Oracle® Fusion Middleware

Developing Enterprise JavaBeans for Oracle WebLogic Server

14c (14.1.2.0.0)
F54013-01
December 2024

Oracle Fusion Middleware Developing Enterprise JavaBeans for Oracle WebLogic Server, 14c (14.1.2.0.0)

F54013-01

Primary Author:

Contents

3 Iterative Development of EJBs

4 Programming the Annotated EJB Class

Preface

This document is a resource for software developers who develop applications that include WebLogic Server EJBs using the Jakarta Platform, Enterprise Edition.

Audience

This document is a resource for software developers who develop applications that include WebLogic Server EJBs. It is assumed that the reader is familiar with Jakarta EE and basic EJB programming concepts.

The document mostly discusses the EJB 3.2 programming model, in particular the use of metadata annotations to simplify development. This document does not address EJB topics that are different between versions 2.x and 3.x, such as design considerations, EJB container architecture, entity beans, deployment descriptor use, and so on. This document also does not address production phase administration, monitoring, or performance tuning.

> **Note:**
>
> The EJB 3.2 specification provided in Java EE 8 provides the same functionality as the Jakarta Enterprise Beans 3.2 specification. All references in this document to EJB 3.2 can be interpreted as references to the Jakarta Enterprise Beans 3.2 specification.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documentation

General EJB Design and Architecture

For information about developing and deploying EJBs with WebLogic Server, see:

- Enterprise Java Beans (EJBs) in *Understanding Oracle WebLogic Server*.
- For instructions on how to organize and build WebLogic Server EJBs in a split directory environment, see Creating a Split Development Directory Environment in *Developing Applications for Oracle WebLogic Server*.
- For information on programming and packaging 3.2 and earlier EJBs, see *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.
- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications in development and production environments.

Basic EJB Concepts

For complete information about basic EJB concepts, such as the benefits of enterprise beans, the types of enterprise beans, and their life cycles, visit the following sites:

- Enterprise JavaBeans 3.2 Specification (JSR-345) at `http://jcp.org/en/jsr/summary?id=345`
- The "Enterprise Beans" chapter of the Java EE 8 Tutorial at `https://javaee.github.io/tutorial/partentbeans.html#BNBLR`

Samples and Tutorials

Oracle provides a variety of code examples and tutorials that show WebLogic Server configuration and API use, and provide practical instructions on how to perform key development tasks. For more information, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

In addition, Oracle provides basic EJB examples described in Simple EJB Examples. Oracle recommends that you run these examples before programming your own application that uses EJBs.

- Packaged EJB 3.2 Examples in WebLogic Server
- Packaged EJB 3.1 Examples in WebLogic Server
- EJB 3.0: Example of Invoking an Entity From A Session Bean

New and Changed WebLogic Server Features

For a comprehensive listing of the new features in EJB features introduced in this release of WebLogic Server, see What Is New and Changed in EJB 3.2.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Understanding EJBs

This chapter describes the new features and programming model of EJB 3.2 and also provides a basic overview EJB components, anatomy, and features.

> **Note:**
>
> The EJB 3.2 specification provided in Java EE 8 provides the same functionality as the Jakarta Enterprise Beans 3.2 specification. All references in this document to EJB 3.2 can be interpreted as references to the Jakarta Enterprise Beans 3.2 specification.

This chapter includes the following sections:

New Features and Changes in EJB

These sections summarize the changes in the EJB programming model and requirements between EJB 3.2, 3.1 and 3.0, as well between EJB 2.x and 3.x.

What Is New and Changed in EJB 3.2

With Jakarta EE 8, there is a continuing focus on ease of development by offering a simplified application architecture with a cohesive integrated platform; increased efficiency with reduced boiler-plate code and broader use of annotations.

The following summarizes the main new functionality and simplifications made in EJB 3.2 to the earlier EJB APIs:

- **Enhanced Message-driven Beans** – Enhanced MDB contract with a no-methods message listener interface to expose all public methods as message listener methods. Also, enhanced the list of standard JMS MDB activation properties.
- **EJB Lite** – Extended the EJB Lite Group to include local asynchronous session bean invocations and non-persistent EJB Timer Service. Also defined clear rules for an EJB Lite Container to support other API groups.

- **EJB Timer Enhancements** – Enhanced the `TimerService` API to access all active timers in the EJB module. Also, removed restrictions on `javax.ejb.Timer` and `javax.ejb.TimerHandle` that required references to be used only inside a bean.
- **Stateful Session Bean Enhancements** – Added an option for the lifecycle callback interceptor methods of stateful session beans to be executed in a transaction context determined by the lifecycle callback method's transaction attribute. Also, introduced an option to disable passivation of stateful session beans.
- **Security Enhancements** – Added container provided security role named "`**`" to indicate any authenticated caller independent of the actual role name. Also, simplified requirements for definition of a security role using the ejb deployment descriptor.
- **Java Persistence 2.1 Support** – JPA 2.1 includes new support or enhancements for features including Criteria Bulk Update/Delete, stored procedures, JPQL Generic function, injectable entity listeners, `TREAT`, converters, DDL generation, and entity graphs. For the complete JPA 2.1 specification, see "JSR-000338 Java Persistence 2.1 (Final Release)" at `http://jcp.org/aboutJava/communityprocess/final/jsr338/index.html`.
- **Technology Pruning** – Support for the following features was made optional in this release:
  - EJB 2.1 and earlier Entity Bean Component Contract for Container-Managed Persistence
  - EJB 2.1 and earlier Entity Bean Component Contract for Bean-Managed Persistence
  - Client View of an EJB 2.1 and earlier Entity Bean
  - EJB QL: Query Language for Container-Managed Persistence Query Methods

For a comprehensive listing of the new features and changes for EJBs in Jakarta EE 8, see the Jakarta Enterprise Beans 3.2 Specification at `https://jakarta.ee/specifications/enterprise-beans/3.2/`.

What Was New and Changed in EJB 3.1

The EJB 3.1 specification provides simplified programming and packaging model changes. The mandatory use of Java interfaces from previous versions has been removed, allowing plain old Java objects to be annotated and used as EJB components. The simplification is further enhanced through the ability to place EJB components directly inside of Web applications, removing the need to produce separate archives to store the Web and EJB components and combine them together in an EAR file.

The following summarizes the new functionality and simplifications made in EJB 3.1 to the earlier EJB APIs:

- **Singleton Session Bean** – Singleton session beans provide a formal programming construct that guarantees a session bean will be instantiated once per application in a particular Java Virtual Machine (JVM), and that it will exist for the life cycle of the application. With singletons, you can easily share state between multiple instances of an enterprise bean component or between multiple enterprise bean components in the application.
- **Simplified No Interface Client View** – The No-interface local client view type simplifies EJB development by providing local session bean access without requiring a separate local business interface, allowing components to have EJB bean class instances directly injected.
- **Packaging and Deploying EJBs Directly in a WAR File** – EJB 3.1 provides the ability to place EJB components directly inside of Web application archive (WAR) files, removing the

need to produce archives to store the Web and EJB components and combine them together in an enterprise application archive (EAR) file.

- **Portable Global JNDI Names** – The Portable Global JNDI naming option in EJB 3.1 provides a number of common, well-known namespaces in which EJB components can be registered and looked up from using the pattern `java:global[/<app-name>]/<module-name>/<bean-name>`. This standardizes how and where EJB components are registered in JNDI, and how they can be looked up and used by applications.
- **Asynchronous Session Bean Invocations** – An EJB 3.1 session bean can expose methods with asynchronous client invocation semantics. Using the `@Asynchronous` annotation in an EJB class or specific method will direct the EJB container to return control immediately to the client when the method is invoked. The method may return a future object to allow the client to check on the status of the method invocation, and retrieve result values that are asynchronously produced.
- **EJB Timer Enhancements** – The EJB 3.1 Timer Service supports calendar-based EJB Timer expressions. The scheduling functionality takes the form of CRON-styled schedule definitions that can be placed on EJB methods, in order to have the methods be automatically invoked according to the defined schedule. EJB 3.1 also supports the automatic creation of a timer based on metadata in the bean class or deployment descriptor, which allows the bean developer to schedule a timer without relying on a bean invocation to programmatically invoke one of the Timer Service timer creation methods. Automatically created timers are created by the container as a result of application deployment.
- **Embeddable EJB Container** – EJB 3.1 supports an embeddable API for executing EJB components within a Java SE environment. Unlike traditional Java EE server-based execution, embeddable usage allows client code and its corresponding enterprise beans to run within the same virtual machine and class loader. This provides better support for testing, offline processing (e.g., batch jobs), and the use of the EJB programming model in desktop applications.
- **JPA 2.1 Support** – Oracle EclipseLink is the default JPA 2.1 persistence provider that is shipped with Oracle WebLogic Server. WebLogic Server runs with the JPA 2.1 JAR in the server `classpath`. Although JPA 2.1 is upwardly compatible with JPA 2.0 and 1.0, JPA 2.1 introduced some methods to existing JPA interfaces that conflict with existing signatures in OpenJPA interfaces.

  As a result, applications that use Kodo/JPA as the persistence provider with earlier releases of WebLogic Server must be recompiled. See Configuring the Persistence Provider in Oracle WebLogic Server.

What Was New and Changed in EJB 3.0

The following summarizes the new functionality and simplifications made in EJB 3.0 to the earlier EJB APIs:

- 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.
- 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.

- The bean file can be a plain old 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`.
- Session beans may expose client views via business interfaces. Session beans may either explicitly implement the business interface or they can specify it using the `@javax.ejb.Remote` or `@javax.ejb.Local` annotations.)
- The business interface is a plain old 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`.

> **Note:**
>
> EJBs with a businesses interface that does not implement `java.rmi.Remote` are not accessible from `wlclient.jar`, also known as the IIOP thin client.

- 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.
  - 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.

Understanding EJB Components

Enterprise JavaBeans (EJB) 3.2 technology is the server-side component architecture for Java EE 8. EJB 3.2 technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology.

Session EJBs Implement Business Logic

Session beans implement business logic. There are three types of session beans: stateful, stateless, and singleton. Stateful and stateless session beans serve one client at a time; whereas, singleton session beans can be invoked concurrently.

For detailed information about the types of session beans and when to use them, see "What Is a Session Bean" in the "Enterprise Beans" chapter of the Java EE 8 Tutorial at https://javaee.github.io/tutorial/ejb-intro002.html#GIPJG.

- Stateful Session Beans
- Stateless Session Beans
- Singleton Session Beans

Stateful Session Beans

Stateful session beans maintain state information that reflects the interaction between the bean and a particular client across methods and transactions. A stateful session bean can manage interactions between a client and other enterprise beans, or manage a workflow.

Example: A company Web site that allows employees to view and update personal profile information could use a stateful session bean to call a variety of other beans to provide the services required by a user, after the user clicks **View my Data** on a page:

- Accept the login data from a JSP, and call another EJB whose job it is to validate the login data.
- Send confirmation of authorization to the JSP.
- Call a bean that accesses profile information for the authorized user.

Stateless Session Beans

A stateless session bean does not store session or client state information between invocations—the only state it might contain is not specific to a client, for instance, a cached database connection or a reference to another EJB. At most, a stateless session bean may store state for the duration of a method invocation. When a method completes, state information is not retained.

Any instance of a stateless session bean can serve any client—any instance is equivalent. Stateless session beans can provide better performance than stateful session beans, because each stateless session bean instance can support multiple clients, albeit one at a time. The client of a stateless session bean can be a web service endpoint.

Example: An Internet application that allows visitors to click a **Contact Us** link and send an email could use a stateless session bean to generate the email, based on the *to* and from information gathered *from* the user by a JSP.

Singleton Session Beans

Singleton session beans provide a formal programming construct that guarantees a session bean will be instantiated once per application in a particular Java Virtual Machine (JVM), and that it will exist for the life cycle of the application. With singletons, you can easily share state between multiple instances of an enterprise bean component or between multiple enterprise bean components in the application.

Singleton session beans offer similar functionality to stateless session beans but differ from them in that there is only one singleton session bean per application, as opposed to a pool of stateless session beans, any of which may respond to a client request. Like stateless session beans, singleton session beans can implement Web service endpoints. Singleton session beans maintain their state between client invocations but are not required to maintain their state across server crashes or shutdowns.

Example: The Apache Web site provides a Simple Singleton: ComponentRegistry example that demonstrates how a singleton bean uses Container-Managed Concurrency to utilize the Read `(@Lock(READ))` functionality, to allow multi-threaded access to the bean, and the Write `(@Lock(WRITE))` functionality, to enforce single-threaded access to the bean.

Message-Driven Beans Implement Loosely Coupled Business Logic

A message-driven bean implements loosely coupled or asynchronous business logic in which the response to a request need not be immediate. A message-driven bean receives messages from a JMS Queue or Topic, and performs business logic based on the message contents. It is an asynchronous interface between EJBs and JMS.

Throughout its life cycle, an MDB instance can process messages from multiple clients, although not simultaneously. It does not retain state for a specific client. All instances of a message-driven bean are equivalent—the EJB container can assign a message to any MDB instance. The container can pool these instances to allow streams of messages to be processed concurrently.

The EJB container interacts directly with a message-driven bean—creating bean instances and passing JMS messages to those instances as necessary. The container creates bean instances at deployment time, adding and removing instances during operation based on message traffic.

For detailed information, see *Developing Message-Driven Beans for Oracle WebLogic Server*.

Example: In an on-line shopping application, where the process of taking an order from a customer results in a process that issues a purchase order to a supplier, the supplier ordering process could be implemented by a message-driven bean. While taking the customer order always results in placing a supplier order, the steps are loosely coupled because it is not necessary to generate the supplier order before confirming the customer order. It is acceptable or beneficial for customer orders to "stack up" before the associated supplier orders are issued.

EJB Anatomy and Environment

These sections briefly describe classes required for each bean type, the EJB run-time environment, and the deployment descriptor files that govern a bean's run-time behavior.

EJB Components

Every bean type requires a bean class. Table 1-1 defines the supported client views that make up each type of EJB, and defines any additional required classes.

> **Note:**
>
> The EJB 2.1 and earlier API required that Local and Remote clients access the stateful or stateless session bean by means of the session bean's local or remote home and the local or remote component interfaces. These interfaces remain available for use with EJB 3.x; however, the EJB 2.1 Remote and Local client view is not supported for singleton session beans.
>
> See Create EJB Classes and Interfaces in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Table 1-1 Supported Client Views in EJB 3.2

Client Views	Session Bean Types	Additional Required Classes
Remote Client	Stateful, Stateless, and Singleton session beans	Remote business interface that defines the bean's business and lifecycle methods.
Local Client	Stateful, Stateless, and Singleton session beans	Local business interface that defines the bean's business and lifecycle methods.
Local No- interface	Stateful, Stateless, and Singleton session beans	Only requires the bean class.
Web Service Clients	Stateless and Singleton session beans	A Web service endpoint that is accessed as a JAX-WS service endpoint using the JAX-WS client view APIs.

The EJB Container

An EJB container is a run-time container for beans that are deployed to an application server. The container is automatically created when the application server starts up, and serves as an interface between a bean and run-time services such as:

- Life cycle management
- Code generation
- Security
- Transaction management
- Locking and concurrency control

EJB Metadata Annotations

The WebLogic Server EJB 3.2 programming model uses the Java EE 8 metadata annotations feature in which you create an annotated EJB 3.2 bean file, and then compile the class with standard Java compiler, which can then be packaged into a target module for deployment. At runtime, WebLogic Server parses the annotations and applies the required behavioral aspects to the bean file.

See Programming the Annotated EJB Class.

Optional EJB Deployment Descriptors

As of EJB 3.0, you are no longer required to create the EJB deployment descriptor files (such as `ejb-jar.xml`). However, you can still use XML deployment descriptors if you want. In the case of conflicts, the deployment descriptor value overrides the annotation value.

If you are continuing to use deployment descriptors in your EJB implementation, refer to EJB Deployment Descriptors in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

The WebLogic Server EJB container supports three deployment descriptors:

- `ejb-jar.xml`—The standard Java EE deployment descriptor. The `ejb-jar.xml` may be used to define EJBs and to specify standard configuration settings for the EJBs. An `ejb-jar.xml` can specify multiple beans that will be deployed together.
- `weblogic-ejb-jar.xml`—WebLogic Server-specific deployment descriptor that contains elements related to WebLogic Server features such as clustering, caching, and transactions. This file is required if your beans take advantage of WebLogic Server-specific features. Like `ejb-jar.xml`, `weblogic-ejb-jar.xml` can specify multiple beans that will be deployed together.
- `weblogic-cmp-jar.xml`—WebLogic Server-specific deployment descriptor that contains elements related to container-managed persistence for entity beans. Entity beans that use container-managed persistence must be specified in a `weblogic-cmp-jar.xml` file.

For descriptions of the WebLogic Server EJB deployment descriptors, refer to Deployment Descriptor Schema and Document Type Definitions Reference in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

EJB Clients and Communications

An EJB can be accessed by server-side or client-side objects such as servlets, Java client applications, other EJBs, web services, and non-Java clients. Any client of an EJB, whether in the same or a different application, accesses it in a similar fashion. WebLogic Server automatically creates implementations of an EJB's remote home and remote business interfaces, which can function remotely.

Accessing EJBs

Clients access enterprise beans either through a no-interface view or through a business interface. A no-interface view of an enterprise bean exposes the public methods of the enterprise bean implementation class to clients. Clients using the no-interface view of an enterprise bean may invoke any public methods in the enterprise bean implementation class or any super-classes of the implementation class. A business interface is a standard Java programming language interface that contains the business methods of the enterprise bean.

The client of an enterprise bean obtains a reference to an instance of an enterprise bean through either dependency injection, using Java programming language annotations, or JNDI lookup, using the Java Naming and Directory Interface syntax to find the enterprise bean instance.

Dependency injection is the simplest way of obtaining an enterprise bean reference. Clients that run within a Java EE server-managed environment, JavaServer Faces web applications, JAX-RS web services, other enterprise beans, or Java EE application clients, support dependency injection using the `javax.ejb.EJB` annotation.

Applications that run outside a Java EE server-managed environment, such as Java SE applications, must perform an explicit lookup. JNDI supports a global syntax for identifying Java EE components to simplify this explicit lookup. For more information see, Using the JNDI Portable Syntax.

Because of network overhead, it is more efficient to access beans from a client on the same machine than from a remote client, and even more efficient if the client is in the same application.

For information on programming client access to an EJB, see "Accessing Enterprise Beans" in the "Enterprise Beans" chapter of the Java EE 8 Tutorial at https://javaee.github.io/tutorial/ejb-intro004.html#GIPJF.

EJB Communications

WebLogic Server EJBs use:

- T3—To communicate with remote objects. T3 is a WebLogic-proprietary remote network protocol that implements the Remote Method Invocation (RMI) protocol.
- RMI—To communicate with remote objects. RMI enables an application to obtain a reference to an object located elsewhere in the network, and to invoke methods on that object as though it were co-located with the client on the same JVM locally in the client's virtual machine. An EJB with a remote interface is an RMI object. For more information on WebLogic RMI, see *Developing RMI Applications for Oracle WebLogic Server*.
- HTTP—An EJB can obtain an HTTP connection to a Web server external to the WebLogic Server environment by using the `java.net.URL` resource connection factory. See Configuring EJBs to Send Requests to an URL in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

You can specify the attributes of the network connection an EJB uses by binding the EJB to a WebLogic Server custom network channel. See Configuring Network Resources in *Administering Server Environments for Oracle WebLogic Server*.

Securing EJBs

By default, any user can invoke the public methods of an EJB. Therefore, if you want to restrict access to the EJB, you can use security-related annotations to specify the roles that are allowed to invoke all, or a subset, of the methods, which is explained in Securing Access to the EJB.

For additional information about security and EJBs, see:

- Security Fundamentals in *Understanding Security for Oracle WebLogic Server* has introductory information about authentication, authorization and other security topics.
- Securing Enterprise JavaBeans (EJBs) in *Developing Applications with the WebLogic Security Service* provides instructions for configuring authentication and authorization for EJBs.
-

2

Simple EJB Examples

This chapter describes Java examples of EJBs that use the version 3.*x* programming model. This chapter includes the following sections:

Simple Java Examples of 3.x EJBs

The following sections describe simple Java examples of EJBs that use the new metadata annotation programming model. Some procedural sections in this guide that describe how to program an EJB may reference these examples.

Example of a Simple No-interface Stateless EJB

The EJB 3.1 no-interface local client view type simplifies EJB development by providing local session bean access without requiring a separate local business interface, allowing components to have EJB bean class instances directly injected.

The following code shows a simple no-interface view for the `ServiceBean` stateless session EJB:

```
package examples;
@Stateless
public class ServiceBean {
  public void sayHelloFromServiceBean() {
    System.out.println("Hello From Service Bean!");
  }
}
```

The main points to note about the preceding code are:

- The EJB automatically exposes the no-interface view because no other client views are exposed and its bean class implements clause is empty.
- The `ServiceBean` bean file is a plain Java file; it is not required to implement any EJB-specific interface.
- The class-level `@Stateless` metadata annotation specifies that the EJB is of type stateless session.

Example of a Simple Business Interface 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 javax.ejb.Stateless;
import javax.interceptor.ExcludeDefaultInterceptors;
/**
 * Bean file that implements the Service business interface.
 * Class uses following EJB 3.x annotations:
 *   - @Stateless - specifies that the EJB is of type stateless session
 *   - @ExcludeDefaultInterceptors - specifies any configured default
 *      interceptors should not be invoked for this class
 */
@Stateless
@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 javax.ejb.Stateless;
  import javax.interceptor.ExcludeDefaultInterceptors
  ```

  The annotations that apply only to EJB 3.1 are in the `javax.ejb` package. Annotations that can be used by other Java EE Version 6 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 `@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 javax.ejb.Stateful;
import javax.ejb.Remote;
import javax.ejb.EJB;
import javax.annotation.PreDestroy;
import javax.interceptor.Interceptors;
import javax.interceptor.ExcludeClassInterceptors;
/**
 * Bean file that implements the Account business interface.
 * Uses the following EJB annotations:
 *    -  @Stateful: specifies that this is a stateful session EJB
 *    -  @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
@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 javax.ejb.Stateful;
  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 3.1 are in the `javax.ejb` package. Annotations that can be used by other Java EE 6 components are in more generic packages, such `javax.interceptor` or `javax.annotation`.
- 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 `@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 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 a 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.

Packaged EJB 3.2 Examples in WebLogic Server

The following sections describe the packaged Java EE 7 examples included with Oracle WebLogic Server, which demonstrate new features in EJB 3.2.

- EJB 3.2: Example of Using the Session Bean Lifecyle
- EJB 3.2: Example of a Message-Driven Bean with No-Methods Listener

EJB 3.2: Example of Using the Session Bean Lifecyle

This example shows the new session bean lifecycle callback interceptor methods API, including `@AroundConstruct` and `@AroundInvoke`.

After you have installed WebLogic Server, the example is in the following directory:

```
EXAMPLES_HOME/examples/src/examples/javaee7/ejb/lifecylce
```

`EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

EJB 3.2: Example of a Message-Driven Bean with No-Methods Listener

This example shows how a message-driven bean to implement a listener interface with no methods. A bean that implements a no-methods interface exposes all non-static public methods of the bean class and of any superclasses, except `java.lang.Object`, as message listener methods.

After you have installed WebLogic Server, the example is in the following directory:

```
EXAMPLES_HOME/examples/src/examples/javaee7/ejb/no-method-listener
```

`EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

Packaged EJB 3.1 Examples in WebLogic Server

The following sections describe the packaged Java EE 6 examples included with Oracle WebLogic Server, which demonstrate new features in EJB 3.1.

- EJB 3.1: Example of a Singleton Session Bean
- EJB 3.1: Example of an Asynchronous Method EJB
- EJB 3.1: Example of a Calendar-based Timer EJB
- EJB 3.1: Example of Simplified No-interface Programming and Packaging in a WAR File
- EJB 3.1: Example of Using a Portable Global JNDI Name in an EJB
- EJB 3.1: Example of Using the Embeddable EJB Container in Java SE

- EJB 3.0: Example of Invoking an Entity From A Session Bean

EJB 3.1: Example of a Singleton Session Bean

This example demonstrates the use of the EJB 3.1 singleton session bean, which provides application developers with a formal programming construct that guarantees a session bean will be instantiated once for an application in a particular Java Virtual Machine (JVM). In this example, a `@Singleton` session bean provides a central counter service. The Counter EJB is called from a Java client to demonstrate it is being used, with the count being consistently incremented by "1" as the client is invoked multiple times.

After you have installed WebLogic Server, the example is in the following directory:

```
EXAMPLES_HOME/examples/src/examples/javaee6/ejb/singletonBean
```

`EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

EJB 3.1: Example of an Asynchronous Method EJB

This example demonstrates the use of the EJB 3.1 asynchronous method invocation. Adding the `@Asynchronous` annotation to an EJB class or specific method will direct the EJB container to return control immediately to the client when the method is invoked. The method may return a Future object to allow the client to check on the status of the method invocation, and then retrieve result values that are asynchronously produced.

In this example, an `@Stateless` bean is annotated at the class level, with `@Asynchronous` indicating its methods are all asynchronous, with each of the methods simulating a long-running calculation. A servlet is used to call the various asynchronous methods, keeping track of the invocation and completion times to demonstrate the asynchronous nature of the method calls.

After you have installed WebLogic Server, the example is in the following directory:

```
EXAMPLES_HOME/examples/src/examples/javaee6/ejb/asyncMethodOfEJB
```

`EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server.*

EJB 3.1: Example of a Calendar-based Timer EJB

This example demonstrates the enhanced scheduling capabilities of EJB 3.1. This scheduling functionality takes the form of CRON-styled schedule definitions that can be placed on EJB methods, in order for the methods to be automatically invoked according to the defined schedule. This example shows the use of the `@Schedule` annotation defined for a method of a `@Singleton` session bean, which generates and stores the timestamp of when the method was called. A corresponding servlet is provided, into which the `TimerBean` is injected, which retrieves the list of timestamps to display in a browser.

After you have installed WebLogic Server, the example is in the following directory:

```
EXAMPLES_HOME/examples/src/examples/javaee6/ejb/calendarStyledTimer
```

`EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

EJB 3.1: Example of Simplified No-interface Programming and Packaging in a WAR File

This example demonstrates the simplified programming and packaging model changes provided in EJB 3.1. Since the mandatory use of Java interfaces from previous versions has been removed in EJB 3.1, plain-old Java objects can be annotated and used as EJB components. The simplification is further enhanced by the ability to place EJB components directly inside of Web applications, thereby removing the need to produce archives to store the Web and EJB components and combine them together in an enterprise archive (EAR) file.

In this example, a `@Stateless` annotation is provided on a plain-old Java class that exposes it as an EJB session bean. This is then injected into a `@WebServlet` class using an `@EJB` annotation to demonstrate that it is being used as an EJB module. The EJB session bean and servlet classes are then packaged and deployed together in a single WAR file, which demonstrates the simplified packaging and deployment changes available in Java EE 6.

After you have installed WebLogic Server, the example is in the following directory:

`EXAMPLES_HOME/examples/src/examples/javaee6/ejb/noInterfaceViewInWAR`

`EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

EJB 3.1: Example of Using a Portable Global JNDI Name in an EJB

This example demonstrates the use of the Portable Global JNDI naming option that is available in EJB 3.1. Portable Global JNDI provides a number of common, well-known namespaces in which EJB components can be registered and looked up from using the pattern `java:global[/<app-name>]/<module-name>/<bean-name>`. This standardizes how and where EJB components are registered in JNDI and how they can be looked up and used by applications. In this example, a servlet is used to look up an EJB session bean using its portable JNDI name `java:module/HelloBean`.

After you have installed WebLogic Server, the example is in the following directory:

`EXAMPLES_HOME/examples/src/examples/javaee6/ejb/portableGlobalJNDIName`

`EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

EJB 3.1: Example of Using the Embeddable EJB Container in Java SE

This example demonstrates using the embeddable EJB container available in EJB 3.1, which allows client code and its corresponding enterprise beans to run in a Java SE environment without having to deploy them to a Java EE server.

The example uses the embeddable WebLogic EJB container to view all the user objects being invoked from a Java SE environment. All the user objects are predefined during eager initialization of a singleton component `InitBean` when the application is started, using the annotations `@Startup` and `@PostConstruct`. An instance of `EJBContainer` is created in the

Java client `UserClient` to look up the session bean reference `UserBean` and call its business method `viewUsers` in the application.

After running the example, the client class is executed automatically and prints run-time messages in the command shell in which the example was built. All the existing user objects are retrieved from the samples database and are displayed in detail in the command shell.

After you have installed WebLogic Server, the example is in the following directory:

```
EXAMPLES_HOME/examples/src/examples/javaee6/ejb/embeddableContainer
```

`EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

EJB 3.0: Example of Invoking an 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:

```
EXAMPLES_HOME/examples/src/examples/ejb/ejb30
```

`EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

3

Iterative Development of EJBs

This chapter describes the general EJB 3.2 implementation process, and provides guidance for how to get an EJB up and running in WebLogic Server.
This chapter includes the following sections:

Overview of the EJB Development Process

This section is a brief overview of the EJB 3.2 development process. It describes the key implementation tasks and associated results.

The following section mostly discusses the EJB 3.2 programming model and points out the differences between the EJB 3.*x* and EJB 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 New Features and Changes in EJB.

Table 3-1 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 Annotated EJB Class	Create the Java file that implements the interface and includes the EJB 3.2 metadata annotations that describe how your EJB behaves.	`.java` file.
3	Program the EJB Interface	Create no-interface client views or business interfaces that describe your EJB.	`.java` file for each interface.

Table 3-1 (Cont.) EJB Development Tasks and Results

#	Step	Description	Result
4	Optionally Program Interceptors	Optionally, create the interceptor classes that describe the interceptors that intercept a business method invocation or a life cycle callback event.	`.java` file for each interceptor class.
5	Optionally Program the EJB Timer Service	Optionally, create timers that schedule callbacks to occur when a timer object expires for timed event.	Either metadata (for automatic timers) and/or bean class changes (for programmatic timers).
6	Programming Access to EJB Clients	Obtain a reference to an EJB through either dependency injection or JNDI lookup.	Metadata (annotations and/or deployment descriptor settings) and/or code changes to the client.
7	Programming and Configuring Transactions	Program container-managed or bean-managed transactions.	Metadata and possibly logic to handle exceptions (retry logic or calls to `setRollbackOnly`).
8	Compile Java Source	Compile source code.	`.class` file for each class and interface.
9	Optionally Create and Edit Deployment Descriptors	Optionally create the EJB-specific deployment descriptors, although this step is no longer required when using the EJB 3.2 programming model.	• `ejb-jar.xml`, • `weblogic-ejb-jar.xml`, which contains elements that control WebLogic Server-specific features.
10	Packaging EJBs	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.
11	Deploying EJBs	Target the archive or application directory to desired Managed Server, or a WebLogic Server cluster, in accordance with selected staging mode.	Deployed EJBs are ready to service invocations.

Create a Source Directory

Create a source directory where you will assemble the EJB 3.2 module.

Oracle 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 3.2 as an enterprise application archive (EAR), see Overview of the Split Development Directory Environment in *Developing Applications for Oracle WebLogic Server*.

- Directory Structure for Packaging a JAR
- Directory Structure for Packaging a WAR

Directory Structure for Packaging a JAR

If you prefer to package and deploy your EJBs in a JAR file, create a directory for your class files. If you are also using the EJB deployment descriptor (which is optional but supported in the EJB 3.2 programming model), you can package it as `META-INF/ejb-jar.xml`.

For more information see, Packaging EJBs in a JAR.

Example 3-1 Directory Structure for Packaging a JAR

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

Directory Structure for Packaging a WAR

EJBs can also be packaged directly in a web application module (WAR) by putting the EJB classes in a subdirectory named `WEB-INF/classes` or in a JAR file within `WEB-INF/lib` directory. Optionally, if you are also using the EJB deployment descriptor, you can package it as `WEB-INF/ejb-jar.xml`.

> **Note:**
>
> EJB 2.1 Entity Beans and EJB 1.1 Entity Beans are not supported within WAR files. These component types must only be packaged in a stand-alone `ejb-jar` file or an `ejb-jar` file packaged within an EAR file.

For more information see, Packaging an EJB In a WAR.

Example 3-2 Directory Structure for Packaging a WAR

```
myEJBwar/
     WEB-INF/
          ejb-jar.xml
          weblogic.xml
          weblogic-ejb-jar.xml
          /classes
               foo.class
               fooServlet.class
               fooBean.class
```

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 3.2 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 Programming the Annotated EJB Class.

Program the EJB Interface

Clients access enterprise beans either through a no-interface view or through a business interface.

- Accessing EJBs Using the No-Interface Client View
- Accessing EJBs Using the Business Interface

Accessing EJBs Using the No-Interface Client View

The EJB 3.2 No-interface local client view type simplifies EJB development by providing local session bean access without requiring a separate local business interface, allowing components to have EJB bean class instances directly injected.

The no-interface view has the same behavior as the EJB 3.0 local view. For example, it supports features such as pass-by-reference calling semantics and transaction, and security propagation. However, a no-interface view does not require a separate interface. That is, all public methods of the bean class are automatically exposed to the caller. By default, any session bean that has an empty `implements` clause and does not define any other local or remote client views, exposes a no-interface client view.

You can follow these links to explore code examples of a no-interface client view:

- Example of a Simple No-interface Stateless EJB
- EJB 3.1: Example of Simplified No-interface Programming and Packaging in a WAR File .
- EJB 3.2: Example of a Message-Driven Bean with No-Methods Listener.

For more detailed information about the implementing the no-interface client view, see "Accessing Local Enterprise Beans Using the No-Interface View" in the "Enterprise Beans" chapter of the Java EE 8 Tutorial at https://javaee.github.io/tutorial/ejb-intro004.html#GIPSC.

Accessing EJBs Using the Business Interface

The EJB 3.2 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 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.

> **Note:**
>
> The only requirement for an EJB 3.2 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 Business Interface Stateless EJB and Example of a Simple Stateful EJB for examples of business interfaces implemented by stateless and stateful session beans.

For additional details and examples of specifying the business interface, see Specifying the Business and Other Interfaces.

- Business Interface Application Exceptions
- Using Generics in EJBs
- Serializing and Deserializing Business Objects

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 *Developing JAX-WS Web Services for Oracle WebLogic Server*.

Using Generics in EJBs

The EJB 3.2 programming model supports the use of generics in the business interface at the class level.

Oracle 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 `RootI<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) {
  }
}
```

If you define the type variables on the business interface or class, they will be erased. In this case, the EJB application can be deployed successfully only when the bean class parameterizes the business interface with upper bounds of the type parameter and no other generic information. For example, in the following example, the upper bound is `Object`:

```
public class StatelessSample implements StatelessI<Object> {
   public Object getObject() {
      return null;
   }
   public void updateObject(Object object) {
   }
}
```

Serializing and Deserializing Business Objects

Business object serialization and deserialization are supported by the following interfaces, which are implemented by all business objects:

- `weblogic.ejb.spi.BusinessObject`
- `weblogic.ejb.spi.BusinessHandle`

Use the `BusinessObject._WL_getBusinessObjectHandle()` method to get the business handle object and serialize the business handle object.

To deserialize a business object, deserialize the business handle object and use the `BusinessHandle.getBusinessObject()` method to get the business object.

Optionally Program Interceptors

An interceptor is a method that intercepts the invocation of a business method or a life cycle 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 Life Cycle Callback Events for information on programming the bean class to use interceptors.

Optionally Program the EJB Timer Service

WebLogic Server supports the EJB timer service defined in the EJB 3.2 Specification. As per the EJB 3.2 "specification, the EJB Timer Service is a "container-managed service that allows callbacks to be scheduled for time-based events. The container provides a reliable and transactional notification service for timed events. Timer notifications may be scheduled to occur according to a calendar-based schedule, at a specific time, after a specific elapsed duration, or at specific recurring intervals."

The Timer Service is implemented by the EJB container. An enterprise bean accesses this service by means of dependency injection, through the `EJBContext` interface, or through lookup in the JNDI namespace.

The Timer Service is intended to be used as a coarse-grained timer service. Rather than having a large number of timer objects performing the same task on a unique set of data, Oracle recommends using a small number of timers that perform bulk tasks on the data. For example, assume you have an EJB that represents an employee's expense report. Each expense report must be approved by a manager before it can be processed. You could use one EJB timer to periodically inspect all pending expense reports and send an email to the corresponding manager to remind them to either approve or reject the reports that are waiting for their approval.

- Overview of the Timer Service
- Calendar-based EJB Timers
- Automatically-created EJB Timers
- Non-persistent Timers
- Clustered Versus Local EJB Timer Services
- Configuring Clustered EJB Timers
- Using Java Programming Interfaces to Program Timer Objects

Overview of the Timer Service

The timer service provides methods for the programmatic creation and cancellation of timers, as well as for locating the timers that are associated with a bean. Timers can also be created automatically by the container at deployment time based on metadata in the bean class or in the deployment descriptor. Timer objects can be created for stateless session beans, singleton session beans, message-driven beans, and 2.1 entity beans. Timers cannot be created for stateful session beans.

> **Note:**
>
> The calendar-based timer, automatically-created timers, and non-persistent timer functionality is not supported for 2.1 Entity beans.

A timer is created to schedule timed callbacks. The bean class of an enterprise bean that uses the timer service must provide one or more timeout callback methods, as follows:

- **Programmatic Timers** – For programmatically-created timers, this method may be a method that is annotated with the `Timeout` annotation, or the bean may implement the `javax.ejb.TimedObject` interface. The `javax.ejb.TimedObject` interface has a single method, the timer callback method `ejbTimeout`.
- **Automatic Timers** – For automatically-created timers, the timeout method may be a method that is annotated with the `Schedule` annotation.
- **2.1 Entity Bean Timers** – A timer that is created for a 2.1 entity bean is associated with the entity bean's identity. The timeout callback method invocation for a timer that is created for a stateless session bean or a message-driven bean may be called on any bean instance in the pooled state.

Calendar-based EJB Timers

The EJB 31. Timer Service supports calendar-based EJB Timer expressions. The scheduling functionality takes the form of CRON-styled schedule definitions that can be placed on EJB methods, in order to have the methods be automatically invoked according to the defined schedule.

> **Note:**
>
> Calendar-based timers are not supported for EJB 2.x entity beans.

Calendar-based timers can be created programmatically using the two methods in the `javax.ejb.TimerService` that accept a `javax.ejb.ScheduleExpression` as an argument. The `ScheduleExpression` is constructed and populated prior to creating the Timer. For creating automatic calendar-base timers, the `javax.ejb.Schedule` annotation (and its corresponding `ejb-jar.xml` element) contains a number of attributes that allow for calendar-based timer expressions to be configured.

For detailed information about the seven attributes in a calendar-based time expression, see "Section 18.2.1, Calendar Based Timer Expressions" in the Enterprise JavaBeans 3.2 Specification (JSR-345) at `http://jcp.org/en/jsr/summary?id=345`.

Automatically-created EJB Timers

The EJB 3.2 Timer Service supports the automatic creation of a timer based on metadata in the bean class or deployment descriptor. This allows the bean developer to schedule a timer without relying on a bean invocation to programmatically invoke one of the Timer Service timer creation methods. Automatically created timers are created by the container as a result of application deployment.

The `javax.ejb.Schedule` annotation can be used to automatically create a timer with a particular timeout schedule. This annotation is applied to a method of a bean class (or super-class) that should receive the timer callbacks associated with that schedule. Multiple automatic timers can be applied to a single timeout callback method using the `javax.ejb.Schedules` annotation.

When the clustered EJB Timer implementation is used, each `Schedule` annotation corresponds to a single persistent timer, regardless of the number of servers across which the EJB is deployed.

Non-persistent Timers

By default, EJB timers are persistent. A non-persistent timer is a timer whose lifetime is tied to the JVM in which it is created. A non-persistent timer is considered canceled in the event of application shutdown, container crash, or a failure/shutdown of the JVM on which the timer was started.

> **Note:**
>
> Non-persistent timers are not supported for EJB 2.x Entity Beans.

Non-persistent timers can be created programmatically or automatically (using `@Schedule` or the deployment descriptor).

Automatic non-persistent timers can be specified by setting the persistent attribute of the `@Schedule` annotation to false. For automatic non-persistent timers, the container creates a new non-persistent timer during application initialization for each JVM across which the container is distributed.

Clustered Versus Local EJB Timer Services

You can configure two types of EJB timer services: clustered or local.

- Clustered EJB Timer Services
- Local EJB Timer Services

Clustered EJB Timer Services

Clustered EJB timer services provide the following advantages:

- Better visibility.

  Timers are accessible from any node in a cluster. For example, the `javax.ejb.TimerService.getTimers()` method returns a complete list of all stateless session or message-driven bean timers in a cluster that were created for the EJB. If you pass the primary key of the entity bean to the `getTimers()` method, a list of timers for that entity bean are returned.

- Automatic load balancing and failover.

  Clustered EJB timer services take advantage of the load balancing and failover capabilities of the Job Scheduler.

For information about the configuring a clustered EJB timer service, see Configuring Clustered EJB Timers.

Local EJB Timer Services

Local EJB timer services execute only on the server on which they are created and are visible only to the beans on that server. With a local EJB timer service, you do not have to configure a cluster, database, JDBC data source, or leasing service, as you do for clustered EJB timer services.

You cannot migrate a local EJB timer object from one server to another; timer objects can only be migrated as part of an entire server. If a server that contains EJB timers goes down for any reason, you must restart the server or migrate the entire server in order for the timers to execute.

> **Caution:**
>
> In a clustered environment, the local timer implementation has severe limitations; therefore Oracle recommends not using local timers in a clustered environment. Instead, use the clustered timer implementation in a clustered environment. A deployment-time warning will be thrown when a local timer implementation is configured to be used in a clustered environment.

Configuring Clustered EJB Timers

> **Note:**
>
> To review the advantages of using clustered EJB timers, see Clustered Versus Local EJB Timer Services.

To configure the clustering of EJB timers, perform the following steps:

1. Ensure that you have configured the following:
   - A clustered domain. See Setting up WebLogic Clusters in *Administering Clusters for Oracle WebLogic Server*.
   - Features of the Job Scheduler, including:
     - HA database, such as Oracle, DB2, Informix, MySQL, Sybase, or MSSQL.
     - JDBC data source that is mapped to the HA database using the `<data-source-for-job-scheduler>` element in the `config.xml` file.
     - Leasing service. By default, database leasing will be used and the JDBC data source defined by the `<data-source-for-job-scheduler>` element in the `config.xml` file will be used.

   For more information about configuring the Job Scheduler, see *The Timer and Work Manager API* in *Developing CommonJ Applications for Oracle WebLogic Server*.
2. To enable the clustered EJB timer service, set the `timer-implementation` element in the `weblogic-ejb-jar.xml` deployment descriptor to `Clustered`:

   ```
   <timer-implementation>Clustered</timer-implementation>
   ```

For more information, see the `timer-implementation` element description in the *weblogic-ejb-jar.xml Deployment Descriptor Reference* in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Please note the following changes in the behavior of the clustered EJB timer service:

- The `weblogic.ejb.WLTimer*` interfaces are not supported with clustered EJB timer services.
- When creating a new clustered EJB timer using the `createTimer()` method, you may notice a delay in timeout execution during the initial setup of the timer.
- The Job Scheduler provides an "at least once" execution guarantee. When a clustered EJB timer expires, the database is not updated until the timer listener callback method completes. If the server were to crash before the database is updated, the timer expiration would be executed twice.
- Timer configuration options related to the actions to take in the event of failure are not valid for the clustered EJB timer service. These configuration options include: retry delay, maximum number of retry attempts, maximum number of time-outs, and time-out failure actions.
- The Job Scheduler queries the database every 30 seconds to identify timers that are due to expire. Execution may be delayed for timers with an interval duration less than 30 seconds.
- Only transactional timers will be retried in the event of failure.
- Fixed rate scheduling of timer execution is not supported.

Using Java Programming Interfaces to Program Timer Objects

This section summarizes the Java programming interfaces defined in the EJB 3.2 Specification that you can use to program timers. For detailed information on these interfaces, refer to the EJB 3.2 Specification. This section also provides details about the WebLogic Server-specific timer-related interfaces.

- EJB 3.2 Timer-related Programming Interfaces
- WebLogic Server-specific Timer-related Programming Interfaces

EJB 3.2 Timer-related Programming Interfaces

EJB 3.2 interfaces you can use to program timers are described in the following table.

Table 3-2 EJB 3.2 Timer-related Programming Interfaces

Programming Interface	Description
`javax.ejb.ScheduleExpression`	Create calendar-based EJB Timer expressions.
`javax.ejb.Schedule`	Automatically create a timer with a particular timeout schedule. Multiple automatic timers can be applied to a single timeout callback method using the `javax.ejb.Schedules` annotation.
`javax.ejb.TimedObject`	Implement for the enterprise bean class of a bean that will be registered with the timer service for timer callbacks. This interface has a single method, `ejbTimeout`.
`EJBContext`	Access the timer service using the `getTimerService` method.

Table 3-2 (Cont.) EJB 3.2 Timer-related Programming Interfaces

Programming Interface	Description
`javax.ejb.TimerService`	Create new EJB timers or access existing EJB timers for the EJB.
`javax.ejb.Timer`	Access information about a particular EJB timer.
`javax.ejb.TimerHandle`	Define a serializable timer handle that can be persisted. Since timers are local objects, a `TimerHandle` must not be passed through a bean's remote interface or Web service interface.

For more information on EJB 2.1 timer-related programming interfaces, see the EJB 2.1 Specification.

WebLogic Server-specific Timer-related Programming Interfaces

WebLogic Server-specific interfaces you can use to program timers include:

- `weblogic.management.runtime.EJBTimerRuntimeMBean`—provides runtime information and administrative functionality for timers from a particular `EJBHome`. The `weblogic.management.runtime.EJBTimerRuntimeMBean` interface is shown in Example 3-3.

Example 3-3 weblogic.management.runtime.EJBTimerRuntimeMBean Interface

```
public interface weblogic.management.runtime.EJBTimerRuntimeMBean {
  public int getTimeoutCount(); // get the number of successful timeout notifications
that have been made
  public int getActiveTimerCount(); // get the number of active timers for this EJBHome
  public int getCancelledTimerCount(); // get the number of timers that have been
cancelled for this EJBHome
  public int getDisabledTimerCount(); // get the number of timers temporarily disabled
for this EJBHome
  public void activateDisabledTimers(); // activate any temporarily disabled timers
}
```

- `weblogic.ejb.WLTimerService` interface—extends the `javax.ejb.TimerService` interface to allow users to specify WebLogic Server-specific configuration information for a timer. The `weblogic.ejb.WLTimerService` interface is shown in Example 3-4; for information on the `javax.ejb.TimerService`, see the EJB 2.1 Specification.

> **Note:**
>
> The `weblogic.ejb.WLTimerService` interface is not supported by the clustered EJB timer service, as described in Configuring Clustered EJB Timers.

Example 3-4 weblogic.ejb.WLTimerService Interface

```
public interface WLTimerService extends TimerService {
  public Timer createTimer(Date initial, long duration, Serializable info,
    WLTimerInfo wlTimerInfo)
    throws IllegalArgumentException, IllegalStateException, EJBException;
  public Timer createTimer(Date expiration, Serializable info,
    WLTimerInfo wlTimerInfo)
    throws IllegalArgumentException, IllegalStateException, EJBException;
  public Timer createTimer(long initial, long duration, Serializable info
```

```
    WLTimerInfo wlTimerInfo)
    throws IllegalArgumentException, IllegalStateException, EJBException;
  public Timer createTimer(long duration, Serializable info,
    WLTimerInfo wlTimerInfo)
    throws IllegalArgumentException, IllegalStateException, EJBException;
}
```

- `weblogic.ejb.WLTimerInfo` interface—used in the `weblogic.ejb.WLTimerService` interface to pass WebLogic Server-specific configuration information for a timer. The `weblogic.ejb.WLTimerInfo` method is shown in Example 3-5.

> **Note:**
>
> The `weblogic.ejb.WLTimerService` interface is not supported by the clustered EJB timer service, as described in Configuring Clustered EJB Timers.

Example 3-5 weblogic.ejb.WLTimerInfo Interface

```
public final interface WLTimerInfo {
  public static int REMOVE_TIMER_ACTION  = 1;
  public static int DISABLE_TIMER_ACTION = 2;
  public static int SKIP_TIMEOUT_ACTION  = 3;
  /**
   * Sets the maximum number of retry attempts that will be
   * performed for this timer.  If all retry attempts
   * are unsuccesful, the timeout failure action will
   * be executed.
   */
  public void setMaxRetryAttempts(int retries);
  public int getMaxRetryAttempts();
  /**
   * Sets the number of milliseconds that should elapse
   * before any retry attempts are made.
   */
  public void setRetryDelay(long millis);
  public long getRetryDelay();
  /**
   * Sets the maximum number of timeouts that can occur
   * for this timer.  After the specified number of
   * timeouts have occurred successfully, the timer
   * will be removed.
   */
  public void setMaxTimeouts(int max);
  public int getMaxTimeouts();
/**
   * Sets the action the container will take when ejbTimeout
   * and all retry attempts fail.  The REMOVE_TIMER_ACTION,
   * DISABLE_TIMER_ACTION, and SKIP_TIMEOUT_ACTION fields
   * of this interface define the possible values.
   */
  public void setTimeoutFailureAction(int action);
  public int getTimeoutFailureAction();
}
```

- `weblogic.ejb.WLTimer` interface—extends the `javax.ejb.Timer` interface to provide additional information about the current state of the timer. The `weblogic.ejb.WLTimer` interface is shown in Example 3-6.

> **Note:**
>
> The `weblogic.ejb.WLTimerService` interface is not supported by the clustered EJB timer service, as described in Configuring Clustered EJB Timers.

Example 3-6 weblogic.ejb.WLTimer Interface

```
public interface WLTimer extends Timer {
  public int getRetryAttemptCount();
  public int getMaximumRetryAttempts();
  public int getCompletedTimeoutCount();
}
```

Programming Access to EJB Clients

This section provides some guidelines in determining the client view to provide for accessing an enterprise bean.

- Remote Clients
- Local Clients
- Looking Up EJBs From Clients
- Configuring EJBs to Send Requests to a URL
- Specifying an HTTP Resource by URL
- Specifying an HTTP Resource by Its JNDI Name
- Accessing HTTP Resources from Bean Code
- Configuring Network Communications for an EJB

Remote Clients

As stated in the EJB 3.2 specification, a remote client accesses a session bean through the bean's remote business interface. For a session bean client and component written to the EJB 2.1 and earlier APIs, the remote client accesses the session bean through the session bean's remote home and remote component interfaces.

> **Note:**
>
> The EJB 2.1 and earlier API required that a remote client access the stateful or stateless session bean by means of the session bean's remote home and remote component interfaces. These interfaces remain available for use with EJB 3.x, and are described in *Create EJB Classes and Interfaces* in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

The remote client view of an enterprise bean is location independent. A client running in the same JVM as a bean instance uses the same API to access the bean as a client running in a different JVM on the same or different machine.

Local Clients

As stated in the EJB 3.2 specification, a local client accesses a session bean through the bean's local business interface or through a no-interface client view representing all the public methods of the bean class. For a session bean or entity bean client and component written to the EJB 2.1 and earlier APIs, the local client accesses the enterprise bean through the bean's local home and local component interfaces. The container object that implements a local business interface or the no-interface local view is a local Java object.

A local client is a client that is collocated in the same application with the session bean that provides the local client view and which may be tightly coupled to the bean. A local client of a session bean may be another enterprise bean or a Web component. Access to an enterprise bean through the local client view requires the collocation in the same application of both the local client and the enterprise bean that provides the local client view. The local client view therefore does not provide the location transparency provided by the remote client view.

Looking Up EJBs From Clients

The client of an enterprise bean obtains a reference to an instance of an enterprise bean through either dependency injection, using Java programming language annotations, or JNDI lookup, using the Java Naming and Directory Interface syntax to find the enterprise bean instance.

> **Note:**
>
> For instructions on how clients can look up 2.x or earlier enterprise beans using EJB Links, see *Using EJB Links* in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

- Using Dependency Injection
- Using the JNDI Portable Syntax
- Customizing JNDI Names

Using Dependency Injection

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.

For detailed information, see Injecting Resource Dependency into a Variable or Setter Method.

Using the JNDI Portable Syntax

The Portable Global JNDI naming option in EJB 3.2 provides a number of common, well-known namespaces in which EJB components can be registered and looked up from using the patterns listed in this section. This standardizes how and where EJB components are registered in JNDI, and how they can be looked up and used by applications.

Three JNDI namespaces are used for portable JNDI lookups: `java:global`, `java:module`, and `java:app`.

- The `java:global` JNDI namespace is the portable way of finding remote enterprise beans using JNDI lookups. JNDI addresses are of the following form:

  `java:global`[*/application name]/module name/enterprise bean name[/interface name*]

  Application name and module name default to the name of the application and module minus the file extension. Application names are required only if the application is packaged within an EAR. The interface name is required only if the enterprise bean implements more than one business interface.

- The `java:module` namespace is used to look up local enterprise beans within the same module. JNDI addresses using the `java:module` namespace are of the following form:

  `java:module`*/enterprise bean name/[interface name*]

  The interface name is required only if the enterprise bean implements more than one business interface.

- The `java:app` namespace is used to look up local enterprise beans packaged within the same application. That is, the enterprise bean is packaged within an EAR file containing multiple Java EE modules. JNDI addresses using the `java:app` namespace are of the following form:

  `java:app`[*/module name]/enterprise bean name[/interface name*]

  The module name is optional. The interface name is required only if the enterprise bean implements more than one business interface.

For example, if an enterprise bean, `MyBean`, is packaged within the Web application archive `myApp.war`, the default module name is `myApp`. (In this example, the module name could be explicitly configured in the `web.xml` file.) The portable JNDI name is `java:module/MyBean`. An equivalent JNDI name using the `java:global` namespace is `java:global/myApp/MyBean`.

Customizing JNDI Names

Though global JNDI bindings are registered by default, you can also customize the JNDI names of your EJB client view bindings by using the `weblogic.javaee.JNDIName` and `weblogic.javaee.JNDINames` annotations. For more information, see weblogic.javaee.JNDIName and weblogic.javaee.JNDINames.

For EJBs using deployment descriptors, you can specify the custom JNDI bindings in the `weblogic-ejb-jar.xml` deployment descriptor by using the `jndi-binding` element. See EJB Deployment Descriptors in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Configuring EJBs to Send Requests to a URL

To enable an EJB to open an `HttpURLConnection` to an external HTTP server using the `java.net.URL` resource manager connection factory type, specify the URL, or specify an object bound in the JNDI tree that maps to a URL, using either the `@Resource` annotation in the bean class, or if using deployment descriptors, by using the `resource-ref` element in `ejb-jar.xml` and the `res-ref-name` element in `weblogic-ejb-jar.xml`.

Specifying an HTTP Resource by URL

When using annotations to specify the URL to which an EJB sends requests:

1. Annotate a URL field in your bean class with `@Resource`.
2. Specify the URL value using the look-up element of `@Resource`.

When using deployment descriptors to specify the URL to which an EJB sends requests:

1. In `ejb-jar.xml`, specify the URL in the `<jndi-name>` element of the `resource-ref` element.
2. In `weblogic-ejb-jar.xml`, specify the URL in the `<jndi-name>` element of the `resource-description` element:

```
<resource-description>
    <res-ref-name>url/MyURL</res-ref-name>
    <jndi-name>http://www.rediff.com/</jndi-name>
</resource-description>
```

   WebLogic Server creates a URL object with the `jndi-name` provided and binds the object to the `java:comp/env`.

Specifying an HTTP Resource by Its JNDI Name

When using annotations to specify an object that is bound in JNDI and maps to a URL, instead of specifying a URL:

1. Annotate a URL field in your bean class with `@Resource`.
2. Specify the name by which the URL is bound in JNDI using the look-up element of `@Resource`.

When using deployment descriptors to specify an object that is bound in JNDI and maps to a URL, instead of specifying a URL:

1. In `ejb-jar.xml`, specify the name by which the URL is bound in JNDI in the `<jndi-name>` element of the `resource-ref` element.
2. In `weblogic-ejb-jar.xml`, specify the name by which the URL is bound in JNDI in the `<jndi-name>` element of the `resource-description` element:

```
<resource-description>
    <res-ref-name>url/MyURL1</res-ref-name>
    <jndi-name>firstName</jndi-name>
</resource-description>
```

   where `firstName` is the object bound to the JNDI tree that maps to the URL. This binding could be done in a startup class. When `jndi-name` is not a valid URL, WebLogic Server

treats it as an object that maps to a URL and is already bound in the JNDI tree, and binds a `LinkRef` with that `jndi-name`.

Accessing HTTP Resources from Bean Code

Regardless of how you specified an HTTP resource—by its URL or a JNDI name that maps to the URL—you can access it from EJB code in this way:

```
URL url = (URL) context.lookup("java:comp/env/url/MyURL");
connection = (HttpURLConnection)url.openConnection();
```

Configuring Network Communications for an EJB

You can control the attributes of the network connection an EJB uses for communications by configuring a custom network channel and assigning it to the EJB. For information about WebLogic Server network channels and associated configuration instructions see Configure Network Resources in *Administering Server Environments for Oracle WebLogic Server*. After you configure a custom channel, assign it to an EJB using the `network-access-point` element in `weblogic-ejb-jar.xml`.

Programming and Configuring Transactions

The following sections contain guidelines for programming transactions.

- Programming Container-Managed Transactions
- Configuring Automatic Retry of Container-Managed Transactions
- Programming Bean-Managed Transactions
- Programming Transactions That Are Distributed Across EJBs

Programming Container-Managed Transactions

Container-managed transactions are simpler to program than bean-managed transactions, because they leave the job of demarcation—starting and stopping the transaction—to the EJB container. You configure the desired transaction behaviors using EJB annotations `javax.ejb.TransactionAttribute` or by using EJB deployment descriptors `ejb-jar.xml` and `weblogic-ejb-jar.xml`.

- For more information about using EJB annotations to specify container-managed transactions in a bean file, see Specifying Transaction Management and Attributes.
- For more information about using EJB deployment descriptors to specify container-managed transactions, see Container-Managed Transaction Elements in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Key programming guidelines for container-managed transactions include:

- Preserve transaction boundaries—Do not invoke methods that interfere with the transaction boundaries set by the container. Do not use:
  - The `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection`
  - The `getUserTransaction` method of `javax.ejb.EJBContext`
  - Any method of `javax.transaction.UserTransaction`

- Roll back transactions explicitly—To cause the container to roll back a container-managed transaction explicitly, invoke the `setRollbackOnly` method of the `EJBContext` interface. (If the bean throws a non-application exception, typically an `EJBException`, the rollback is automatic.)
- Avoid serialization problems—Many data stores provide limited support for detecting serialization problems, even for a single user connection. In such cases, even with `transaction-isolation` in `weblogic-ejb-jar.xml` set to `TransactionSerializable`, exceptions or rollbacks in the EJB client might occur if contention occurs between clients for the same rows. To avoid such exceptions, you can:
  - Include code in your client application to catch SQL exceptions, and resolve them appropriately; for example, by restarting the transaction.
  - For Oracle databases, use the transaction isolation settings described in `isolation-level` in the weblogic-ejb-jar.xml Deployment Descriptor Reference appendix in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Configuring Automatic Retry of Container-Managed Transactions

In Oracle WebLogic Server, you can specify that, if a business method that has started a transaction fails because of a transaction rollback that is not related to a system exception, the EJB container will start a new transaction and retry the failed method up to a specified number of times. If the method fails for the specified number of retry attempts, the EJB container throws an exception.

> **Note:**
>
> The EJB container does not retry any transactions that fail because of system exception-based errors.

To configure automatic retry of container-managed transactions:

1. Make sure your bean is a container-managed session or entity bean.

   You can configure automatic retry of container-managed transactions for container-managed session and entity beans only. You cannot configure automatic retry of container-managed transactions for message-driven beans because MDBs do not acknowledge receipt of a message they are processing when the transaction that brackets the receipt of the message is rolled back; messages are automatically retried until they are acknowledged. You also cannot configure automatic retry of container-managed transactions for timer beans because, when a timer bean's ejbTimeout method starts and is rolled back, the timeout is always retried.
2. Make sure the business methods for which you want to configure automatic retry of transactions are defined in the bean's remote or local interface or as home methods (local home business logic that is not specific to a particular bean instance) in the home interface; the methods must have one of the following container-managed transaction attributes:
   - `RequiresNew`. If a method's transaction attribute (`trans-attribute` element in `ejb-jar.xml`) is `RequiresNew`, a new transaction is always started prior to the invocation of the method and, if configured, automatic retry of transactions occurs if the transaction fails.

- `Required`. If a method's transaction attribute (`trans-attribute` element in `ejb-jar.xml`) is `Required`, the method is retried with a new transaction only if the failed transaction was begun on behalf of the method.

For more information on:

- Programming interfaces, see Programming Access to EJB Clients.
- The `trans-attribute` element in `ejb-jar.xml`, see Container-Managed Transaction Elements in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*, which provides detailed information about creating and editing EJB deployment descriptors.

3. Make sure the methods for which you want to enable automatic retry of transactions are safe to be re-invoked. A retry of a failed method must yield results that are identical to the results the previous attempt, had it been successful, would have yielded. In particular:
   - If invoking a method initiates a call chain, it must be safe to re-invoke the entire call chain when the method is retried.
   - All of the method's parameters must be safe for reuse; when a method is retried, it is retried with the same parameters that were used to invoke the failed attempt. In general, parameters that are primitives, immutable objects, or are references to read-only objects are safe for reuse. If a parameter is a reference to an object that is to be modified by the method, re-invoking the method must not negatively affect the result of the method call.
   - If the bean that contains the method that is being retried is a stateful session bean, the bean's conversational state must be safe to re-invoke. Since a stateful session bean's state is not transactional and is not restored during a transaction rollback, in order to use the automatic retry of transactions feature, you must first be sure the bean's state is still valid after a rollback.
4. Specify the methods for which you want the EJB container to automatically retry transactions and the number of retry attempts you want the EJB container to make in the `retry-methods-on-rollback` element in `weblogic-ejb-jar.xml`.

Programming Bean-Managed Transactions

This section contains programming considerations for bean-managed transactions.

- Demarcate transaction boundaries—To define transaction boundaries in EJB or client code, you must obtain a `UserTransaction` object and begin a transaction before you obtain a Java Transaction Service (JTS) or JDBC database connection. To obtain the `UserTransaction` object, use this command:

  ```
  ctx.lookup("javax.transaction.UserTransaction");
  ```

  After obtaining the `UserTransaction` object, specify transaction boundaries with `tx.begin()`, `tx.commit()`, `tx.rollback()`.

  If you start a transaction after obtaining a database connection, the connection has no relationship to the new transaction, and there are no semantics to "enlist" the connection in a subsequent transaction context. If a JTS connection is not associated with a transaction context, it operates similarly to a standard JDBC connection that has `autocommit` equal to `true`, and updates are automatically committed to the data store.

  Once you create a database connection within a transaction context, that connection is reserved until the transaction commits or rolls back. To optimize performance and

throughput, ensure that transactions complete quickly, so that the database connection can be released and made available to other client requests.

> **Note:**
>
> You can associate only a single database connection with an active transaction context.

- Setting transaction isolation level—For bean-managed transactions, you define isolation level in the bean code. Allowable isolation levels are defined on the `java.sql.Connection` interface. For information on isolation level behaviors, see `isolation-level` in the weblogic-ejb-jar.xml Deployment Descriptor Reference appendix in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

  See Example 3-7 for a code sample.

Example 3-7 Setting Transaction Isolation Level in BMT

```
import javax.transaction.Transaction;
import java.sql.Connection
import weblogic.transaction.TxHelper:
import weblogic.transaction.Transaction;
import weblogic.transaction.TxConstants;
User Transaction tx = (UserTransaction)
ctx.lookup("javax.transaction.UserTransaction");
//Begin user transaction
    tx.begin();
//Set transaction isolation level to TransactionReadCommitted
Transaction tx = TxHelper.getTransaction();
    tx.setProperty (TxConstants.ISOLATION_LEVEL, new Integer
    (Connection.TransactionReadCommitted));
//perform transaction work
    tx.commit();
```

- Avoid restricted methods—Do not invoke the `getRollbackOnly` and `setRollbackOnly` methods of the `EJBContext` interface in bean-managed transactions. These methods should be used only in container-managed transactions. For bean-managed transactions, invoke the `getStatus` and rollback methods of the `UserTransaction` interface.
- Use one connection per active transaction context—You can associate only a single database connection with an active transaction context.

Programming Transactions That Are Distributed Across EJBs

This section describes two approaches for distributing a transaction across multiple beans, which may reside on multiple server instances.

- Calling multiple EJBs from a client's transaction context
- Using an EJB "Wrapper" to Encapsulate a Cross-EJB Transaction

Calling multiple EJBs from a client's transaction context

The code fragment below is from a client application that obtains a `UserTransaction` object and uses it to begin and commit a transaction. The client invokes two EJBs within the context of the transaction.

```
import javax.transaction.*;
...
u = (UserTransaction) jndiContext.lookup("javax.transaction.UserTransaction");
u.begin();
account1.withdraw(100);
account2.deposit(100);
u.commit();
...
```

The updates performed by the `account1` and `account2` beans occur within the context of a single `UserTransaction`. The EJBs commit or roll back together, as a logical unit, whether the beans reside on the same server instance, different server instances, or a WebLogic Server cluster.

All EJBs called from a single transaction context must support the client transaction—each beans' `trans-attribute` element in `ejb-jar.xml` must be set to `Required`, `Supports`, or `Mandatory`.

Using an EJB "Wrapper" to Encapsulate a Cross-EJB Transaction

You can use a *wrapper* EJB that encapsulates a transaction. The client calls the wrapper EJB to perform an action such as a bank transfer, and the wrapper starts a new transaction and invokes one or more EJBs to do the work of the transaction.

The wrapper EJB can explicitly obtain a transaction context before invoking other EJBs, or WebLogic Server can automatically create a new transaction context, if the wrapper's `trans-attribute` element in `ejb-jar.xml` is set to `Required` or `RequiresNew`.

All EJBs invoked by the wrapper EJB must support the wrapper EJB's transaction context—their `trans-attribute` elements must be set to `Required`, `Supports`, or `Mandatory`.

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, typically using the standard Java compiler. The resulting class files can then be packaged into a target module for deployment. Typical tools to compile include:

- `javac` —The `javac` compiler provided with the Java SE SDK provides Java compilation capabilities. See `http://www.oracle.com/technetwork/java/javase/documentation/index.html`.
- `weblogic.appc`—To reduce deployment time, use the `weblogic.appc` Java class (or its equivalent Ant task `wlappc`) to pre-compile a deployable archive file, (WAR, JAR, or EAR). Precompiling with `weblogic.appc` generates certain helper classes and performs validation checks to ensure your application is compliant with the current Java EE specifications. See Building Modules and Applications Using wlappc in *Developing Applications for Oracle WebLogic Server*.
- `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 in *Developing Applications for Oracle WebLogic Server*.

Optionally Create and Edit Deployment Descriptors

An important aspect of the EJB 3.x programming model was 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 3.2 fully supports the use of deployment descriptors, even though the standard Java EE 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 3.2 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 3.2 because currently they do not have any annotation equivalent.

The 2.x version of *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server* 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)
- Edit Deployment Descriptors
- Deployment Descriptor Schema and Document Type Definitions Reference
- weblogic-ejb-jar.xml Deployment Descriptor Reference
- weblogic-cmp-jar.xml Deployment Descriptor Reference

Packaging EJBs

Oracle recommends that you package EJBs as part of an enterprise application. See Deploying and Packaging from a Split Development Directory in *Developing Applications for Oracle WebLogic Server*.

However, EJB 3.2 simplifies packaging by providing the ability to place EJB components directly inside of Web application archive (WAR) files, removing the need to produce separate archives to store the Web and EJB components and combine them together in an enterprise application archive (EAR) file.

- Packaging EJBs in a JAR

Packaging EJBs in a JAR

WebLogic Server supports the use of `ejb-client.jar` files for packaging the EJB classes that a programmatic client in a different application requires to access the EJB.

Specify the name of the client JAR in the `ejb-client-jar` element of the bean's `ejb-jar.xml` file. When you run the `appc` compiler, a JAR file with the classes required to access the EJB is generated.

Make the client JAR available to the remote client. For Web applications, put the `ejb-client.jar` in the `/lib` directory. For non-Web clients, include `ejb-client.jar` in the client's classpath.

> **Note:**
>
> WebLogic Server classloading behavior varies, depending on whether the client is stand-alone. Stand-alone clients with access to the `ejb-client.jar` can load the necessary classes over the network. However, for security reasons, programmatic clients running in a server instance cannot load classes over the network.

Packaging an EJB In a WAR

EJB 3.2 has removed the restriction that enterprise bean classes must be packaged in an `ejb-jar` file. Therefore, EJB classes can be packaged directly inside a Web application archive (WAR) using the same packaging guidelines that apply to Web application classes. Simply put your EJB classes in the `WEB-INF/classes` directory or in a JAR file within `WEB-INF/lib` directory. Optionally, if you are also using the EJB deployment descriptor, you can package it as `WEB-INF/ejb-jar.xml`. When you run the `appc` compiler, a WAR file with the classes required to access the EJB components is generated.

In a WAR file there is a single component naming environment shared between all the components (web, enterprise bean, etc.) defined by the module. Each enterprise bean defined by the WAR file shares this single component environment namespace with all other enterprise beans defined by the WAR file and with all other web components defined by the WAR file.

Enterprise beans (and any related classes) packaged in a WAR file have the same class loading requirements as other non-enterprise bean classes packaged in a WAR file. This means, for example, that a servlet packaged within a WAR file is guaranteed to have visibility to an enterprise bean component packaged within the same WAR file, and vice versa.

> **Caution:**
>
> EJB 2.1 Entity Beans and EJB 1.1 Entity Beans are not supported within WAR files. These component types must only be packaged in a stand-alone `ejb-jar` file or an `ejb-jar` file packaged within an EAR file. Applications that violate this restriction will fail to deploy.

There is an example of using the simplified WAR packaging method bundled in the WebLogic Server distribution kit. See EJB 3.1: Example of Simplified No-interface Programming and Packaging in a WAR File .

Deploying EJBs

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.

For general instructions on deploying WebLogic Server applications and modules, including EJBs, see *Deploying Applications to Oracle WebLogic Server*. For EJB-specific deployment issues and procedures, see Deploying Standalone EJBs as Part of an Enterprise Application. and Deploying EJBs as Part of an Web Application.

For more information about deploying an EJB created with the 2.x programming model, see Deployment Guidelines For Enterprise JavaBeans in the *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server* guide, which concentrates on the 2.x programming model.

4

Programming the Annotated EJB Class

This chapter describes how to program the annotated EJB 3.2 class file.
This chapter includes the following sections:

Overview of Metadata Annotations and EJB Bean Files

The WebLogic Server EJB 3.2 programming model uses the Jakarta EE 8 metadata annotations feature in which you create an annotated EJB 3.2 bean file, compile the class with the standard Java compiler, and the resulting class can then be packaged into a target module for deployment. At runtime, WebLogic Server parses the annotations and applies the required behavioral aspects to the bean file.

> **Tip:**
>
> To reduce deployment time, you can also use the WebLogic compile tool `weblogic.appc` (or its Ant equivalent `wlappc`) to pre-compile a deployable archive file, (WAR, JAR, or EAR). Precompiling with `weblogic.appc` generates certain helper classes and performs validation checks to ensure your application is compliant.

The annotated 3.2 bean file is the core of your EJB. It contains the Java code that determines how your EJB behaves. The 3.2 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 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 for a breakdown of the annotations you can specify in a bean file, by function. These annotations include those described by the following specifications:

- Jakarta Enterprise Beans 3.2 Specification (JSR-345) at `https://jakarta.ee/specifications/enterprise-beans/3.2/`
- Jakarta Annotations at `https://jakarta.ee/specifications/annotations/1.3/`

See EJB 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 3.2, described in Iterative Development of EJBs.

For more information on general EJB design and architecture, see the following documents:

- Enterprise JavaBeans web site at http://www.oracle.com/technetwork/java/ejb-141389.html
- Introducing the Java EE 6 Platform: Part 3 (EJB Technology, Even Easier to Use) at http://www.oracle.com/technetwork/articles/javaee/javaee6overview-part3-139660.html#ejbeasy
- The Java EE 8 Tutorial at https://javaee.github.io/tutorial/toc.html

Programming the Bean File: Requirements and Changes From EJB 2.x

The requirements for programming the EJB 3.2 bean class file are essentially the same as the EJB 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 EJB 2.*x* and EJB 3.2.

See *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server* for detailed information about the mandatory and optional requirements for programming the bean class.

Bean Class Requirements and Changes From EJB 2.*x*

The following bullets list the EJB 3.2 requirements for programming a bean class, as well as the EJB 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.Singleton`
  - `@javax.ejb.MessageDriven`

  > **Note:**
  >
  > Oracle Kodo JPA/JDO is not supported in this release of WebLogic Server. However, if you still using Oracle Kodo, programming a 3.0 entity bean (`@javax.ejb.Entity`) is discussed in a separate document.
  >
  > Customers are encouraged to use Oracle TopLink, which supports JPA 2.1. Kodo supports only JPA 1.0. For more information, see Configuring the Persistence Provider in Oracle WebLogic Server.

- If the bean is a session bean, the bean class can implement either:
  - The no-interface local client view type, which simplifies EJB development by providing local session bean access without requiring a separate local business interface. (As of EJB 3.2, MDBs can also use the no-interface local client view.)
  - The bean's business interface(s) or the methods of the bean's business interface(s), if any.

- Session beans no longer need 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 longer 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

The sections that follow provide the recommended steps when programming the annotated EJB 3.2 class file.

- Programming Application Exceptions
- Securing Access to the EJB
- Specifying Transaction Management and Attributes

Typical Steps When Programming the Bean File

The following procedure describes the typical basic steps when programming the 3.2 bean file for a EJB. The steps you follow depends, of course, on what your EJB does.

Refer to Simple EJB Examples, for code examples of the topics discussed in the remaining sections.

1. Import the EJB 3.2 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 3.2 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 interface that your EJB is going to implement, either a business interface or a no-interface view, 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.

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

   See Specifying the Bean Type (Stateless, Singleton, Stateful, or Message-Driven).

4. Optionally, use dependency injection to use external resources, such as another EJB or other Java EE 8 object.

   See Injecting Resource Dependency into a Variable or Setter Method.

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

   See Invoking a 3.0 Entity.

6. Optionally, program and configure business method or life cycle 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 Life Cycle Callback Events.

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 for EJB 3.2 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.

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

   See Specifying Transaction Management and Attributes.

Specifying the Business and Other Interfaces

The EJB 3.x local or remote client of a session bean written to the EJB 3.x API accesses a session bean through its business interface. A local client may also access a session bean through a no-interface view that exposes all public methods of the bean class.

- Specifying the Business Interface
- Specifying the No-interface View

Specifying the 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 Business Interface Stateless EJB.

Specifying the No-interface View

Client access to an enterprise bean that exposes a local, no-interface view is accomplished through either dependency injection or JNDI lookup. As of EJB 3.2, MDBs can also use the no-interface local client view.

- To obtain a reference to the no-interface view of an enterprise bean through dependency injection, use the `javax.ejb.EJB` annotation and specify the enterprise bean's implementation class:

```
@EJB
ExampleBean exampleBean;
```

- To obtain a reference to the no-interface view of an enterprise bean through JNDI lookup, use the javax.naming.InitialContext interface's lookup method:

```
ExampleBean exampleBean = (ExampleBean)
InitialContext.lookup("java:module/ExampleBean");
```

Clients do not use the new operator to obtain a new instance of an enterprise bean that uses a no-interface view.

There are code examples of using the No-interface client view bundled in the WebLogic Server distribution kit. See EJB 3.1: Example of Simplified No-interface Programming and Packaging in a WAR File and EJB 3.2: Example of a Message-Driven Bean with No-Methods Listener.

For more detailed information about the implementing the no-interface client view, see "Accessing Local Enterprise Beans Using the No-Interface View" in the "Enterprise Beans" chapter of the Java EE 8 Tutorial at `https://javaee.github.io/tutorial/ejb-intro004.html#GIPSC`.

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

There is only one required metadata annotation in a 3.2 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.Singleton`—Specifies that you are programming a singleton 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.

> **Note:**
>
> Oracle Kodo JPA/JDO is not supported in this release of WebLogic Server. However, if you still using Oracle Kodo, programming a 3.0 entity bean (`@javax.ejb.Entity`) is discussed in a separate document.
>
> Customers are encouraged to use Oracle TopLink, which supports JPA 2.1. Kodo supports only JPA 1.0. For more information, see Configuring the Persistence Provider in Oracle WebLogic Server.

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 Business Interface Stateless EJB.

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 and javax.annotation.Resource 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:**
>
> Oracle TopLink, a JPA 2.1 persistence provider, is the default JPA provider, replacing Kodo, which was the default provider in previous releases. Any application that does not specify a JPA provider in `persistence.xml` will now use TopLink by default. For more information, see Configuring the Persistence Provider in Oracle WebLogic Server.

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
- Finding an Entity Using the EntityManager API
- Creating and Updating an Entity Using EntityManager

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 life cycle 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 life cycle 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 life cycle of an entity; see Configuring the Persistence Provider in Oracle WebLogic Server 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 `Person`s 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 Configuring the Persistence Provider in Oracle WebLogic Server.

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 Life Cycle Callback Events

An interceptor is a method that intercepts a business method invocation or a life cycle callback event. There are two types of interceptors: those that intercept business methods and those that intercept life cycle 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 life cycle 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 Life Cycle Interceptors: Typical Steps
- Programming the Interceptor Class
- Programming Business Method Interceptor Methods
- Programming Asynchronous Business Methods
- Programming Life Cycle Callback Interceptor Methods
- Specifying Default Interceptor Methods
- Saving State Across Interceptors With the InvocationContext API

Specifying Business or Life Cycle 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 for an example of specifying interceptors and Example of an Interceptor Class 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.
   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 life cycle callback annotations become the life cycle 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.
4. In the bean class or interceptor class (wherever you are programming the interceptor methods), program life cycle callback interceptor methods, as described in Programming Business Method Interceptor Methods.
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.

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 life cycle 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.

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 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.

Programming Asynchronous Business Methods

Session beans can implement asynchronous methods, business methods where control is returned to the client by the enterprise bean container before the method is invoked on the session bean instance. Clients may then use the Java SE concurrency API to retrieve the result, cancel the invocation, and check for exceptions. Asynchronous methods are typically used for long-running operations, for processor-intensive tasks, for background tasks, to increase application throughput, or to improve application response time if the method invocation result isn't required immediately.

When a session bean client invokes a typical non-asynchronous business method, control is not returned to the client until the method has completed. Clients calling asynchronous methods, however, immediately have control returned to them by the enterprise bean container. This allows the client to perform other tasks while the method invocation completes. If the method returns a result, the result is an implementation of the `java.util.concurrent.Future<V>` interface, where “V" is the result value type. The `Future<V>` interface defines methods the client may use to check if the computation is completed, wait for the invocation to complete, retrieve the final result, and cancel the invocation.

Asynchronous method invocation semantics only apply to the no-interface, local business, and remote business client views. Methods exposed through the EJB 2.x local, EJB 2.x remote, and Web service client views must not be designated as asynchronous.

For detailed instructions on creating an asynchronous business method, see "Asynchronous Method Invocation" in the "Enterprise Beans" chapter of the Java EE 8 Tutorial at https://javaee.github.io/tutorial/ejb-async001.html#GKKQG.

Programming Life Cycle Callback Interceptor Methods

You specify a method to be a life cycle 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 life cycle 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 life cycle interceptor annotations to specify that a method is a life cycle 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 life cycle 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 life cycle callback event, you must associate multiple interceptor classes with the bean file, in addition to optionally specifying the life cycle 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:

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

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

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

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

See Saving State Across Interceptors With the InvocationContext API for additional information about the `InvocationContext` API.

See javax.ejb.PostActivate, javax.ejb.PrePassivate, javax.annotation.PostConstruct, and javax.annotation.PreDestroy for additional requirements when programming the life cycle interceptor class.

For an example, see Example of an Interceptor Class.

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 life cycle). 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="3.2"
     xmlns="xmlns.jcp.org/xml/ns/javaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
           http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd">

...

   <assembly-descriptor>

...

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

   </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 life cycle 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 life cycle 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 life cycle 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.

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 3.2.

You create security roles and map users to roles using the WebLogic Remote Console to update your security realm. See Security Policies and Roles in the *Oracle WebLogic Remote Console Online Help*.

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.

Complete List of Metadata Annotations By Function

The tables in the sections that follow group the annotations based on what task they perform. EJB Metadata Annotations Reference, provides full reference information about the EJB 3.2 metadata annotations in alphabetical order.

Annotations to Specify the Bean Type

The following summarize the annotations used to specify the bean type.

Table 4-1 Annotations to Specify the Bean Type

Annotation	Description
`@javax.ejb.Stateless`	Specifies that the bean class is a stateless session bean. For more information, see javax.ejb.Stateless.
`@javax.ejb.Singleton`	Specifies that the bean class is a singleton session bean. For more information, see javax.ejb.Singleton.

Table 4-1 (Cont.) Annotations to Specify the Bean Type

Annotation	Description
`@javax.ejb.Stateful`	Specifies that the bean class is a stateful session bean. For more information, see javax.ejb.Startup.
`@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. For more information, see javax.ejb.Init.
`@javax.ejb.Remove`	Specifies a remove method of a stateful session bean. For more information, see javax.ejb.Remove.
`@javax.ejb.MessageDriven`	Specifies that the bean class is a message-driven bean. For more information, see javax.ejb.MessageDriven.
`@javax.ejb.ActivationConfigProperty`	Specifies properties used to configure a message-driven bean in its operational environment. For more information, see javax.ejb.ActivationConfigProperty.

Annotations to Specify the Local or Remote Interfaces

The following summarize the annotations used to specify the local or remote interfaces.

Table 4-2 Annotations to Specify the Local or Remote Interfaces

Annotation	Description
`@javax.ejb.Local`	Specifies a local interface of the bean. For more information, see javax.ejb.Local.
`@javax.ejb.Remote`	Specifies a remote interface of the bean. For more information, see javax.ejb.Remote.

Annotations to Support EJB 2.x Client View

The following summarize the annotations used to support EJB 2.x client view.

Table 4-3 Annotations to Support EJB 2.x Client View

Annotation	Description
`@javax.ejb.LocalHome`	Specifies a local home interface of the bean. For more information, see javax.ejb.LocalHome.
`@javax.ejb.RemoteHome`	Specifies a remote home interface of the bean. For more information, see javax.ejb.RemoteHome.

Annotations to Invoke a 3.0 Entity Bean

The following summarize the annotations used to invoke a 3.0 entity bean.

Table 4-4 Annotations to Invoke a 3.0 Entity Bean

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

Transaction-Related Annotations

The following summarize the annotations used for transactions.

Table 4-5 Transaction-Related Annotations

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

Annotations to Specify Interceptors

The following summarize the annotations used to specify interceptors.

Table 4-6 Annotations to Specify Interceptors

Annotation	Description
`@javax.interceptor.Interceptors`	Specifies the list of interceptor classes associated with a bean class or method. For more information, see javax.interceptor.Interceptors.
`@javax.interceptor.AroundInvoke`	Specifies an interceptor method. For more information, see javax.interceptor.AroundInvoke.
`@javax.interceptor.ExcludeClassInterceptors`	Specifies that, when the annotated method is invoked, the class-level interceptors should *not* invoke. For more information, see javax.interceptor.ExcludeClassInterceptors.
`@javax.interceptor.ExcludeDefaultInterceptors`	Specifies that, when the annotated method is invoked, the default interceptors should *not* invoke. For more information, see javax.interceptor.ExcludeDefaultInterceptors.

Annotations to Specify Life Cycle Callbacks

The following summarize the annotations used to specify life cycle callbacks.

Table 4-7 Annotations to Specify Life Cycle Callbacks

Annotation	Description
`@javax.ejb.PostActivate`	Designates a method to receive a callback after a stateful session bean has been activated. For more information, see javax.ejb.PostActivate.
`@javax.ejb.PrePassivate`	Designates a method to receive a callback before a stateful session bean is passivated. For more information, see javax.ejb.PrePassivate.
`@javax.annotation.PostConstruct`	Specifies the method that needs to be executed after dependency injection is done to perform any initialization. For more information, see javax.annotation.PostConstruct.
`@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. For more information, see javax.annotation.PreDestroy.

Security-Related Annotations

The following metadata annotations are not specific to EJB 3.2, but rather, are general security-related annotations in the `javax.annotation.security` package.

Table 4-8 Security-Related Annotations

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

Context Dependency Annotations

The following summarize the annotations used for context dependency.

Table 4-9 Context Dependency Annotations

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

Timeout and Exceptions Annotations

The following summarize the annotations used for timeout and exceptions.

Table 4-10 Timeout and Exception Annotations

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

Timer and Scheduling Annotations

The following summarize the annotations used for timers scheduling-specific annotations.

Table 4-11 Timer and Scheduling Annotations

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

5

Deployment Guidelines for EJBs

This chapter provides EJB-specific deployment guidelines. For deployment topics that are common to all deployable application units, this chapter gives cross-references to topics in *Deploying Applications to Oracle WebLogic Server*, a comprehensive guide to deploying WebLogic Server applications and modules.
This chapter includes the following sections:

Before You Deploy an EJB

Before starting the deployment process you should have:

- Functional, tested bean code, in an exploded directory format or packaged in an archive file—a JAR for a stand-alone EJB, an EAR if the EJB is part of an enterprise application, or a WAR if the EJB is part of a Web application—along with the deployment descriptors. For production environments, Oracle recommends that you package your application as an EAR.

  > **Note:**
  >
  > EJB 3.1 (and later) removed the restriction that enterprise bean classes must be packaged in an `ejb-jar` file. Therefore, EJB classes can be packaged directly inside a Web application archive (WAR) using the same packaging guidelines that apply to Web application classes. See Deploying EJBs as Part of an Web Application.

  For an overview of the steps required to create and package an EJB, see Overview of the EJB Development Process.

- Program the required annotated EJB class to specify the type of EJB—either: `@javax.ejb.Stateful`, `@javax.ejb.Stateless`, `@javax.ejb.Singleton`, or `@javax.ejb.MessageDriven`.

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

- Configured the optional, but supported, deployment descriptors—`ejb-jar.xml` and `weblogic-ejb-jar.xml`, and, for entity EJBs that use container-managed persistence, `weblogic-cmp-jar.xml`.

  To create EJB deployment descriptors, see *Generate Deployment Descriptors* in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Understanding and Performing Deployment Tasks

Table 5-1 is a guide to WebLogic Server documentation topics that help you make decisions about deployment strategies and provide instructions for performing deployment tasks. For EJB-specific deployment topics, see Deployment Guidelines for EJBs.

Table 5-1 Deployment Tasks and Topics

If You Want To....	See This Topic
Deploy in a development environment	Deploying and Packaging from a Split Development Directory in *Developing Applications for Oracle WebLogic Server*.
Select a deployment tool	Deployment Tools in *Deploying Applications to Oracle WebLogic Server*
Determine appropriate packaging for a deployment	Preparing Applications and Modules for Deployment in *Deploying Applications to Oracle WebLogic Server*.
Organizing EJB components in a split directory structure.	EJBs in *Developing Applications for Oracle WebLogic Server*.
Select staging mode	Controlling Deployment File Copying with Staging Modes in *Deploying Applications to Oracle WebLogic Server*.
Perform specific deployment tasks	Overview of the Deployment Process in *Deploying Applications to Oracle WebLogic Server*.

Deployment Guidelines for EJBs

The following sections provide guidelines for deploying EJBs.

- Deploying Standalone EJBs as Part of an Enterprise Application
- Deploying EJBs as Part of an Web Application
- Deploying EJBs That Call Each Other in the Same Application
- Deploying EJBs That Use Dependency Injection
- Deploying Homogeneously to a Cluster
- Deploying EJBs to a Cluster
- Redeploying an EJB
- Using FastSwap Deployment to Minimize Deployment
- Understanding Warning Messages
- Disabling EJB Deployment Warning Messages

Deploying Standalone EJBs as Part of an Enterprise Application

Oracle recommends that you package and deploy your stand-alone EJB applications as part of an Enterprise application. An Enterprise application is a Jakarta EE 8 deployment unit that bundles together Web applications, EJBs, and Resource Adapters into a single deployable unit.

This is a Oracle best practice, which allows for easier application migration, additions, and changes. Also, packaging your applications as part of an Enterprise application allows you to

take advantage of the split development directory structure, which provides a number of benefits over the traditional single directory structure.

See Overview of the Split Development Directory Environment in *Developing Applications for Oracle WebLogic Server*.

Deploying EJBs as Part of an Web Application

Enterprise beans can also be packaged within a web application module (WAR).

EJB 3.1 (and later) removed the restriction that enterprise bean classes must be packaged in an `ejb-jar` file. Therefore, EJB classes can be packaged directly inside a Web application archive (WAR) using the same packaging guidelines that apply to Web application classes. Simply put your EJB classes in the `WEB-INF/classes` directory or in a JAR file within `WEB-INF/lib` directory. Optionally, if you are also using the EJB deployment descriptor, you can package it as `WEB-INF/ejb-jar.xml`. When you run the `appc` compiler, a WAR file with the classes required to access the EJB components is generated.

See Packaging an EJB In a WAR.

Deploying EJBs That Call Each Other in the Same Application

When an EJB in one application calls an EJB in another application, WebLogic Server passes method arguments by value, due to classloading requirements. When EJBs are in the same application, WebLogic Server can pass method arguments by reference; this improves the performance of method invocation because parameters are not copied.

For best performance, package components that call each other in the same application, and set `enable-call-by-reference` in `weblogic-ejb-jar.xml` to `True`. (By default, `enable-call-by-reference` is `False`.)

- Switching Protocol Limitation

Switching Protocol Limitation

If an application client request has multiple hops, and QOS is configured differently between servers, then you must switch the protocol.

For example, when a client sends an SSL request to a JMS front-end cluster, the JMS front-end cluster then forwards the request to the JMS back-end cluster using clear text. In this case, you may need to switch from the *t3s* protocol to the *t3* protocol.

> **Note:**
>
> You can switch the protocol only in a default channel. Custom channels do not support protocol switching.

Deploying EJBs That Use Dependency Injection

When an EJB uses dependency injection, the resource name defined in the class and the superclass must be unique. For example:

```
public class ClientServlet extends HttpServlet {
    @EJB(name = 'DateServiceBean', beanInterface = DateService.class)
```

```
    private DateService bean; .... }
public class DerivedClientServlet extends ClientServlet {
    @EJB(name = MyDateServiceBean', beanInterface = DateService.class)
    private DateService bean; .... }
```

For more information about dependency injection, see Using Jakarta Annotations and Dependency Injection in *Developing Applications for Oracle WebLogic Server*.

Deploying Homogeneously to a Cluster

If your EJBs will run on a WebLogic Server cluster, Oracle recommends that you deploy them homogeneously—to each Managed Server in the cluster. Alternatively, you can deploy an EJB to only to a single server in the cluster (that is, "pin" a module to a server). This type of deployment is less common, and should be used only in special circumstances where pinned services are required. See Understanding Cluster Configuration in *Administering Clusters for Oracle WebLogic Server*.

Deploying EJBs to a Cluster

During deployment, the uncompiled EJB is copied to every server instance in the cluster, but it is compiled only on the server instance to which it has been deployed. As a result, the server instances in the cluster to which the EJB was not targeted lack the classes necessary to invoke the EJB.

If you are deploying or redeploying an EJB to a single server instance in a cluster, a client can now invoke the EJB application through other servers in the cluster.

For information on pinned deployments, see Deploying to a Single Server Instance (Pinned Deployment) in *Administering Clusters for Oracle WebLogic Server*.

Redeploying an EJB

When you make changes to a deployed EJB's classes, you must redeploy the EJB. If you use automatic deployment, deployment occurs automatically when you restart WebLogic Server. Otherwise, you must explicitly redeploy the EJB.

Redeploying an EJB deployment enables an EJB provider to make changes to a deployed EJB's classes, recompile, and then "refresh" the classes in a running server.

When you redeploy, the classes currently loaded for the EJB are immediately marked as unavailable in the server, and the EJB's classloader and associated classes are removed. At the same time, a new EJB classloader is created, which loads and maintains the revised EJB classes.

When clients next acquire a reference to the EJB, their EJB method calls use the changed EJB classes.

You can redeploy an EJB that is standalone or part of an application using any of the administration tools listed in Summary of System Administration Tools and APIs in *Understanding Oracle WebLogic Server*. See Redeploying Applications in a Production Environment in *Deploying Applications to Oracle WebLogic Server*.

Production redeployment is not supported for:

- applications that use JTS drivers.

- applications that include EJB 1.1 container-managed persistence (CMP) EJBs. To use production redeployment with applications that include CMP EJBs, use EJB 2.x CMP instead of EJB 1.1 CMP.

For more information on production redeployment limitations, see Requirements and Restrictions for Production Redeployment in *Deploying Applications to Oracle WebLogic Server*.

Using FastSwap Deployment to Minimize Deployment

During iterative development of an EJB application, you make many modifications to the EJB implementation class file, typically redeploying an EJB module multiple times during its development.

Java EE 5 introduces the ability to redefine a class at runtime without dropping its ClassLoader or abandoning existing instances. This allows containers to reload altered classes without disturbing running applications, vastly speeding up iterative development cycles and improving the overall development and testing experiences.

With FastSwap, Java classes are redefined in-place without reloading the ClassLoader, thereby having the decided advantage of fast turnaround times. This means that you do not have to wait for an application to redeploy for your changes to take affect. Instead, you can make your changes, auto compile, and then see the effects immediately.

For more information about FastSwap, see Using FastSwap Deployment to Minimize Redeployment in *Deploying Applications to Oracle WebLogic Server*.

Understanding Warning Messages

To get information about a particular warning, use the `weblogic.GetMessage` tool. For example:

```
java weblogic.GetMessage -detail -id BEA-010202
```

Disabling EJB Deployment Warning Messages

You can disable certain WebLogic Server warning messages that occur during deployment. You may find this useful if the messages provide information of which you are already aware.

For example, if the methods in your EJB makes calls by reference rather than by value, WebLogic Server generates this warning during deployment: `"Call-by-reference not enabled."`

You can use the `disable-warning` element in `weblogic-ejb-jar.xml` to disable certain messages. For a list of messages you can disable, and instructions for disabling the messages, see *disable-warning* in the *weblogic-ejb-jar.xml Deployment Descriptor Reference* chapter of *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

6

Using an Embedded EJB Container in Oracle WebLogic Server

This chapter provides an overview of using an embeddable EJB container in Oracle WebLogic Server.
This chapter includes the following sections:

- Overview of the Embeddable EJB Container
- EJB Lite Functionality Supported in the Embedded EJB Container

Overview of the Embeddable EJB Container

Unlike traditional Java EE server-based execution, embeddable container usage allows client code and its corresponding enterprise beans to run in a Java SE environment without having to deploy them to a Java EE server. This provides better support for testing, offline processing (e.g., batch jobs), and the use of the EJB programming model in desktop applications.

Most of the services present in the enterprise bean container in a Java EE server are available in the embedded enterprise bean container, including injection, container-managed transactions, and security. Enterprise bean components execute similarly in both embedded and Java EE environments, and therefore the same enterprise bean can be easily reused in both standalone and networked applications.

For a detailed example of using the Embedded EJB container in a Java SE environment, see EJB 3.1: Example of Using the Embeddable EJB Container in Java SE.

EJB Lite Functionality Supported in the Embedded EJB Container

The EJB Lite subset of the EJB 3.2 API is supported in the Embedded EJB Container. EJB Lite is by definition a subset of functionality and doesn't describe any new feature or functionality. This section outlines the requirements of EJB Lite support as defined by the EJB 3.2 specification.

Table 6-1 represents the official requirements for EJB Lite functionality support as defined by the EJB 3.2 specification.

Table 6-1 Requirements for EJB Lite vs. EJB 3.2 Full

Requirements	EJB Lite	EJB 3.2 Full
Components		
Session Beans (stateful, stateless, singleton)	Yes	Yes
Message-Driven Beans	No	Yes
2.x/1.1 CMP/BMP Entity Beans	No	Yes
JPA 2.1	Yes	Yes

Table 6-1 (Cont.) Requirements for EJB Lite vs. EJB 3.2 Full

Requirements	EJB Lite	EJB 3.2 Full
Session Bean Client Views		
Local/No Interface	Yes	Yes
3.x Remote	No	Yes
2.x Remote Home/Component	No	Yes
JAX-WS Web Services Endpoint	No	Yes
Services		
EJB Timer Service (non-persistent)	Yes	Yes
Asynchronous Session Bean Invocations (local)	Yes	Yes
Interceptors	Yes	Yes
RMI-IIOP Interoperability	No	Yes
Container-managed Transactions/Bean-managed Transactions	Yes	Yes
Declarative and Programmatic Security	Yes	Yes
Miscellaneous		
Embeddable API	Yes	Yes

7

Configuring the Persistence Provider in Oracle WebLogic Server

This chapter describes Oracle TopLink, the default persistence provider in Oracle WebLogic Server, and introduces how to use it. This chapter also tells how to set the default persistence provider in WebLogic Server.
This chapter includes the following sections:

Overview of Oracle TopLink

Oracle TopLink is the default persistence provider in WebLogic Server 12*c* and later. It is a comprehensive standards-based object-persistence and object-transformation framework that provides APIs, schemas, and run-time services for the persistence layer of an application.

The core component of TopLink is the EclipseLink project's produced libraries and utilities. EclipseLink is the open source implementation of the development framework and the runtime provided in TopLink. EclipseLink implements the following specifications, plus value-added extensions:

- Jakarta Persistence 2.2 (JPA 2.2).

  JPA 2.2 includes new support or enhancements for features including Criteria Bulk Update/Delete, stored procedures, JPQL Generic function, injectable entity listeners, `TREAT`, converters, DDL generation, and entity graphs. For the complete JPA 2.2 specification, see `https://jakarta.ee/specifications/persistence/2.2/`.

- Java Architecture for XML Binding (JAXB) 2.2. (The EclipseLink JAXB implementation, plus EclipseLink extensions, is called MOXy.)

  For the JAXB 2.0 specification, see "JSR-000222 Java Architecture for XML Binding (JAXB) 2.0 " at `http://jcp.org/aboutJava/communityprocess/pfd/jsr222/index.html`.

- EclipseLink also includes Database Web Service (DBWS), which provides access to relational database artifacts by using a Java API for XML Web Services (JAX-WS) 2 Web service.

EclipseLink also provides support for Oracle Spatial and Oracle XDB mapping.

For more information about EclipseLink, including other supported services, see the EclipseLink project home at `http://wiki.eclipse.org/EclipseLink` and the EclipseLink Documentation Center at `http://wiki.eclipse.org/EclipseLink/UserGuide`.

In addition to all of EclipseLink, Oracle TopLink includes:

- TopLink Grid, an integration between EclipseLink JPA with Oracle Coherence that allows EclipseLink to use Oracle Coherence as a level 2 (L2) cache and persistence layer for entities. See *Developing Applications with Oracle Coherence*.

> **Note:**
>
> You must have a license for Oracle Coherence to be able to use TopLink Grid.

- Logging integration with WebLogic Server.
- MBean support in WebLogic Server.

For information about developing, deploying, and configuring Oracle TopLink applications, see the following:

- Understanding TopLink
- *Java API Reference for Oracle TopLink*

> **Note:**
>
> The preceding documents are for Oracle TopLink 12.1.3, but they also apply to the current version of WebLogic Server.

See also the following EclipseLink resources:

- EclipseLink Documentation Center at `http://www.eclipse.org/eclipselink/documentation/`
- EclipseLink examples at `http://wiki.eclipse.org/EclipseLink/Examples`.

Specifying a Persistence Provider

You can specify what persistence provider to use for a persistence unit in the application code or by accepting the default persistence provider set for the WebLogic Server domain, as described in the following sections:

- Setting the Default Provider for the Domain
- Specifying the Persistence Provider in an Application

Setting the Default Provider for the Domain

Unless you specify otherwise, TopLink is used as the default persistence provider for a WebLogic Server domain. The default provider is used for any entities in an application that are not configured to use a different persistence provider. The default provider is used for both injected and application-managed entity managers and factories.

You can set the default provider in the WebLogic Remote Console or by directly setting `JPAMBean.DefaultJPAProvider`.

> **Note:**
>
> Oracle Kodo JPA/JDO is not supported in this release of WebLogic Server. Customers are encouraged to use Oracle TopLink, which supports JPA 2.1. Kodo supports only JPA 1.0.

If you change the default provider, you must do the following for any deployed applications that do not specify a JPA provider:

- Restart applications that use application-managed entity manager factories.
- Redeploy applications that use injected entity manager factories or entity managers.

Specifying the Persistence Provider in an Application

A persistence provider specified in an application takes precedence over the default provider set for the WebLogic Server domain.

You can set the provider to use in the following ways:

- Specify the provider in the `<provider>` element for a persistence unit in the `persistence.xml` file, for example:

```
 <persistence-unit name="example">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
...
  </persistence-unit>
```

- Specify the provider in the `javax.persistence.provider` property passed to the `Map` parameter of the `javax.persistence.Persistence.createEntityManagerFactory(String, Map)` method.

A

EJB Metadata Annotations Reference

This appendix provides reference information for the EJB 3.*x* metadata annotations. This appendix includes the following sections:

Overview of EJB 3.x Annotations

The WebLogic Server EJB 3.2 programming model uses the Jakarta EE 8 metadata annotations feature in which you create an annotated EJB 3.2 bean file, and then compile the class with standard Java compiler, which can then be packaged into a target module for deployment. At runtime, WebLogic Server parses the annotations and applies the required behavioral aspects to the bean file.

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 Jakarta EE 8 components, and are thus in more generic packages, such as `javax.annotation`.

> **Note:**
>
> If you are continuing to use deployment descriptors in your EJB implementation, refer to EJB Deployment Descriptors in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Annotations for Stateless, Stateful, and Message-Driven Beans

This section provides reference information for the following annotations:

javax.ejb.AccessTimeout

The following sections describe the annotation in more detail.

Description

Target: Method, Type

Specifies the amount of time in a given time unit that a concurrent access attempt should block before timing out.

This annotation may be applied to a stateful session bean or to a singleton session bean that uses container managed concurrency.

By default, clients are allowed to make concurrent calls to a stateful session object and the container is required to serialize such concurrent requests. The `AccessTimeout` annotation is used to specify the amount of time a stateful session bean request should block in the case that the bean instance is already processing a different request. Use of the `AccessTimeout` annotation with a value of 0 specifies to the container that concurrent client requests to a stateful session bean are prohibited.

The `AccessTimeout` annotation can be specified on a business method or a bean class. If it is specified on a class, it applies to all business methods of that class. If it is specified on both a class and on a business method of the class, the method-level annotation takes precedence for the given method.

Access time-outs for a singleton session bean only apply to methods eligible for concurrency locks. The `AccessTimeout` annotation can be specified on the singleton session bean class or on an eligible method of the class. If `AccessTimeout` is specified on both a class and on a method of that class, the method-level annotation takes precedence for the given method.

For details, see Optionally Program the EJB Timer Service..

Attributes

The following table summarizes the attributes.

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

Name	Description	Data Type	Required?
value	Specifies the amount of time in a given time unit that a concurrent access attempt should block before timing out. • A value > 0 indicates a timeout value in the units specified by the unit element. • A value of 0 means concurrent access is not permitted. • A value of -1 indicates that the client request will block indefinitely until forward progress it can proceed. Values less than -1 are not valid.	Long	No
unit	Specifies the units used for the specified value. The default value for this attribute is `java.util.concurrent.TimeUnit.MILLISECONDS`.	TimeUnit	No

javax.ejb.ActivationConfigProperty

The following sections describe the annotation in more detail.

> **Note:**
>
> Based on the Enterprise JavaBean specification, the `javax.ejb.ActivationConfigProperty` annotation is used for MDBs only. This annotation is not used for session or entity beans.

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. The `ActivationConfigProperty` is used only for message-driven beans only; it is not used for session beans or entity beans.

This annotation is used only as a value to the `activationConfig` attribute of the `@javax.ejb.MessageDriven` annotation. For more information about this annotation, see Using EJB 3.2 Compliant MDBs and Deployment Elements and Annotations for MDBs in *Developing Message-Driven Beans for Oracle WebLogic Server*.

Attributes

The following table summarizes the attributes.

Table A-2 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.AfterBegin

The following sections describe the annotation in more detail.

Description

Target: Method

Designate a stateful session bean method to receive the after begin session synchronization callback.

The after begin callback notifies a stateful session bean instance that a new transaction has started and that the subsequent business methods on the instance will be invoked in the context of the transaction.

This method executes in the proper transaction context. A bean must have at most one `AfterBegin` method. The signature of this method must observe the following rules:

- The method must not be declared as `final` or `static`.
- The method may have any access type.
- The return type must be `void`.
- The method must take no arguments.

This method executes with no transaction context.

A stateful session bean class may use either the `SessionSynchronization` interface or the session synchronization annotations, but not both.

javax.ejb.AfterCompletion

The following sections describe the annotation in more detail.

- Description

Description

Target: Method

Designate a stateful session bean method to receive the after completion session synchronization callback.

The after completion callback notifies a stateful session bean instance that a transaction commit protocol has completed. A completion status of true indicates that the transaction has committed. A status of false indicates that a rollback has occurred.

A bean must have at most one `AfterCompletion` method. The signature of this method must observe the following rules:

- The method must not be declared as `final` or `static`.
- The method may have any access type.
- The return type must be `void`.
- The method must take a single argument of type `boolean`.

This method executes with no transaction context.

A stateful session bean class may use either the `SessionSynchronization` interface or the session synchronization annotations, but not both.

javax.ejb.ApplicationException

The following sections describe the annotation in more detail.

- Description
- Attributes

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

The following table summarizes the attributes.

Table A-3 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.Asynchronous

The following sections describe the annotation in more detail.

- Description

Description

Target: Method, Type

Used to mark a session bean method as an asynchronous method or to designate all business methods of a session bean class as asynchronous, where control is returned to the client by the enterprise bean container before the method is invoked on the session bean instance. Asynchronous methods are typically used for long-running operations, for processor-intensive tasks, for background tasks, to increase application throughput, or to improve application response time if the method invocation result isn't required immediately.

Clients calling asynchronous methods, immediately have control returned to them by the enterprise bean container. This allows the client to perform other tasks while the method invocation completes. If the method returns a result, the result is an implementation of the return type void or `java.util.concurrent.Future<V>` interface, where `V` is the result value type. The `Future<V>` interface defines methods the client may use to check if the computation is completed, wait for the invocation to complete, retrieve the final result, and cancel the invocation.

Asynchronous method invocation semantics only apply to the no-interface, local business, and remote business client views. Methods exposed through the EJB 2.x local, EJB 2.x remote, and Web service client views must not be designated as asynchronous.

javax.ejb.BeforeCompletion

The following sections describe the annotation in more detail.

- Description

Description

Target: Method

Designate a stateful session bean method to receive the before completion session synchronization callback.

The before completion callback notifies a stateful session bean instance that a transaction is about to be committed.

This method executes in the proper transaction context.

> **Note:**
>
> The instance may still cause the container to rollback the transaction by invoking the `setRollbackOnly()` method on the session context or by throwing an exception. A bean must have at most one `BeforeCompletion` method.

The signature of this method must observe the following rules:

- The method must not be declared as `final` or `static`.
- The method may have any access type.
- The return type must be `void`.
- The method must take no arguments.

This method executes with no transaction context.

A stateful session bean class may use either the `SessionSynchronization` interface or the session synchronization annotations, but not both.

javax.ejb.ConcurrencyManagement

The following sections describe the annotation in more detail.

Description

Target: Type

Declares a singleton session bean's concurrency management type.

If this annotation is not specified, the singleton bean is assumed to have container managed concurrency.

This annotation may be applied to stateful session beans, but doing so has no impact on the semantics of concurrency management for such beans. The concurrency management type for bean-managed concurrency (BEAN) does not apply to stateful session beans.

Attributes

The following table summarizes the attributes.

Table A-4 Attributes of the javax.ejb.ConcurrencyManagement Annotation

Name	Description	Data Type	Required?
value	Specifies the concurrency management type used by the bean class. Valid values for this attribute are: • `ConcurrencyManagementType.CONTAINER` • `ConcurrencyManagementType.BEAN` The default value for this attribute is `javax.ejb.ConcurrencyManagementType.CONTAINER`.	String	No.

javax.ejb.DependsOn

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Type

Used to express an initialization dependency between singleton components.

The container ensures that all singleton beans with which a singleton has a `DependsOn` relationship have been initialized before the singleton's `PostConstruct` method is called.

During application shutdown the container ensures that all singleton beans on with which the singleton has a `DependsOn` relationship are still available during the singleton's `PreDestroy` method.

Attributes

The following table summarizes the attributes.

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

Name	Description	Data Type	Required?
value	Specifies ejb-names of singleton components whose initialization must occur before this singleton. The order in which these names are listed is not significant.	String	No.

javax.ejb.EJB

The following sections describe the annotation in more detail.

- Description
- Attributes

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

The following table summarizes the attributes.

Table A-6 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. This name must be unique within the deployment unit, which consists of the class and its superclass.	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
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

Table A-6 (Cont.) Attributes of the javax.ejb.EJB Annotation

Name	Description	Data Type	Required?
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

The following sections describe the annotation in more detail.

- Description
- Attribute

Description

Target: Class

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

Attribute

The following table summarizes the attributes.

Table A-7 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

The following sections describe the annotation in more detail.

- Description
- Attributes

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

The following table summarizes the attributes.

Table A-8 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

The following sections describe the annotation in more detail.

- Description
- Attributes

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

The following table summarizes the attributes.

Table A-9 Attributes of the javax.ejb.Local

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.LocalBean

The following sections describe the annotation in more detail.

- Description

Description

Target: Type

Designates that a session bean exposes a no-interface view.

javax.ejb.LocalHome

The following sections describe the annotation in more detail.

- Description
- Attributes

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.x bean. You can also use this annotation with bean classes that have been written to the EJB 2.1 APIs.

Attributes

The following table summarizes the attributes.

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

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

javax.ejb.Lock

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Type, Method

Declares a concurrency lock for a singleton session bean with container managed concurrency.

This annotation may be specified on the bean class, the business methods of the bean class or both. Specifying the annotation on a business method overrides the value specified at class level, if any.

If this annotation is not used, a value of `Lock(WRITE)` is assumed.

Attributes

The following table summarizes the attributes.

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

Name	Description	Data Type	Required?
value	Specifies the concurrency lock used by the singleton session bean with container managed concurrency. Valid values for this attribute are: • `LockType.READ` • `LockType.WRITE` Default value is `javax.ejb.LockType.WRITE`.	String	No.

javax.ejb.MessageDriven

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class

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

Attributes

The following table summarizes the attributes.

Table A-12 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
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

Table A-12 (Cont.) Attributes of the javax.ejb.MessageDriven Annotation

Name	Description	Data Type	Required?
description	Specifies a description of the message-driven bean.	String	No

javax.ejb.PostActivate

The following sections describe the annotation in more detail.

- Description

Description

Target: Method

Specifies the life cycle 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

The following sections describe the annotation in more detail.

- Description

Description

Target: Method

Specifies the life cycle 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

The following sections describe the annotation in more detail.

- Description
- Attributes

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

The following table summarizes the attributes.

Table A-13 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

The following sections describe the annotation in more detail.

- Description
- Attributes

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.x bean. You can also use this annotation with bean classes that have been written to the EJB 2.1 APIs.

Attributes

The following table summarizes the attributes.

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

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

javax.ejb.Remove

The following sections describe the annotation in more detail.

- Description
- Attributes

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

The following table summarizes the attributes.

Table A-15 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.Schedule

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class

Schedule a timer for automatic creation with a timeout schedule based on a CRON-like time expression. The annotated method is used as the timeout callback method.

All elements of this annotation are optional. If none are specified a persistent timer will be created with callbacks occurring every day at midnight in the default time zone associated with the container in which the application is executing.

There are seven elements that constitute a schedule specification which are listed below. In addition, the `timezone` element may be used to specify a non-default time zone in whose context the schedule specification is to be evaluated; the persistent element may be used to specify a non-persistent timer, and the info element may be used to specify additional information that may be retrieved when the timer callback occurs.

- Calendar-based Schedule Elements
- Forms of Supported Element Values
- Additional Rules for Schedule Specification Elements

Calendar-based Schedule Elements

The elements that specify the calendar-based schedule itself are as follows:

- `second` – one or more seconds within a minute.

  Allowable values: [0,59]
- `minute` – one or more minutes within an hour.

  Allowable values: [0,59]
- `hour` – one or more hours within a day.

  Allowable values: [0,23]
- `dayOfMonth` – one or more days within a month.

  Allowable values:]

  - 1,31]
  - -7, -1
  - "Last"
  - {"1st", "2nd", "3rd", "4th", "5th", "Last"} {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}

  "Last" means the last day of the month.

  -x (where x is in the range [-7, -1]) means x day(s) before the last day of the month

  "1st","2nd", etc. applied to a day of the week identifies a single occurrence of that day within the month.

- `month` – one or more months within a year.

  Allowable values:]

  - [1,12]
  - {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", Dec"}

- `dayOfWeek` – one or more days within a week.

  Allowable values:]

  - [0,7]
  - {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}

  "0" and "7" both refer to Sunday

- `year` – a particular calendar year.

  Allowable values: a four-digit calendar year

Forms of Supported Element Values

Each element supports values expressed in one of the following forms:

- **Single Value** – Constrains the attribute to only one of its possible values.

  ```
  Example: second = "10"
  Example: month = "Sep"
  ```

- **Wild Card** – `"*"` Represents all allowable values for a given attribute.

  ```
  Example: second = "*"
  Example: dayOfWeek = "*"
  ```

- **List** – Constrains the attribute to two or more allowable values or ranges, with a comma used as a separator character within the string. Each item in the list must be a single value or range. List items cannot be lists, wild cards, or increments. Duplicate values are ignored.

  ```
  Example: second = "10,20,30"
  Example: dayOfWeek = "Mon,Wed,Fri"
  Example: minute = "0-10,30,40"
  ```

- **Range** – Constrains the attribute to an inclusive range of values, with a dash separating both ends of the range. Each side of the range must be a single attribute value. Members of a range cannot be lists, wild cards, ranges, or increments. If `x` is larger than `y` in a range `"x-y"`, the range is equivalent to `"x-max, min-y"`, where `max` is the largest value of the corresponding attribute and `min` is the smallest. The range `"x-x"`, where both range values are the same, evaluates to the single value `x`. The day of the week range `"0-7"` is equivalent to `"*"`.

```
Example: second = "1-10"
Example: dayOfWeek = "Fri-Mon"
Example: dayOfMonth = "27-3" (Equivalent to "27-Last , 1-3"
```

- **Increments** – The forward slash constrains an attribute based on a starting point and an interval, and is used to specify every `N` seconds, minutes, or hours within the minute, hour, or day, respectively. For the expression `x/y`, the attribute is constrained to every *yth* value within the set of allowable values beginning at time `x`. The `x` value is inclusive. The wild card character (`*`) can be used in the x position, and is equivalent to `0`. The use of increments is only supported within the `second`, `minute`, and `hour` elements. For the `second` and `minute` elements, `x` and `y` must each be in the range `[0,59]`. For the `hour` element, `x` and `y` must each be in the range `[0,23]`.

  ```
  Example: minute = "?/5" (Every five minutes within the hour)
  ```

  This is equivalent to: minute = "0,5,10,15,20,25,30,35,40,45,50,55"

  ```
  Example: second = "30/10" (Every 10 seconds within the minute, starting at second
  30)
  ```

  This is equivalent to: second = "30,40,50"

  Note that the set of matching increment values stops once the maximum value for that attribute is exceeded. It does not "roll over" past the boundary.

  ```
  Example : ( minute = "?/14", hour="1,2")
  ```

  This is equivalent to: (minute = "0,14,28,42,56", hour = "1,2") (Every 14 minutes within the hour, for the hours of 1 and 2 a.m.)

Additional Rules for Schedule Specification Elements

The following additional rules apply to the schedule specification elements.

- If the `dayOfMonth` element has a non-wildcard value and the `dayOfWeek` element has a non-wildcard value, then any day matching either the `dayOfMonth` value or the `dayOfWeek` value will be considered to apply.
- Whitespace is ignored, except for string constants and numeric values.
- All string constants (e.g., "Sun", "Jan", "1st", etc.) are case insensitive.

Schedule-based timer times are evaluated in the context of the default time zone associated with the container in which the application is executing. A schedule-based timer may optionally override this default and associate itself with a specific time zone. If the schedule-based timer is associated with a specific time zone, all its times are evaluated in the context of that time zone, regardless of the default time zone in which the container is executing.

The timeout callback method to which the `Schedule` annotation is applied must have one of the following signatures, where `<METHOD>` designates the method name:

```
void <METHOD>()
 void <METHOD>(Timer timer)
```

A timeout callback method can have public, private, protected, or package level access. A timeout callback method must not be declared as final or static. Timeout callback methods must not throw application exceptions.

Attributes

The following table summarizes the attributes.

Table A-16 Attributes of the javax.ejb.Schedule Annotation

Name	Description	Data Type	Required?
dayofMonth	Specifies one or more days within a month. Default value is `*`.	String	No
dayofWeek	Specifies one or more days within a week. Default value is `*`.	String	No
hour	Specifies one or more hours within a day. Default value is `0`.	String	No
info	Specifies an information string that is associated with the timer. Default value is `0`.	String	No
minute	Specifies one or more minutes within a hour. Default value is `0`.	String	No
month	Specifies one or more months within a year. Default value is `*`.	String	No
persistent	Specifies whether the timer that is created is persistent. Valid values for this attribute are `true` and `false`. Default value is `true`.	Boolean	No
second	Specifies one or more seconds with in a minute. Default value is `0`.	String	No
timezone	Specifies the time zone within which the schedule is evaluated. Time zones are specified as an ID string. The set of required time zone IDs is defined by the `Zone Name(TZ)` column of the public domain `zoneinfo` database. Default value: If a timezone is not specified, the schedule is evaluated in the context of the default timezone associated with the container in which the application is executing.	String	No
year	Specifies one or more years. Default value is `*`.	String	No

javax.ejb.Schedules

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Method

Applied to a timer callback method to schedule multiple calendar-based timers for the method. The method to which the `Schedules` annotation is applied must have one of the following signatures, where `<METHOD>` designates the method name:

```
void <METHOD>()
void <METHOD>(Timer timer)
```

Attributes

The following table summarizes the attributes.

Table A-17 Attributes of the javax.ejb.Schedules Annotation

Name	Description	Data Type	Required?
value	Specifies one or more calendar-based timer specifications.	Schedule[]	Yes

javax.ejb.Singleton

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class

Specifies that the Enterprise JavaBean is a singleton session bean.

Attributes

The following table summarizes the attributes.

Table A-18 Attributes of the javax.ejb.Singleton Annotation

Name	Description	Data Type	Required?
name	Specifies the name of the singleton session bean. If you do not specify this attribute, the default value is the unqualified name of the bean class.	String	No

Table A-18 (Cont.) Attributes of the javax.ejb.Singleton Annotation

Name	Description	Data Type	Required?
mappedName	Specifies the product-specific name to which the singleton session bean should be mapped. You can also use this attribute to specify the JNDI name of this singleton 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 singleton session bean may not be portable.	String	No
description	Describes the singleton session bean.	String	No.

javax.ejb.Startup

The following sections describe the annotation in more detail.

- Description

Description

Target: Class

Specifies that the Enterprise JavaBean is a stateful session bean.

javax.ejb.StatefulTimeout

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Type

Specifies the amount of time a stateful session bean can be idle (not receive any client invocations) before it is eligible for removal by the container.

Attributes

The following table summarizes the attributes.

Table A-19 Attributes of the javax.ejb.StatefulTimeout Annotation

Name	Description	Data Type	Required?
value	Specifies the amount of time the stateful session bean can be idle. • A value > 0 indicates a timeout value in the units specified by the unit element. • A value of 0 means concurrent access is not permitted. • A value of -1 indicates that the client request will block indefinitely until forward progress it can proceed. Values less than -1 are not valid.	Long	No
unit	Specifies the units used for the specified value. The default value for this attribute is `java.util.concurrent.TimeUnit.MINUTES`.	TimeUnit	No

javax.ejb.Stateless

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class

Specifies that the Enterprise JavaBean is a stateless session bean.

Attributes

The following table summarizes the attributes.

Table A-20 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

Table A-20 (Cont.) Attributes of the javax.ejb.Stateless Annotation

Name	Description	Data Type	Required?
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

The following sections describe the annotation in more detail.

- Description

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.x, 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.x 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.x way of programming the timer service if you want.

For details, see Optionally Program the EJB Timer Service.

This annotation does not have any attributes.

javax.ejb.TransactionAttribute

The following sections describe the annotation in more detail.

Description

Target: Class, Method

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

> **Note:**
>
> 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

The following table summarizes the attributes.

Table A-21 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 Transaction Elements table in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*. 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`.	String	No.

javax.ejb.TransactionManagement

The following sections describe the annotation in more detail.

- Description
- Attributes

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 in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Attributes

The following table summarizes the attributes.

Table A-22 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`.	String	No.

Annotations Used to Configure Interceptors

This section provides reference information for the annotations described in the following sections:

- javax.interceptor.AroundInvoke
- javax.interceptor.ExcludeClassInterceptors
- javax.interceptor.ExcludeDefaultInterceptors
- javax.interceptor.Interceptors

javax.interceptor.AroundInvoke

The following sections describe the annotation in more detail.

- Description

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

The following sections describe the annotation in more detail.

- Description

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

The following sections describe the annotation in more detail.

- Description

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

The following sections describe the annotation in more detail.

- Description
- Attributes

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 life cycle events on the bean.

The interceptor class can include both an business interceptor method (annotated with the `@javax.interceptor.AroundInvoke` annotation) and life cycle 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

The following table summarizes the attributes.

Table A-23 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 annotations described in the following sections:

- javax.persistence.PersistenceContext
- javax.persistence.PersistenceContexts
- javax.persistence.PersistenceUnit
- javax.persistence.PersistenceUnits

javax.persistence.PersistenceContext

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class, Method, Field

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

You use this annotation to interact with a 3.x 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

The following table summarizes the attributes.

Table A-24 Attributes of the javax.persistence.PersistenceContextAnnotation

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
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

javax.persistence.PersistenceContexts

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class

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

Attributes

The following table summarizes the attributes.

Table A-25 Attributes of the javax.persistence.PersistenceContexts Annotation

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

javax.persistence.PersistenceUnit

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class, Method, Field

Specifies a dependency on an `EntityManagerFactory` object.

You use this annotation to interact with a 3.x 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

The following table summarizes the attributes.

Table A-26 Attributes of the javax.persistence.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

Table A-26 (Cont.) Attributes of the javax.persistence.PersistenceUnit Annotation

Name	Description	Data Type	Required?
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

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class

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

Attributes

The following table summarizes the attributes.

Table A-27 Attributes of the javax.persistence.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.x

This section provides reference information about the annotations described in the following sections:

javax.annotation.PostConstruct

The following sections describe the annotation in more detail.

Description

Target: Method

Specifies the life cycle 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

The following sections describe the annotation in more detail.

Description

Target: Method

Specifies the life cycle 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

The following sections describe the annotation in more detail.

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

The following table summarizes the attributes.

Table A-28 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

Table A-28 (Cont.) Attributes of the javax.annotation.Resource Annotation

Name	Description	Data Type	Required?
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
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

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class

Specifies an array of `@Resource` annotations.

Attributes

The following table summarizes the attributes.

Table A-29 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.x

This section provides reference information about the annotations described in the following sections:

- javax.annotation.security.DeclareRoles
- javax.annotation.security.DenyAll
- javax.annotation.security.PermitAll
- javax.annotation.security.RolesAllowed
- javax.annotation.security.RunAs

javax.annotation.security.DeclareRoles

The following sections describe the annotation in more detail.

- Description
- Attributes

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 WebLogic Remote Console. See Security Roles in the *Oracle WebLogic Remote Console Online Help*.

Attributes

The following table summarizes the attributes.

Table A-30 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

The following sections describe the annotation in more detail.

- Description

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

The following sections describe the annotation in more detail.

- Description

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

The following sections describe the annotation in more detail.

- Description
- Attributes

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 WebLogic Remote Console. See Security Roles in the *Oracle WebLogic Remote Console Online Help*.

Attributes

The following table summarizes the attributes.

Table A-31 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

The following sections describe the annotation in more detail.

- Description
- Attributes

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. See Security Roles in the *Oracle WebLogic Remote Console Online Help*.

Attributes

The following table summarizes the attributes.

Table A-32 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.

WebLogic Annotations

This section provides reference information for the WebLogic annotations described in the following sections:

- weblogic.javaee.AllowRemoveDuringTransaction
- weblogic.javaee.CallByReference
- weblogic.javaee.DisableWarnings
- weblogic.javaee.EJBReference
- weblogic.javaee.Idempotent
- weblogic.javaee.JMSClientID
- weblogic.javaee.JNDIName
- weblogic.javaee.JNDINames
- weblogic.javaee.MessageDestinationConfiguration
- weblogic.javaee.TransactionIsolation

- weblogic.javaee.TransactionTimeoutSeconds

weblogic.javaee.AllowRemoveDuringTransaction

The following sections describe the annotation in more detail.

- Description

Description

Target: Class (Stateful session EJBs only)

Flag that specifies whether an instance can be removed during a transaction.

> **Note:**
>
> This annotation is overridden by the `allow-remove-during-transaction` element in the `weblogic-ejb-jar.xml` deployment descriptor. See weblogic-ejb-jar.xml Deployment Descriptor Reference in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

weblogic.javaee.CallByReference

The following sections describe the annotation in more detail.

- Description

Description

Target: Class (Stateful or stateless sessions EJBs only)

Flag that specifies whether parameters are copied—or passed by reference—regardless of whether the EJB is called remotely or from within the same EAR.

> **Note:**
>
> Method parameters are *always* passed by value when an EJB is called remotely. This annotation is overridden by the `enable-call-by-reference` element in the `weblogic-ejb-jar.xml` deployment descriptor. See weblogic-ejb-jar.xml Deployment Descriptor Reference in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

weblogic.javaee.DisableWarnings

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class

Specifies that WebLogic Server should disable the warning message whose ID is specified.

> **Note:**
>
> This annotation is overridden by the `disable-warning` element in the `weblogic-ejb-jar.xml` deployment descriptor. See weblogic-ejb-jar.xml Deployment Descriptor Reference in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Attributes

The following table summarizes the attributes.

Table A-33 Attributes of the weblogic.javaee.DisableWarnings

Name	Description	Data Type	Required?
WarningCode	Specifies the warning code. Set this element to one of the following four values: • `BEA-010001`—Disables this warning message: "EJB class loaded from system classpath during deployment." • `BEA-010054`—Disables this warning message: "EJB class loaded from system classpath during compilation." • `BEA-010200`—Disables this warning message: "EJB impl class contains a public static field, method or class." • `BEA-010202`—Disables this warning message: "Call-by-reference not enabled."	String	Yes

weblogic.javaee.EJBReference

The following sections describe the annotation in more detail.

- Description
- Attribute

Description

Target: Class, Method, Field

Maps EJB reference name to its JNDI name.

Attribute

The following table summarizes the attributes.

Table A-34 Attribute of the weblogic.javaee.EJBReference Annotation

Name	Description	Data Type	Required?
name	Specifies the name by which the referenced EJB is to be looked up in the environment. This name must be unique within the deployment unit, which consists of the class and its superclass.	String	Yes
jndiName	Specifies the JNDI name of an actual EJB, resource, or reference available in WebLogic Server.	String	Yes

weblogic.javaee.Idempotent

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class

Specifies an EJB that is written in such a way that repeated calls to the same method with the same arguments has exactly the same effect as a single call. This allows the failover handler to retry a failed call without knowing whether the call actually compiled on the failed server. When you enable idempotent for a method, the EJB stub can automatically recover from any failure as long as it can reach another server hosting the EJB.

> **Note:**
>
> This annotation is overridden by the `idempotent-method` and `retry-methods-on-rollback` elements in the `weblogic-ejb-jar.xml` deployment descriptor. See weblogic-ejb-jar.xml Deployment Descriptor Reference in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Attributes

The following table summarizes the attributes.

Table A-35 Attributes of the weblogic.javaee.Idempotent

Name	Description	Data Type	Required?
retryOnRollbackCount	Number of times to automatically retry container-managed transactions that have rolled back. This attribute defaults to 0.	int	No

weblogic.javaee.JMSClientID

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Method

Specifies a client ID for the MDB when it connects to a JMS destination. Required for durable subscriptions to JMS topics.

If you specify the connection factory that the MDB uses in weblogic.javaee.MessageDestinationConfiguration, the client ID can be defined in the `ClientID` element of the associated `JMSConnectionFactory` element in `config.xml`.

If `JMSConnectionFactory` in `config.xml` does not specify a `ClientID`, or if you use the default connection factory, (you do not specify weblogic.javaee.MessageDestinationConfiguration) the MDB uses the `jms-client-id` value as its client id.

> **Note:**
>
> This annotation is overridden by the `jms-client-id` element in the `weblogic-ejb-jar.xml` deployment descriptor. See weblogic-ejb-jar.xml Deployment Descriptor Reference in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Attributes

The following table summarizes the attributes.

Table A-36 Attributes of the weblogic.javaee.JMSClientID

Name	Description	Data Type	Required?
value	Client ID.	String	No
generateUniqueID	Flag that indicates whether or not you want the EJB container to generate a unique client ID for every instance of an MDB. Enabling this flag makes it easier to deploy durable MDBs to multiple server instances in a WebLogic Server cluster.	Class	No

weblogic.javaee.JNDIName

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class (Stateful or stateless session EJBs only)

Specifies a custom JNDI name that can be applied to a bean class for a certain client view. When applied to a bean class to indicate the JNDI name of a no-interface view, the `className` is optional.

> **Note:**
>
> This annotation is overridden by the `jndi-binding` element in the `weblogic-ejb-jar.xml` deployment descriptor. See weblogic-ejb-jar.xml Deployment Descriptor Reference in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Attributes

The following table summarizes the attributes.

Table A-37 Attributes of the weblogic.javaee.JNDIName

Name	Description	Data Type	Required?
classname	Class name of the client view.	String	No
value	JNDI name of the client view.	String	No

weblogic.javaee.JNDINames

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class (Stateful or stateless session EJBs only)

Specifies the multiple, custom JNDI names that can be applied to an EJB.

Attributes

The following table summarizes the attributes.

Table A-38 Attributes of the weblogic.javaee.JNDINames

Name	Description	Data Type	Required?
value	Multiple, custom JNDI names for the EJB.	JNDIName	No

weblogic.javaee.MessageDestinationConfiguration

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class (Message-driven EJBs only)

Specifies the JNDI name of the JMS Connection Factory that a message-driven EJB looks up to create its queues and topics.

> **Note:**
>
> This annotation is overridden by the `connection-factory-jndi-name` element in the `weblogic-ejb-jar.xml` deployment descriptor. See weblogic-ejb-jar.xml Deployment Descriptor Reference in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Attributes

The following table summarizes the attributes.

Table A-39 Attributes of the weblogic.javaee.MessageDestinationConfiguration

Name	Description	Data Type	Required?
connectionFactoryJNDIName	Connection factory JNDI name. This attribute defaults to an empty string.	String	No
initialContextFactory	WebLogic initial context factory. This attribute defaults to `weblogic.jndi.WLInitialContextFactory.class`.	Class	No
providerURL	URL of the provider. This attribute defaults to `t3://localhost:7001`.	String	No

weblogic.javaee.TransactionIsolation

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Method

Method-level transaction isolation settings for an EJB.

> **Note:**
>
> This annotation is overridden by the `trans-timeout-seconds` element in the `weblogic-ejb-jar.xml` deployment descriptor. See weblogic-ejb-jar.xml Deployment Descriptor Reference in *Developing Enterprise JavaBeans, Version 3.2, for Oracle WebLogic Server*.

Attributes

The following table summarizes the attributes.

Table A-40 Attributes of the weblogic.javaee.Idempotent

Name	Description	Data Type	Required?
IsolationLevel	Isolation level. Valid values include: • READ_COMMITTED—Transaction can view only committed updates from other transactions. • READ_UNCOMITTED—Transactions can view uncomitted updates from other transactions. • REPEATABLE_READ—Once the transaction reads a subset of data, repeated reads of the same data return the same values, even if other transactions have subsequently modified the data. • SERIALIZABLE—Simultaneously executing this transaction multiple times has the same effect as executing the transaction multiple times in a serial fashion. This attribute defaults to DEFAULT.	int	No

weblogic.javaee.TransactionTimeoutSeconds

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Class

Defines the timeout for transactions in seconds.

Attributes

The following table summarizes the attributes.

Table A-41 Attributes of the weblogic.javaee.TransactionTimeoutSeconds

Name	Description	Data Type	Required?
value	Transaction timeout value in seconds. This attribute defaults to 30 (seconds).	int	No