—whose methods are invoked in response to business method invocations and/or lifecycle events on the bean.

The interceptor class can include both an business interceptor method (annotated with the `@javax.interceptor.AroundInvoke` annotation) and lifecycle callback methods (annotated with the `@javax.annotation.PostConstruct`, `@javax.annotation.PreDestroy`, `@javax.ejb.PostActivate`, and `@javax.ejb.PrePassivate` annotations).

Any number of interceptor classes may be defined for a bean class. If more than one interceptor class is defined, they are invoked in the order they are specified in the annotation.

If the annotation is specified at the class-level, the interceptors apply to all business methods of the EJB. If specified at the method-level, the interceptors apply to just that method. You can specify the same interceptor class to more than one method of the bean class. By default, method-level interceptors are invoked after all applicable interceptors (default interceptors, class-level interceptors, and so on).

Attributes

Table A-16 Attributes of the javax.interceptor.Interceptors Annotation

Name	Description	Data Type	Required?
value	Specifies the array of interceptor classes. If there is more than one interceptor class in the array, the order in which they are listed defines the order in which they are invoked.	Class[]	Yes

Annotations Used to Interact With Entity Beans

This section provides reference information about the following annotations:

- "javax.persistence.PersistenceContext" on page A-20
- "javax.persistence.PersistenceContexts" on page A-22
- "javax.persistence.PersistenceUnit" on page A-24
- "javax.persistence.PersistenceUnits" on page A-25

javax.persistence.PersistenceContext

Description

Target: Class, Method, Field

Specifies a dependency on a container-managed `EntityManager` persistence context.

You use this annotation to interact with a 3.0 entity bean, typically by performing dependency injection into an `EntityManager` instance.

The `EntityManager` interface defines the methods that are used to interact with the persistence context. A persistence context is a set of entity instances; an entity is a lightweight persistent domain object. The `EntityManager` API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

Attributes

javax.persistence.PersistenceContexts

Table A-17 Attributes of the javax.pesistence.PersistenceContext Annotation

Name	Description	Data Type	Required?
name	Specifies the name by which the `EntityManager` and its persistence unit are to be known within the context of the session or message-driven bean. You only need to specify this attribute if you use a JNDI lookup to obtain an `EntityManager`; if you use dependency injection, then you do not need to specify this attribute.	String	No.

Table A-17 Attributes of the javax.pesistence.PersistenceContext Annotation

Name	Description	Data Type	Required?
unitName	Specifies the name of the persistence unit. If you specify a value for this attribute that is the same as the name of a persistence unit in the `persistence.xml` file, the EJB container automatically deploys the persistence unit and sets its JNDI name to its persistence unit name. Similarly, if you do not specify this attribute, but the name of the variable into which you are injecting the persistence context information is the same as the name of a persistence unit in the `persistence.xml` file, then the EJB container again automatically deploys the persistence unit with its JNDI name equal to its unit name. **Note:** The `persistence.xml` file is an XML file, located in the `META-INF` directory of the EJB JAR file, that specifies the database used with the entity beans and specifies the default behavior of the `EntityManager`. You must specify this attribute if there is more than one persistence unit within the referencing scope.	String	No.
type	Specifies whether the lifetime of the persistence context is scoped to a transaction or whether it extends beyond that of a single transaction. Valid values for this attribute are: • `PersistenceContextType.TRANSACTION` • `PersistenceContextType.EXTENDED` Default value is `PersistenceContextType.TRANSACTION`.	`PersistenceContextType`	No.

Description

Target: Class

Specifies an array of `@javax.persistence.PersistencContext` annotations.

Attributes

Table A-18 Attributes of the javax.pesistence.PersistenceContexts Annotation

Name	Description	Data Type	Required?
value	Specifies the array of `@javax.persistence.PersistencContext` annotations.	`PersistenceContext[]`	Yes.

javax.persistence.PersistenceUnit

Description

Target: Class, Method, Field

Specifies a dependency on an `EntityManagerFactory` object.

You use this annotation to interact with a 3.0 entity bean, typically by performing dependency injection into an `EntityManagerFactory` instance. You can then use the `EntityManagerFactory` to create one or more `EntityManager` instances. This annotation is similar to the `@PersistenceContext` annotation, except that it gives you more control over the life of the `EntityManager` because you create and destroy it yourself, rather than let the EJB container do it for you.

The `EntityManager` interface defines the methods that are used to interact with the persistence context. A persistence context is a set of entity instances; an entity is a lightweight persistent domain object. The `EntityManager` API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

Attributes

Table A-19 Attributes of the javax.pesistence.PersistenceUnit Annotation

Name	Description	Data Type	Required?
name	Specifies the name by which the `EntityManagerFactory` is to be known within the context of the session or message-driven bean You are not required to specify this attribute if you use dependency injection, only if you also use JNDI to look up information.	String	No
unitName	Refers to the name of the persistence unit as defined in the `persistence.xml` file. This file is an XML file, located in the `META-INF` directory of the EJB JAR file, that specifies the database used with the entity beans and specifies the default behavior of the `EntityManager`. If you set this attribute, the EJB container automatically deploys the referenced persistence unit and sets its JNDI name to its persistence unit name. Similarly, if you do not specify this attribute, but the name of the variable into which you are injecting the persistence context information is the same as the name of a persistence unit in the `persistence.xml` file, then the EJB container again automatically deploys the persistence unit with its JNDI name equal to its unit name. You are required to specify this attribute only if there is more than one persistence unit in the referencing scope.	String	No

javax.persistence.PersistenceUnits

Description

Target: Class

Specifies an array of `@javax.persistence.PersistenceUnit` annotations.

Attributes

Table A-20 Attributes of the javax.pesistence.PersistenceUnits Annotation

Name	Description	Data Type	Required?
value	Specifies the array of `@javax.persistence.PersistenceUnit` annotations.	`PersistenceUnit[]`	Yes

Standard JDK Annotations Used By EJB 3.0

This section provides reference information about the following annotations:

- "javax.annotation.PostConstruct" on page A-26
- "javax.annotation.PreDestroy" on page A-27
- "javax.annotation.Resource" on page A-27
- "javax.annotation.Resources" on page A-29

javax.annotation.PostConstruct

Description

Target: Method

Specifies the lifecycle callback method that the EJB container should execute before the first business method invocation and after dependency injection is done to perform any initialization.

You may specify a `@PostConstruct` method in any bean class that includes dependency injection.

Only one method in the bean class can be annotated with this annotation. If you annotate more than one method with this annotations, the EJB will not deploy.

The method annotated with `@PostConstruct` must follow these requirements:

- The return type of the method must be `void`.
- The method must not throw a checked exception.

- The method may be `public`, `protected`, `package private` or `private`.
- The method must not be `static`.
- The method must not be `final`.

This annotation does not have any attributes.

javax.annotation.PreDestroy

Description

Target: Method

Specifies the lifecycle callback method that signals that the bean class instance is about to be destroyed by the EJB container. You typically apply this annotation to methods that release resources that the bean class has been holding.

Only one method in the bean class can be annotated with this annotation. If you annotate more than one method with this annotations, the EJB will not deploy.

The method annotated with `@PreDestroy` must follow these requirements:

- The return type of the method must be `void`.
- The method must not throw a checked exception.
- The method may be `public`, `protected`, `package private` or `private`.
- The method must not be `static`.
- The method must not be `final`.

This annotation does not have any attributes.

javax.annotation.Resource

Description

Target: Class, Method, Field

Specifies a dependence on an external resource, such as a JDBC data source or a JMS destination or connection factory.

If you specify the annotation on a field or method, the EJB container injects an instance of the requested resource into the bean when the bean is initialized. If you apply the annotation to a class, the annotation declares a resource that the bean will look up at runtime.

Attributes

Table A-21 Attributes of the javax.annotation.Resource Annotation

Name	Description	Data Type	Required?
name	Specifies the name of the resource reference. If you apply the `@Resource` annotation to a field, the default value of the `name` attribute is the field name, qualified by the class name. If you apply it to a method, the default value is the JavaBeans property name corresponding to the method, qualified by the class name. If you apply the annotation to class, there is no default value and thus you are required to specify the attribute.	String	No
type	Specifies the Java data type of the resource. If you apply the `@Resource` annotation to a field, the default value of the `type` attribute is the type of the field. If you apply it to a method, the default is the type of the JavaBeans property. If you apply it to a class, there is no default value and thus you are required to specify this attribute.	Class	No
authenticationType	Specifies the authentication type to use for the resource. You specify this attribute only for resources representing a connection factory of any supported type. Valid values for this attribute are: • `AuthenticationType.CONTAINER` • `AuthenticationType.APPLICATION` Default value is `AuthenticationType.CONTAINER`	AuthenticationType	No

Table A-21 Attributes of the javax.annotation.Resource Annotation

Name	Description	Data Type	Required?
shareable	Indicates whether a resource can be shared between this EJB and other EJBs. You specify this attribute only for resources representing a connection factory of any supported type or ORB object instances. Valid values for this attribute are `true` and `false`. Default value is `true`.	boolean	No.
mappedName	Specifies the global JNDI name of the dependent resource. For example: `mappedName="my.Datasource"` specifies that the JNDI name of the dependent resources is `my.Datasource` and is deployed in the WebLogic Server JNDI tree.	String	No.
description	Specifies a description of the resource.	String	No.

javax.annotation.Resources

Description

Target: Class

Specifies an array of `@Resource` annotations.

Attributes

Table A-22 Attributes of the javax.annotation.Resources Annotation

Name	Description	Data Type	Required?
value	Specifies the array of `@Resource` annotations.	`Resource[]`	Yes.

Standard Security-Related JDK Annotations Used by EJB 3.0

This section provides reference information about the following annotations:

- "javax.annotation.security.DeclareRoles" on page A-30
- "javax.annotation.security.DenyAll" on page A-31
- "javax.annotation.security.PermitAll" on page A-31
- "javax.annotation.security.RolesAllowed" on page A-31
- "javax.annotation.security.RunAs" on page A-32

javax.annotation.security.DeclareRoles

Description

Target: Class

Defines the security roles that will be used in the EJB.

You typically use this annotation to define roles that can be tested from within the methods of the annotated class, such as using the `isUserInRole` method. You can also use the annotation to explicitly declare roles that are implicitly declared if you use the `@RolesAllowed` annotation on the class or a method of the class.

You create security roles in WebLogic Server using the Administration Console. For details, see Manage Security Roles.

Attributes

Table A-23 Attributes of the javax.annotation.security.DeclareRoles Annotation

Name	Description	Data Type	Required?
value	Specifies an array of security roles that will be used in the bean class.	String[]	Yes.

javax.annotation.security.DenyAll

Description

Target: Method

Specifies that no security role is allowed to access the annotated method, or in other words, the method is excluded from execution in the EJB container.

This annotation does not have any attributes.

javax.annotation.security.PermitAll

Description

Target: Method

Specifies that all security roles currently defined for WebLogic Server are allowed to access the annotated method.

This annotation does not have any attributes.

javax.annotation.security.RolesAllowed

Description

Target: Class, Method

Specifies the list of security roles that are allowed to access methods in the EJB.

If you specify it at the class-level, then it applies to all methods in the bean class. If you specify it at the method-level, then it only applies to that method. If you specify the annotation at both the class- and method-level, the method value overrides the class value.

You create security roles in WebLogic Server using the Administration Console. For details, see Manage Security Roles.

Attributes

Table A-24 Attributes of the javax.annotation.security.RolesAllowed Annotation

Name	Description	Data Type	Required?
value	List of security roles that are allowed to access methods of the bean class.	String[]	Yes.

javax.annotation.security.RunAs

Description

Target: Class

Specifies the security role which actually executes the EJB in the EJB container.

The security role must exist in the WebLogic Server security realm and map to a user or group. For details, see Manage Security Roles.

Attributes

Table A-25 Attributes of the javax.annotation.security.RunAs Annotation

Name	Description	Data Type	Required?
value	Specifies the security role which the EJB should run as.	String	Yes.