Oracle® Fusion Middleware

Developing Jakarta Enterprise Beans for Oracle WebLogic Server



26 (26.1.0.0.0)
G56951-01
November 2026



Oracle Fusion Middleware Developing Jakarta Enterprise Beans for Oracle WebLogic Server, 26 (26.1.0.0.0)

G56951-01

Copyright © 2007, 2026, Oracle and/or its affiliates.

Primary Author: Oracle Corporation

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

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.

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Contents

Preface

Audience i
Documentation Accessibility i
Diversity and Inclusion i
Related Documentation ii
Conventions ii

1 Understanding EJBs

New Features and Changes in EJB 1
Understanding EJB Components 1
- Session EJBs Implement Business Logic 1
  - Stateful Session Beans 2
  - Stateless Session Beans 2
  - Singleton Session Beans 2
- Message-Driven Beans Implement Loosely Coupled Business Logic 3

EJB Anatomy and Environment 3
- EJB Components 3
- The EJB Container 4
- EJB Metadata Annotations 4
- Optional EJB Deployment Descriptors 5

EJB Clients and Communications 5
- Accessing EJBs 5
- EJB Communications 6

Securing EJBs 6

2 Simple EJB Examples

Simple Java Examples of EJBs 1
- Example of a Simple No-interface Stateless EJB 1
- Example of a Simple Business Interface Stateless EJB 2
- Example of a Simple Stateful EJB 3
- Example of an Interceptor Class 5

Packaged EJB 3.2 Examples in WebLogic Server 6

Copyright © 2007, 2026, Oracle and/or its affiliates.

EJB 3.2: Example of Using the Session Bean Lifecycle 6
EJB 3.2: Example of a Message-Driven Bean with No-Methods Listener 6
Packaged EJB 3.1 Examples in WebLogic Server 6
EJB 3.1: Example of a Singleton Session Bean 7
EJB 3.1: Example of an Asynchronous Method EJB 7
EJB 3.1: Example of a Calendar-based Timer EJB 7
EJB 3.1: Example of Simplified No-interface Programming and Packaging in a WAR File 8
EJB 3.1: Example of Using a Portable JNDI Name in an EJB 8
EJB 3.1: Example of Using the Embeddable EJB Container in Java SE 8
EJB 3.0: Example of Invoking an Entity From A Session Bean 9

3 Iterative Development of EJBs

Overview of the EJB Development Process 1
Create a Source Directory 2
Directory Structure for Packaging a JAR 3
Directory Structure for Packaging a WAR 3
Program the Annotated EJB Class 3
Program the EJB Interface 4
Accessing EJBs Using the No-Interface Client View 4
Accessing EJBs Using the Business Interface 4
Business Interface Application Exceptions 5
Using Generics in EJBs 5
Serializing and Deserializing Business Objects 6
Optionally Program Interceptors 6
Optionally Program the EJB Timer Service 7
Overview of the Timer Service 7
Calendar-based EJB Timers 8
Automatically-created EJB Timers 8
Non-persistent Timers 8
Clustered Versus Local EJB Timer Services 9
Clustered EJB Timer Services 9
Local EJB Timer Services 9
Configuring Clustered EJB Timers 10
Using Java Programming Interfaces to Program Timer Objects 11
EJB Timer-related Programming Interfaces 11
WebLogic Server-specific Timer-related Programming Interfaces 11
Programming Access to EJB Clients 14
Remote Clients 14
Local Clients 15
Looking Up EJBs From Clients 15
Using Dependency Injection 15

Copyright © 2007, 2026, Oracle and/or its affiliates.

Using the JNDI Portable Syntax 16
Customizing JNDI Names 16
Configuring EJBs to Send Requests to a URL 16
Specifying an HTTP Resource by URL 17
Specifying an HTTP Resource by Its JNDI Name 17
Accessing HTTP Resources from Bean Code 18
Configuring Network Communications for an EJB 18
Programming and Configuring Transactions 18
Programming Container-Managed Transactions 18
Configuring Automatic Retry of Container-Managed Transactions 19
Programming Bean-Managed Transactions 20
Programming Transactions That Are Distributed Across EJBs 22
Calling multiple EJBs from a client's transaction context 22
Using an EJB "Wrapper" to Encapsulate a Cross-EJB Transaction 22
Compile Java Source 22
Optionally Create and Edit Deployment Descriptors 23
Packaging EJBs 24
Packaging EJBs in a JAR 24
Packaging an EJB In a WAR 24
Deploying EJBs 25

4 Programming the Annotated EJB Class

Overview of Metadata Annotations and EJB Bean Files 1
Programming the Bean File: Requirements and Changes From EJB 2.x 1
Bean Class Requirements and Changes From EJB 2.x 2
Bean Class Method Requirements 3
Programming the Bean File 3
Typical Steps When Programming the Bean File 3
Specifying the Business and Other Interfaces 4
Specifying the Business Interface 4
Specifying the No-interface View 5
Specifying the Bean Type (Stateless, Singleton, Stateful, or Message-Driven) 6
Injecting Resource Dependency into a Variable or Setter Method 7
Using Jakarta Persistence Entities from a Session Bean 8
Injecting Persistence Context Using Metadata Annotations 8
Finding an Entity Using the EntityManager API 9
Creating and Updating an Entity Using EntityManager 10
Specifying Interceptors for Business Methods or Life Cycle Callback Events 12
Specifying Business or Life Cycle Interceptors: Typical Steps 12
Programming the Interceptor Class 13
Programming Business Method Interceptor Methods 13

Copyright © 2007, 2026, Oracle and/or its affiliates.

Programming Asynchronous Business Methods 14
Programming Life Cycle Callback Interceptor Methods 14
Specifying Default Interceptor Methods 16
Saving State Across Interceptors With the InvocationContext API 16
Programming Application Exceptions 17
Securing Access to the EJB 18
Specifying Transaction Management and Attributes 20

5 Deployment Guidelines for EJBs

Before You Deploy an EJB 1
Understanding and Performing Deployment Tasks 2
Deployment Guidelines for EJBs 2
Deploying Standalone EJBs as Part of an Enterprise Application 2
Deploying EJBs as Part of a Web Application 3
Deploying EJBs That Call Each Other in the Same Application 3
Switching Protocol Limitation 3
Deploying EJBs That Use Dependency Injection 3
Deploying Homogeneously to a Cluster 4
Deploying EJBs to a Cluster 4
Redeploying an EJB 4
Using FastSwap Deployment to Minimize Deployment 5
Understanding Warning Messages 5
Disabling EJB Deployment Warning Messages 5

6 Using an Embedded EJB Container in Oracle WebLogic Server

Overview of the Embeddable EJB Container 1
EJB Lite Functionality Supported in the Embedded EJB Container 1

7 Configuring the Persistence Provider in Oracle WebLogic Server

Overview of Oracle TopLink 1
Specifying a Persistence Provider 2
Setting the Default Provider for the Domain 3
Specifying the Persistence Provider in an Application 3

8 Using Jakarta Data Repositories and NoSQL Resource References

Use a Repository from an Enterprise Bean 1
Declare an EJB-Scoped NoSQL Reference 1

Copyright © 2007, 2026, Oracle and/or its affiliates.

Apply Security and Lifecycle Rules 2

A EJB Metadata Annotations Reference

Overview of EJB Annotations A-1
WebLogic Annotations A-1
weblogic.javaee.AllowRemoveDuringTransaction A-2
Description A-2
weblogic.javaee.CallByReference A-2
Description A-2
weblogic.javaee.DisableWarnings A-3
Description A-3
Attributes A-3
weblogic.javaee.EJBReference A-3
Description A-3
Attribute A-3
weblogic.javaee.Idempotent A-4
Description A-4
Attributes A-4
weblogic.javaee.JMSClientID A-4
Description A-5
Attributes A-5
weblogic.javaee.JNDIName A-5
Description A-5
Attributes A-6
weblogic.javaee.JNDINames A-6
Description A-6
Attributes A-6
weblogic.javaee.MessageDestinationConfiguration A-6
Description A-7
Attributes A-7
weblogic.javaee.TransactionIsolation A-7
Description A-7
Attributes A-8
weblogic.javaee.TransactionTimeoutSeconds A-8
Description A-8
Attributes A-8

Copyright © 2007, 2026, Oracle and/or its affiliates.

Preface

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

- Audience
- Documentation Accessibility
- Diversity and Inclusion
- Related Documentation
- Conventions

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 programming model, in particular the use of metadata annotations to simplify development. This document does not address EJB topics that are different between different versions of EJB, 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.

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 customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 EJBs, see *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.
- *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:

- The Jakarta Enterprise Beans 4.0 specification at `https://jakarta.ee/specifications/enterprise-beans/`
- The Enterprise Beans chapter of the Jakarta EE Tutorial.

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

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Copyright © 2007, 2026, Oracle and/or its affiliates.

1

Understanding EJBs

This chapter describes the new features and programming model of the EJB specification supported by WebLogic Server and also provides a basic overview EJB components, anatomy, and features.
This chapter includes the following sections:

- New Features and Changes in EJB
- Understanding EJB Components
- EJB Anatomy and Environment
- EJB Clients and Communications
- Securing EJBs

New Features and Changes in EJB

WebLogic Server 26.1.0.0.0 supports the Jakarta Enterprise Beans 4.0 specification.

> **Note**
>
> In EJB 4.0, the package namespace changed from `javax.*` to `jakarta.*`.
>
> For a complete list of changes and comparisons to previous EJB specifications, see What is New in This Release in the *Jakarta Enterprise Beans 4.0 Specification*.

Understanding EJB Components

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

- Session EJBs Implement Business Logic
- Message-Driven Beans Implement Loosely Coupled Business Logic

Session EJBs Implement Business Logic

Session beans implement business logic. There are three types of session beans: stateful, stateless, and singleton. The container serializes calls to each stateful and stateless bean instance, while using multiple instances to serve clients concurrently. A singleton has one instance per application in each JVM and supports concurrent client access; container-managed locks or bean-managed concurrency determine which calls may execute simultaneously on that instance.

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 Jakarta EE Tutorial.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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
- The EJB Container
- EJB Metadata Annotations
- Optional EJB Deployment Descriptors

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **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 4.0; 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 Jakarta Enterprise Beans Using Deployment Descriptors*.

Table 1-1 Supported Client Views in EJB

Client Views	Session Bean Types	Additional Required Classes
Remote Client	Stateful, Stateless, and Singleton session beans	Remote business interface that defines the business methods exposed through the remote view.
Local Client	Stateful, Stateless, and Singleton session beans	Local business interface that defines the business methods exposed through the local view.
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 programming model uses the Jakarta EE metadata annotations feature in which you create an annotated EJB 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Jakarta Enterprise Beans Using Deployment Descriptors*.

The WebLogic Server EJB container supports three deployment descriptors:

- `ejb-jar.xml`—The standard Jakarta 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.
- **CMP deployment descriptor**—The WebLogic Server-specific XML descriptor for container-managed persistence of entity beans. This guide uses `weblogic-cmp-jar.xml` as its example filename. WebLogic tooling also uses the conventional path `META-INF/weblogic-cmp-rdbms-jar.xml`. In `weblogic-ejb-jar.xml`, set each CMP bean's `entity-descriptor/persistence/persistence-use/type-storage` to the actual descriptor path within the EJB module, for example `META-INF/weblogic-cmp-jar.xml`. The configured path and packaged filename must match.

For descriptions of the WebLogic Server EJB deployment descriptors, refer to Deployment Descriptor Schema and Document Type Definitions Reference in *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.

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
- EJB Communications

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

lookup, using the Java Naming and Directory Interface (JNDI) syntax to find the enterprise bean instance.

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

Applications that run outside a Jakarta EE server-managed environment, such as Java SE applications, must perform an explicit lookup. JNDI supports a global syntax for identifying Jakarta 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 Jakarta EE Tutorial.

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 Jakarta Enterprise Beans Using Deployment Descriptors*.

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 Jakarta Enterprise Beans (EJBs) in *Developing Applications with the WebLogic Security Service* provides instructions for configuring authentication and authorization for EJBs.

Copyright © 2007, 2026, Oracle and/or its affiliates.

2

Simple EJB Examples

This chapter provides Java examples of the EJB programming model.
This chapter includes the following sections:

- Simple Java Examples of EJBs
- Packaged EJB 3.2 Examples in WebLogic Server
- Packaged EJB 3.1 Examples in WebLogic Server

Simple Java Examples of 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
- Example of a Simple Business Interface Stateless EJB
- Example of a Simple Stateful EJB
- Example of an Interceptor Class

Example of a Simple No-interface Stateless EJB

The EJB 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;

import jakarta.ejb.Stateless;

@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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 jakarta.ejb.Stateless;
import jakarta.interceptor.ExcludeDefaultInterceptors;
/**
 * Bean file that implements the Service business interface.
 * Class uses following EJB 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 jakarta.ejb.Stateless;
  import jakarta.interceptor.ExcludeDefaultInterceptors;
  ```

  The annotations that apply only to EJB are in the `jakarta.ejb` package. Annotations that can be used by other Jakarta EE components are in more generic packages, such `jakarta.interceptor` or `jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 jakarta.ejb.Stateful;
import jakarta.ejb.Remote;
import jakarta.ejb.EJB;
import jakarta.annotation.PreDestroy;
import jakarta.interceptor.Interceptors;
import jakarta.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 destroyed 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;
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
  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 jakarta.ejb.Stateful;
  import jakarta.ejb.Remote;
  import jakarta.ejb.EJB;

  import jakarta.annotation.PreDestroy;

  import jakarta.interceptor.Interceptors;
  import jakarta.interceptor.ExcludeClassInterceptors;
  ```

  The annotations that apply only to EJB are in the `jakarta.ejb` package. Annotations that can be used by other Jakarta EE components are in more generic packages, such `jakarta.interceptor` or `jakarta.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 a reference to the `ServiceBean` stateless session bean's `Service` business view. 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();
  ```


Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 jakarta.interceptor.AroundInvoke;
import jakarta.interceptor.InvocationContext;
import jakarta.ejb.PostActivate;
import jakarta.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 jakarta.interceptor.AroundInvoke;
  import jakarta.interceptor.InvocationContext;
  import jakarta.ejb.PostActivate;
  import jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

EJB 3.2: Example of Using the Session Bean Lifecycle

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 implements 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

```
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 JNDI Name in an EJB

This example demonstrates the portable JNDI naming support introduced in EJB 3.1. Portable JNDI defines the `java:global`, `java:app`, and `java:module` namespaces. The bundled servlet uses the module-scoped name `java:module/HelloBean` to look up an EJB in the same module.

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

3

Iterative Development of EJBs

This chapter describes the general EJB 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
- Create a Source Directory
- Program the Annotated EJB Class
- Program the EJB Interface
- Optionally Program Interceptors
- Optionally Program the EJB Timer Service
- Programming Access to EJB Clients
- Programming and Configuring Transactions
- Compile Java Source
- Optionally Create and Edit Deployment Descriptors
- Packaging EJBs
- Deploying EJBs

Overview of the EJB Development Process

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

The following section discusses the EJB programming model. For a comprehensive overview, see the Jakarta Enterprise Beans 4.0 Specification. *Developing Jakarta Enterprise Beans Using Deployment Descriptors* may also provide useful information.

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 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.	No separate interface file for a no-interface view. One `.java` file for each business interface.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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. Annotations may define this metadata instead, although deployment descriptors remain supported.	• `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 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 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 also use the optional EJB deployment descriptor, 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.

When a bean is defined with annotations, apply the component-defining annotation for its bean type: `@jakarta.ejb.Stateful`, `@jakarta.ejb.Stateless`, `@jakarta.ejb.Singleton`, or

Copyright © 2007, 2026, Oracle and/or its affiliates.

`@jakarta.ejb.MessageDriven`. A bean may instead be denoted by its bean type in `ejb-jar.xml`. Other annotations are optional when their defaults provide the required 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 no-interface local client view, introduced in EJB 3.1, 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 Jakarta EE Tutorial.

Accessing EJBs Using the Business Interface

The EJB 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.

A session bean's business interface can extend other interfaces and may define any methods needed by the business view. A message-driven bean does not expose a business interface or other direct client view. Instead, its bean class implements, or its metadata specifies, the message-listener interface for its messaging type, such as `jakarta.jms.MessageListener` for Jakarta Messaging.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> The only requirement for an EJB 3.0 or later business interface is that it must *not* extend `jakarta.ejb.EJBObject` or `jakarta.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 `jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 `@jakarta.interceptor.Interceptors({MyInterceptor.class})`.

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 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.

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 `jakarta.ejb.TimedObject` interface. The `jakarta.ejb.TimedObject` interface has a single method, the timer callback method `ejbTimeout`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- **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 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 entity beans.

Calendar-based timers can be created programmatically using the two methods in the `jakarta.ejb.TimerService` that accept a `jakarta.ejb.ScheduleExpression` as an argument. The `ScheduleExpression` is constructed and populated prior to creating the Timer. For creating automatic calendar-base timers, the `jakarta.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 "Calendar Based Time Expressions" in the Jakarta Enterprise Beans Specification at https://jakarta.ee/specifications/enterprise-beans/.

Automatically-created EJB Timers

The EJB 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 `jakarta.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 `jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> Non-persistent timers are not supported for 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 `jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **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 Jakarta Enterprise Beans Using Deployment Descriptors*.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 4.0 specification that you can use to program timers. For detailed information on these interfaces, refer to the EJB 4.0 specification. This section also provides details about the WebLogic Server-specific timer-related interfaces.

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

EJB Timer-related Programming Interfaces

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

Table 3-2 EJB Timer-related Programming Interfaces

Programming Interface	Description
`jakarta.ejb.ScheduleExpression`	Create calendar-based EJB Timer expressions.
`jakarta.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 `jakarta.ejb.Schedules` annotation.
`jakarta.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.
`jakarta.ejb.TimerService`	Create new EJB timers or access existing EJB timers for the EJB.
`jakarta.ejb.Timer`	Access information about a particular EJB timer.
`jakarta.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 timer-related programming interfaces, see the EJB 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

`weblogic.management.runtime.EJBTimerRuntimeMBean` interface is shown in Example 3-3.

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

```
package weblogic.management.runtime;

public interface EJBTimerRuntimeMBean extends RuntimeMBean {
    long getTimeoutCount();
    long getCancelledTimerCount();
    int getActiveTimerCount();
    int getDisabledTimerCount();
    void activateDisabledTimers();
}
```

- `weblogic.ejb.WLTimerService` interface—extends the `jakarta.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 `jakarta.ejb.TimerService`, see the EJB 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

```
package weblogic.ejb;

import java.io.Serializable;
import java.util.Date;
import jakarta.ejb.EJBException;
import jakarta.ejb.Timer;
import jakarta.ejb.TimerService;

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

    Timer createTimer(Date expiration, Serializable info,
        WLTimerInfo wlTimerInfo)
        throws IllegalArgumentException, IllegalStateException, EJBException;

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

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


Copyright © 2007, 2026, Oracle and/or its affiliates.

- `weblogic.ejb.WLTimerInfo` class—used in the `weblogic.ejb.WLTimerService` interface to pass WebLogic Server-specific configuration information for a timer. The `weblogic.ejb.WLTimerInfo` class 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 Class

```
public final class WLTimerInfo {
    public static final int REMOVE_TIMER_ACTION = 1;
    public static final int DISABLE_TIMER_ACTION = 2;
    public static final int SKIP_TIMEOUT_ACTION = 3;

    private int maxRetryAttempts = -1;
    private int maxTimeouts = 0;
    private long retryDelay = 0;
    private int timeoutFailureAction = DISABLE_TIMER_ACTION;

    public void setMaxRetryAttempts(int retries) {
        if (retries < -1) {
            throw new IllegalArgumentException(String.valueOf(retries));
        }
        maxRetryAttempts = retries;
    }

    public int getMaxRetryAttempts() {
        return maxRetryAttempts;
    }

    public void setRetryDelay(long millis) {
        if (millis < 0) {
            throw new IllegalArgumentException(String.valueOf(millis));
        }
        retryDelay = millis;
    }

    public long getRetryDelay() {
        return retryDelay;
    }

    public void setMaxTimeouts(int max) {
        if (max < 0) {
            throw new IllegalArgumentException(String.valueOf(max));
        }
        maxTimeouts = max;
    }

    public int getMaxTimeouts() {
        return maxTimeouts;
    }
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    public void setTimeoutFailureAction(int action) {
        if (action != REMOVE_TIMER_ACTION
                && action != DISABLE_TIMER_ACTION
                && action != SKIP_TIMEOUT_ACTION) {
            throw new IllegalArgumentException(String.valueOf(action));
        }
        timeoutFailureAction = action;
    }

    public int getTimeoutFailureAction() {
        return timeoutFailureAction;
    }
}
```

- `weblogic.ejb.WLTimer` interface—extends the `jakarta.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 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **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 Jakarta Enterprise Beans Using Deployment Descriptors*.

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 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 Jakarta Enterprise Beans Using Deployment Descriptors*.

- 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

bean's context. Dependency injection is simply an easier-to-program alternative to using the `jakarta.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:

- `@jakarta.ejb.EJB`—Specifies a dependency on another EJB.
- `@jakarta.annotation.Resource`—Specifies a dependency on an external resource, such as a JDBC data source 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 the EJB specification 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`.

- `java:global[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]` — `<app-name>` is present only for an EAR; `<module-name>` is required.
- `java:app/<module-name>/<bean-name>[!<fully-qualified-interface-name>]` — `<module-name>` is required, including for a standalone module.
- `java:module/<bean-name>[!<fully-qualified-interface-name>]` — use this namespace from a component in the same module.

The container registers a suffixed entry for each applicable client view. When the bean exposes only one applicable client interface, or only a no-interface view, it also registers the unsuffixed name.

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 Jakarta Enterprise Beans Using Deployment Descriptors*.

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 `lookup` element of `@Resource`.

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

1. In `ejb-jar.xml`, update `resource-ref` as follows:

```
<resource-ref>
  <res-ref-name>url/MyURL</res-ref-name>
  <res-type>java.net.URL</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

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 `lookup` 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`, update `resource-ref` as follows:

```
<resource-ref>
  <res-ref-name>url/MyURL1</res-ref-name>
  <res-type>java.net.URL</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    <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 `jakarta.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 Jakarta Enterprise Beans Using Deployment Descriptors*.

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`

Copyright © 2007, 2026, Oracle and/or its affiliates.

  - The `getUserTransaction` method of `jakarta.ejb.EJBContext`
  - Any method of `jakarta.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 Jakarta Enterprise Beans Using Deployment Descriptors*.

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Jakarta Enterprise Beans Using Deployment Descriptors*, 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("java:comp/UserTransaction")
  ```

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

  Alternatively, the bean code may call `EJBContext.getUserTransaction()`.

  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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

- 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 Jakarta Enterprise Beans Using Deployment Descriptors*.

  See Example 3-7 for a code sample.

Example 3-7 Setting Transaction Isolation Level in BMT

```
import jakarta.transaction.UserTransaction;
import java.sql.Connection;
import javax.naming.Context;
import weblogic.transaction.Transaction;
import weblogic.transaction.TransactionHelper;
import weblogic.transaction.TxConstants;

public final class BmtIsolationExample {
    public static void performWork(Context context) throws Exception {
        UserTransaction userTransaction = (UserTransaction)
            context.lookup("java:comp/UserTransaction");
        userTransaction.begin();
        boolean transactionActive = true;
        try {
            Transaction transaction = (Transaction)
                TransactionHelper.getTransactionHelper().getTransaction();
            transaction.setProperty(
                TxConstants.ISOLATION_LEVEL,
                Integer.valueOf(Connection.TRANSACTION_READ_COMMITTED));

            // Perform transaction work.
            userTransaction.commit();
            transactionActive = false;
        } catch (Exception failure) {
            if (transactionActive) {
                try {
                    userTransaction.rollback();
                } catch (Exception rollbackFailure) {
                    failure.addSuppressed(rollbackFailure);
                }
            }
            throw failure;
        }
    }
}
```

- 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Close connections promptly—A global transaction can enlist multiple XA-capable resources, including connections from multiple XA data sources, and WebLogic Server coordinates them with two-phase commit. Close each connection promptly so it can return to its pool. Non-XA resources have additional enlistment restrictions. See the WebLogic Server JDBC global-transactions documentation before combining them with other resources.

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 jakarta.transaction.*;
...
u = (UserTransaction) jndiContext.lookup("jakarta.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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 The javac Command Reference.
- `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 Jakarta 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.

Later versions of the EJB programming model continue to fully support the use of deployment descriptors, even though the standard Jakarta 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 `@jakarta.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 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 because currently they do not have any annotation equivalent.

Developing Jakarta Enterprise Beans Using Deployment Descriptors provides detailed information about creating and editing EJB deployment descriptors, both the Jakarta 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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*.

EJB now provides 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 an EJB In a WAR

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 `ejb-client.jar` in the web application's `WEB-INF/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 3.1 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **⚠ 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 a 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 Jakarta Enterprise Beans Using Deployment Descriptors* guide, which concentrates on the 2.x programming model.

Copyright © 2007, 2026, Oracle and/or its affiliates.

4

Programming the Annotated EJB Class

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

- Overview of Metadata Annotations and EJB Bean Files
- Programming the Bean File: Requirements and Changes From EJB 2.x
- Programming the Bean File

Overview of Metadata Annotations and EJB Bean Files

The WebLogic Server EJB programming model uses the Jakarta EE metadata annotations feature in which you create an annotated EJB 4.0 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 4.0 bean file is the core of your EJB. It contains the Java code that determines how your EJB behaves. The 4.0 bean file is an ordinary Java class file that implements an EJB business interface that outlines the business methods of your EJB. You then annotate the bean file with JDK 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 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, described in Iterative Development of EJBs.

See also the annotations described by the following specifications:

- Jakarta Enterprise Beans 4.0 Specification at `https://jakarta.ee/specifications/enterprise-beans/4.0/`
- Jakarta Annotations at `https://jakarta.ee/specifications/annotations/3.0/`

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

The requirements for programming the Jakarta Enterprise Beans 4.0 bean class file are essentially the same as the EJB 2.*x* requirements. This section briefly describes the basic

Copyright © 2007, 2026, Oracle and/or its affiliates.

mandatory requirements of the bean class, mostly for overview purposes, as well as changes in requirements between EJB 2.*x* and Jakarta Enterprise Beans 4.0.

See *Developing Jakarta Enterprise Beans Using Deployment Descriptors* for detailed information about the mandatory and optional requirements for programming the bean class.

- Bean Class Requirements and Changes From EJB 2.x
- Bean Class Method Requirements

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

The following bullets list the EJB requirements for programming a bean class, as well as the EJB 2.*x* requirements that no longer apply:

- The bean type must be specified either by exactly one component-defining annotation `@jakarta.ejb.Stateless`, `@jakarta.ejb.Stateful`, `@jakarta.ejb.Singleton`, or `@jakarta.ejb.MessageDriven`.

  Or, when the bean class has no component-defining annotation, by the corresponding session or message-driven entry in ejb-jar.xml. A deployment descriptor cannot override a component-defining annotation.

  > **Note**
  >
  > WebLogic Server ships the Jakarta Persistence 3.2 API and EclipseLink 5.0.1. EclipseLink is the default persistence provider unless the application selects another supported provider in `persistence.xml` or the domain configures a different default. A Jakarta Persistence entity is annotated with `@jakarta.persistence.Entity`.
  >
  > Do not confuse a Jakarta Persistence entity with an optional Enterprise Beans 2.x entity bean, which uses the `jakarta.ejb.EntityBean` component model and deployment descriptors.
  >
  > 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.
  - 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 `jakarta.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 `jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 `jakarta.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 class file.

- Typical Steps When Programming the Bean File
- Specifying the Business and Other Interfaces
- Specifying the Bean Type (Stateless, Singleton, Stateful, or Message-Driven)
- Injecting Resource Dependency into a Variable or Setter Method
- Using Jakarta Persistence Entities from a Session Bean
- Specifying Interceptors for Business Methods or Life Cycle Callback Events
- 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 bean file for an 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

1. Import the EJB and other common annotations that will be used in your bean file. The general EJB annotations are in the `jakarta.ejb` package, the interceptor annotations are in the `jakarta.interceptor` package, the Jakarta Persistence annotations are in the `jakarta.persistence` package, and the common annotations are in the `jakarta.annotation` or `jakarta.annotation.security` packages. For example:

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

2. Decide which client views the session bean exposes. A no-interface view uses the bean class and needs no separate Java interface. For each local or remote business view, provide the business interface and either implement it or declare it with `@Local` or `@Remote`.

   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 Jakarta EE object.

   See Injecting Resource Dependency into a Variable or Setter Method.

5. Optionally, inject an `EntityManager` or `EntityManagerFactory` and use the Jakarta Persistence API to manage entities.

   See Using Jakarta Persistence Entities from a Session Bean.

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.

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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

- By explicitly implementing the business interface, using the `implements` Java keyword.
- By using metadata annotations (such as `jakarta.ejb.Local` and `jakarta.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 `jakarta.ejb.Remote` annotation or updating the appropriate EJB deployment descriptor. You can specify the `jakarta.ejb.Remote` annotation, as well as the `jakarta.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 `jakarta.ejb.Local` or `jakarta.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 `jakarta.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 a session bean that exposes a local no-interface view is accomplished through dependency injection or JNDI lookup. Message-driven beans do not expose local, remote, or no-interface client views; clients send messages to the destination or endpoint serviced by the message-driven bean.

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

  ```
  @EJB
  ExampleBean exampleBean;
  ```

- Within a Jakarta EE component in the same module, obtain a no-interface view through the component naming context as follows:

  ```
  import javax.naming.InitialContext;
  import javax.naming.NamingException;

  public ExampleBean lookupExampleBean() throws NamingException {
      InitialContext context = new InitialContext();
      try {
          return (ExampleBean) context.lookup("java:module/ExampleBean");
      } finally {
  ```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
            context.close();
        }
    }
```

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 implementing the no-interface client view, see "Accessing Local Enterprise Beans Using the No-Interface View" in the "Enterprise Beans chapter of the Jakarta EE Tutorial.

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

For an annotation-defined bean class, specify exactly one component-defining annotation:

- `@jakarta.ejb.Stateless`—Specifies that you are programming a stateless session bean.
- `@jakarta.ejb.Singleton`—Specifies that you are programming a singleton session bean.
- `@jakarta.ejb.Stateful`—Specifies that you are programming a stateful session bean.
- `@jakarta.ejb.MessageDriven`—Specifies that you are programming a message-driven bean.

> **Note**
>
> WebLogic Server ships the Jakarta Persistence 3.2 API and EclipseLink 5.0.1. EclipseLink is the default persistence provider unless the application selects another supported provider in `persistence.xml` or the domain configures a different default. A Jakarta Persistence entity is annotated with `@jakarta.persistence.Entity`.
>
> Do not confuse a Jakarta Persistence entity with an optional Enterprise Beans 2.x entity bean, which uses the `jakarta.ejb.EntityBean` component model and deployment descriptors.
>
> See Configuring the Persistence Provider in Oracle WebLogic Server.

Alternatively, declare the bean type and class in ejb-jar.xml without a component-defining annotation.

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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

- `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 `jakarta.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:

- `@jakarta.ejb.EJB`—Specifies a dependency on another EJB.
- `@jakarta.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 Jakarta EE components.

Both annotations have an equivalent grouping annotation to specify a dependency on multiple resources (`@jakarta.ejb.EJBs` and `@jakarta.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 `jakarta.ejb.EJB` and `jakarta.annotation.Resource` in jakarta.ejb Package for detailed information on the attributes and when you should specify them.

The following code snippet shows how to use the `@jakarta.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 jakarta.ejb.EJB;
...
@Stateful
public class AccountBean
 implements Account
{
  @EJB(beanName="ServiceBean")
  private Service service;
...
  public void sayHelloFromAccountBean() {
    service.sayHelloFromServiceBean();
  }
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

In the preceding example, the private variable `service` is annotated with the `@jakarta.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 exposes the business methods declared by the `Service` business interface and implemented by `ServiceBean`, including the `sayHelloFromServiceBean` method invoked by `sayHelloFromAccountBean`.

Using Jakarta Persistence Entities from a Session Bean

This section describes how a session bean uses the Jakarta Persistence API to find, create, and update entities.

> **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 Jakarta 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 `jakarta.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 Jakarta Persistence query-language (JPQL) 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 to inject a container-managed `EntityManager` or an `EntityManagerFactory`:

- `@jakarta.persistence.PersistenceContext` expresses a dependency on a container-managed `EntityManager` and its associated persistence context. Its optional `unitName` selects a persistence unit defined in `persistence.xml`.
- `@jakarta.persistence.PersistenceContexts` is the repeatable-container annotation for one or more `@PersistenceContext` declarations.
- `@jakarta.persistence.PersistenceUnit` expresses a dependency on an `EntityManagerFactory` for a persistence unit.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- `@jakarta.persistence.PersistenceUnits` is the repeatable-container annotation for one or more `@PersistenceUnit` declarations.

Use `@PersistenceContext` when the container should manage the `EntityManager`. Use `@PersistenceUnit` when application code needs an `EntityManagerFactory` and will create application-managed `EntityManager` instances. Application-managed instances created with `EntityManagerFactory.createEntityManager()` must be closed with `EntityManager.close()` in a `finally` block or equivalent cleanup path. Application code must not close a container-managed `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 jakarta.ejb.Stateless;

import jakarta.persistence.PersistenceContext;
import jakarta.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:

```
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionAttribute;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import java.util.List;
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
import static jakarta.ejb.TransactionAttributeType.REQUIRED;

@Stateless
@TransactionAttribute(REQUIRED)
public class PersonService {
    @PersistenceContext
    private EntityManager em;

    public List<Person> findPeopleNamedPatrick() {
        return em.createQuery(
                "SELECT p FROM Person p WHERE p.name = :name", Person.class)
            .setParameter("name", "Patrick")
            .getResultList();
    }

    public Person createPerson(String name, int age) {
        Address address = new Address();
        Person person = new Person(name, age);
        person.setAddress(address);
        em.persist(address);
        em.persist(person);
        return person;
    }

    public Person changeAge(int id, int newAge) {
        Person person = em.find(Person.class, id);
        if (person == null) {
            throw new IllegalArgumentException("No Person with id " + id);
        }
        person.setAge(newAge);
        return person;
    }

    public Person applyOfflineChanges(Person detachedPerson) {
        return em.merge(detachedPerson);
    }
}
```

`createQuery` executes JPQL. Use `createNativeQuery` only when native SQL is intended. In `createPerson`, both `Address` and `Person` are newly constructed transient entity instances. Because this example does not assume `cascade=PERSIST`, it explicitly persists both. With `cascade=PERSIST` on `Person.address`, persisting `Person` may cascade to the new `Address`. `persist` makes each new instance managed. Changes to the managed `Person` returned by `find` are detected automatically. `applyOfflineChanges` requires a detached `Person`. `merge` copies its state into a managed instance and returns that managed instance, so callers should use the returned object. Inserts and updates are synchronized with the database when the persistence context is flushed, normally before transaction commit, not necessarily at the Java method call.

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) {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
  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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Specifying Interceptors for Business Methods or Life Cycle Callback Events

Jakarta Interceptors can interpose on business-method invocations (`@AroundInvoke`), timeout invocations (`@AroundTimeout`), constructor invocations (`@AroundConstruct`), and lifecycle callback events.

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 `@jakarta.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 `jakarta.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 `@jakarta.interceptor.Interceptors` annotation to associate the interceptor class with the bean class. The method in the interceptor class annotated with the `@jakarta.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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Life Cycle Callback Interceptor Methods .
5. In the bean class, optionally annotate methods with the `@jakarta.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 `@jakarta.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 `@jakarta.interceptor.Interceptors` 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
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

An `@AroundInvoke` method normally calls `InvocationContext.proceed()` to invoke the next interceptor or target business method and returns that result. It may intentionally omit `proceed()` to short-circuit the invocation; in that case, no remaining interceptor or target method is invoked, and the interceptor supplies the result or throws an exception. If it calls `proceed()`, it may wrap the call in `try/catch/finally` to inspect, transform, suppress, or rethrow a permitted result or exception. 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 Jakarta EE Tutorial.

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:

- `@jakarta.ejb.PrePassivate`—Specifies the method that the EJB container notifies when it is about to passivate a stateful session bean.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- `@jakarta.ejb.PostActivate`—Specifies the method that the EJB container notifies right after it has reactivated a stateful session bean.
- `@jakarta.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 `jakarta.annotation` package, rather than `jakarta.ejb`.

- `@jakarta.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 `jakarta.annotation` package, rather than `jakarta.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 `@jakarta.interceptor.Interceptors` 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 `jakarta.ejb.PostActivate`, `jakarta.ejb.PrePassivate`, `jakarta.annotation.PostConstruct`, and `jakarta.annotation.PreDestroy` in jakarta.ejb Package and jakarta.annotation Package for additional requirements when programming the life cycle interceptor class.

For an example, see Example of an Interceptor Class.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 `@jakarta.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="4.0"
     xmlns="https://jakarta.ee/xml/ns/jakartaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/
jakartaee/ejb-jar_4_0.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 `jakarta.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.

`Around-invoke`, `around-timeout`, `around-construct`, and lifecycle interceptor methods take one `InvocationContext` parameter. A lifecycle callback method declared directly on the target class instead has no `InvocationContext` parameter.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 4-1 Interceptor Methods

Method	Description
`Object getTarget()`	Returns the intercepted target; before `proceed()` in `@AroundConstruct`, it returns `null`.
`Object getTimer()`	Returns the timer for `@AroundTimeout`, otherwise `null`.
`Method getMethod()`	Returns the intercepted target method, or `null` when no corresponding method exists and for `@AroundConstruct`.
`Constructor<?> getConstructor()`	Returns the intercepted constructor for `@AroundConstruct`, otherwise `null`.
`Object[] getParameters()` / `void setParameters(Object[] parameters)`	Reads or replaces method/constructor parameters; these operations are illegal in lifecycle callbacks other than `@AroundConstruct`.
`Map<String,Object> getContextData()`	Returns mutable data shared across the invocation chain.
`Object proceed() throws Exception`	Continues to the next interceptor or target operation.
`Set<Annotation> getInterceptorBindings()`	Returns the immutable set of interceptor bindings (Interceptors 2.2).
`<T extends Annotation> T getInterceptorBinding(Class<T> type)`	Returns the single binding of a type, or `null` (Interceptors 2.2).
`<T extends Annotation> Set<T> getInterceptorBindings(Class<T> type)`	Returns all bindings of a repeatable type (Interceptors 2.2).

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 exception, then you must program the exception as a separate class from the bean class.

Use the `@jakarta.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 jakarta.ejb.ApplicationException;

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


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
 */

@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:

- `jakarta.annotation.security.DeclareRoles`—Explicitly lists the security roles that will be used to secure the EJB. Role names used in `@RolesAllowed` are implicitly declared. Use `@DeclareRoles`, or `security-role-ref` in `ejb-jar.xml`, for role names tested with `EJBContext.isCallerInRole` that are not otherwise declared.
- `jakarta.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.)
- `jakarta.annotation.security.DenyAll`—Specifies that the annotated method can not be invoked by any role.
- `jakarta.annotation.security.PermitAll`—Specifies that the annotated method can be invoked by all roles.
- `jakarta.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 Jakarta EE components that allow metadata annotations, not just EJB.

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 jakarta.ejb.Stateless;

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


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
/**
 * 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 jakarta.annotation.security.DeclareRoles;
  import jakarta.annotation.security.PermitAll;
  import jakarta.annotation.security.DenyAll;
  import jakarta.annotation.security.RolesAllowed;
  import jakarta.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 after the original caller is authorized to invoke a method, outbound calls made by `ServiceBean` use the principal mapped to the `admin` run-as role. The annotation does not grant the original caller the admin role and does not change the caller reported by `getCallerPrincipal` or tested by `isCallerInRole`.
- 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Specifying Transaction Management and Attributes

By default, session beans and message-driven beans use container-managed transaction demarcation. For a bean with container-managed transactions, a business method whose transaction attribute is not specified uses `TransactionAttributeType.REQUIRED`. Use `@TransactionManagement(BEAN)` only when the bean is to demarcate its own transactions, and use `@TransactionAttribute` only when an explicit container-managed transaction attribute is needed or improves clarity.

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

- `jakarta.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).
- `jakarta.ejb.TransactionAttribute`—Specifies whether the EJB container invokes methods within a transaction.

For an example of using the `jakarta.ejb.TransactionAttribute` annotation, see Example of a Simple Stateful EJB.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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
- Understanding and Performing Deployment Tasks
- Deployment Guidelines for EJBs

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 a Web Application.

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

- Programmed the required annotated EJB class to specify the type of EJB—either: `@jakarta.ejb.Stateful`, `@jakarta.ejb.Stateless`, `@jakarta.ejb.Singleton`, or `@jakarta.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 Jakarta Enterprise Beans Using Deployment Descriptors*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 a 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 deployment unit that bundles together Web applications, EJBs, and Resource Adapters into a single deployable unit.

This is an 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 a 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)
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    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 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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 introduced 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 effect. 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 make 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 Jakarta Enterprise Beans Using Deployment Descriptors*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Jakarta 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 Jakarta 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 Jakarta 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 Jakarta 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

WebLogic Server's embeddable enterprise bean container implements the Enterprise Beans Lite feature group defined by Jakarta Enterprise Beans 4.0. Enterprise Beans Lite is a subset of the Enterprise Beans API, not a separate product or API. An embeddable container is an optional Enterprise Beans 4.0 feature. When provided, it must support the Lite group.

#unique_151/unique_151_Connect_42_CHDDHHDH represents the official requirements for EJB Lite functionality support as defined by the EJB 3.2 specification.

The third column reports the API-group classification in the Jakarta Enterprise Beans 4.0 specification; it is not a statement of Jakarta EE 11 platform requirements. Jakarta EE 11 removed the Enterprise Beans 2.x API group and Jakarta XML Web Services from the platform. Jakarta EE 10 had already removed entity beans and the embeddable Enterprise Beans Container from the platform.

For WebLogic-specific support, the embeddable provider documented by this chapter implements the Lite group. Separately, WebLogic Server ships a Jakarta XML Web Services 4.0 stack and retains its documented compatibility features. Jakarta Persistence is a separate specification rather than an Enterprise Beans Lite feature.

Copyright © 2007, 2026, Oracle and/or its affiliates.

WebLogic Server supplies Jakarta Persistence 3.2 with EclipseLink 5.0.1 as its default provider.

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

Feature group	Enterprise Beans Lite member?	Enterprise Beans 4.0 specification API group
Stateful, stateless, and singleton session beans	Yes	Required by the full EJB API
Local business interface and no-interface view	Yes	Required by the full EJB API
Container-managed and bean-managed transactions	Yes	Required by the full EJB API
Declarative and programmatic security	Yes	Required by the full EJB API
Interceptors	Yes	Required by the full EJB API
Local asynchronous session bean invocations	Yes	Required by the full EJB API
Non-persistent Enterprise Beans Timer Service	Yes	Required by the full EJB API
Message-driven beans	No	Required by the full EJB API
Enterprise Beans 3.x remote business view	No	Required by the full EJB API
Persistent Enterprise Beans Timer Service	No	Required by the full EJB API
Jakarta XML Web Services endpoints	No	Required by the full EJB API
Enterprise Beans 2.x API	No	Optional API group
2.x/1.x CMP/BMP entity beans and Enterprise Beans QL	No	Optional API group
Embeddable Enterprise Beans Container	Not a Lite member; its provider must implement Lite	Optional API group

Copyright © 2007, 2026, Oracle and/or its affiliates.

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
- Specifying a Persistence Provider

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 (JPA)

  For the complete JPA specification, see https://jakarta.ee/specifications/persistence/.

- Jakarta XML Binding (JAXB). The EclipseLink JAXB implementation, plus EclipseLink extensions, is called MOXy.

  For the JAXB specification, see https://jakarta.ee/specifications/xml-binding/.

- EclipseLink also includes Database Web Service (DBWS), which provides access to relational database artifacts by using a Jakarta XML Web Services 4.0 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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*

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.

Oracle Database Identity Column Compatibility with EclipseLink in WebLogic Server 26.1.0.0.0

In WebLogic Server 26.1.0.0.0, applications that use EclipseLink with Oracle Database should not rely on the EclipseLink discovery of Oracle system-managed identity sequences for entities mapped with `GenerationType.IDENTITY`. If an entity maps its primary key to an Oracle column defined as `GENERATED ALWAYS AS IDENTITY`, EclipseLink may attempt to insert an explicit primary-key value, causing Oracle Database to reject the insert with the `ORA-32795` database error message.

To migrate these applications, use an application-owned sequence and configure the entity with `@SequenceGenerator` and `GenerationType.SEQUENCE` as demonstrated in Example 7-1. The sequence must be initialized above the current maximum primary-key value. The table definition must also allow application-provided primary-key values, such as by using `GENERATED BY DEFAULT AS IDENTITY` or another DBA-approved schema migration. Do not rely on Oracle's system-managed identity sequence as the application sequence.

Example 7-1 Configuring an entity with `@SequenceGenerator` and `GenerationType.SEQUENCE`

```
@Id
  @SequenceGenerator(
      name = "usersSeq",
      sequenceName = "USERS_ID_SEQ",
      allocationSize = 1
  )
  @GeneratedValue(
      strategy = GenerationType.SEQUENCE,
      generator = "usersSeq"
  )
  private Long id;
```

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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`.

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 `jakarta.persistence.provider` property passed to the `Map` parameter of the `jakarta.persistence.Persistence.createEntityManagerFactory(String, Map)` method.

> **Note**
>
> WebLogic Server 26.1.0.0.0 uses Jakarta Persistence 3.2 as the default for Jakarta EE 11 applications. If you want your applications to continue using the Jakarta Persistence 3.0 schema, explicitly declare `version="3.0"` and reference `persistence_3_0.xsd` in `xsi:schemaLocation`, as demonstrated in Example 7-2.
>
> Update the descriptor to version 3.2 only when the application intentionally adopts Jakarta Persistence 3.2 descriptor features.

Example 7-2 Configure applications to use the Jakarta Persistence 3.0 schema

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence https://
jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    ...
</persistence>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

8

Using Jakarta Data Repositories and NoSQL Resource References

Learn how to use Jakarta Data repositories with NoSQL resources in Oracle WebLogic Server and how application resource references map to configured NoSQL system resources.

- Use a Repository from an Enterprise Bean
  An enterprise bean can inject an eligible Jakarta Data repository. Discovery includes supported EJB modules within an EAR, subject to CDI eligibility, visibility, and the owning deployment's supported scope.
- Declare an EJB-Scoped NoSQL Reference
  For a NoSQL repository, declare a local reference of type `org.eclipse.jnosql.communication.semistructured.DatabaseManager` for the consuming bean. Supply the corresponding explicit JNDI mapping under the same effective EJB identity.
- Apply Security and Lifecycle Rules
  Configure `NoSQLResource access` for the global management-resource name. WebLogic Server evaluates the current subject on guarded use. The deployer or original handle recipient does not permanently authorize later calls. EJB roles, targeting, and JNDI mappings do not substitute for the resource policy. Calling-application ownership also applies to retained capabilities.

Use a Repository from an Enterprise Bean

An enterprise bean can inject an eligible Jakarta Data repository. Discovery includes supported EJB modules within an EAR, subject to CDI eligibility, visibility, and the owning deployment's supported scope.

See About Jakarta Data Applications in *Developing Jakarta Data Applications for Oracle WebLogic Server*.

Jakarta Persistence-backed repository operations use the selected visible persistence unit and applicable JTA transaction contract. NoSQL-backed repository operations do not participate in JTA global transactions, including when called from a bean with container-managed transactions. Do not infer NoSQL XA enlistment or global rollback from the EJB transaction attribute.

Declare an EJB-Scoped NoSQL Reference

For a NoSQL repository, declare a local reference of type `org.eclipse.jnosql.communication.semistructured.DatabaseManager` for the consuming bean. Supply the corresponding explicit JNDI mapping under the same effective EJB identity.

The following table identifies the required descriptor relationship.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Descriptor	Required Content	Identity Check
`META-INF/ejb-jar.xml`	Standard `resource-ref` or supported `resource-env-ref` under the consuming bean	Local alias and full DatabaseManager type
`META-INF/weblogic-ejb-jar.xml`	Matching `resource-description` or `resource-env-description` under `weblogic-enterprise-bean`	Same `ejb-name`, alias, and reference kind
Global NoSQL descriptor	One valid matching JNDI binding targeted locally	Intended global management-resource identity

Another bean's mapping or a sibling WAR cannot supply a missing same-scope mapping. Matching alias text alone does not bridge EJB identities.

For illustrative descriptor fragments, see Package an EJB Binding in *Administering NoSQL Resources for Oracle WebLogic Server*. Merge the fragments into the application's descriptors using the release roots, namespaces, schemas, and element order.

Complete packaged pairs need no plan. Environment-specific overrides remain subject to the same EJB identity, declaration/type, and mapping rules. Review effective descriptors after applying the plan. See Apply a Deployment-Plan Override in *Administering NoSQL Resources for Oracle WebLogic Server*.

Apply Security and Lifecycle Rules

Configure `NoSQLResource access` for the global management-resource name. WebLogic Server evaluates the current subject on guarded use. The deployer or original handle recipient does not permanently authorize later calls. EJB roles, targeting, and JNDI mappings do not substitute for the resource policy. Calling-application ownership also applies to retained capabilities.

Repositories in one deployment using the WebLogic NoSQL system-resource path must select one bound global identity. The binding remains fixed for that deployment generation; compatible property updates can advance configuration generations behind stable handles. Reacquire former-generation retained capabilities.

If resource changes terminally invalidate the deployment, restore the intended configuration and fully redeploy the affected application. For an EJB within an EAR, do not assume a module-only refresh establishes the required new application generation. See *Deploying Applications to Oracle WebLogic Server* and Update a Resource and Recover Applications in *Administering NoSQL Resources for Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

A

EJB Metadata Annotations Reference

This appendix provides reference information for the EJB metadata annotations.

- Overview of EJB Annotations
- WebLogic Annotations

Overview of EJB Annotations

The WebLogic Server EJB programming model uses the Jakarta EE metadata annotations feature in which you create an annotated EJB 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.

For detailed reference information regarding which metadata annotations you can specify in the EJB bean file, see the package summaries in Jakarta EE Platform API.

Some of the annotations are in the `jakarta.ejb` package, and are thus specific to EJBs; others are more common and are used by other Jakarta EE components, and are thus in more generic packages, such as `jakarta.annotation`. See:

- jakarta.ejb Package
- jakarta.interceptor Package
- jakarta.persistence Package
- jakarta.annotation Package
- jakarta.annotation.security Package

> **Note**
>
> If you are continuing to use deployment descriptors in your EJB implementation, refer to EJB Deployment Descriptors in *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.

WebLogic Annotations

This section provides reference information for WebLogic annotations.

- weblogic.javaee.AllowRemoveDuringTransaction
- weblogic.javaee.CallByReference
- weblogic.javaee.DisableWarnings
- weblogic.javaee.EJBReference
- weblogic.javaee.Idempotent

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 Jakarta Enterprise Beans Using Deployment Descriptors*.

weblogic.javaee.CallByReference

The following sections describe the annotation in more detail.

- Description

Description

Target: Class (stateful, stateless, or singleton session beans only)

Enables call-by-reference semantics for eligible invocations within the same application. It does not change remote invocation semantics; arguments that cross a remote boundary are passed by value.

> **Note**
>
> 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 Jakarta Enterprise Beans Using Deployment Descriptors*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Jakarta Enterprise Beans Using Deployment Descriptors*.

Attributes

The following table summarizes the attributes.

Table A-1 Attributes of the weblogic.javaee.DisableWarnings

Name	Description	Data Type	Required?
value	Warning codes to disable: `WarningCode.BEA_010001`, `WarningCode.BEA_010054`, `WarningCode.BEA_010200`, and/or `WarningCode.BEA_010202`	WarningCode[]	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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-2 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	No Defaults to an empty string.
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: Method (session-bean or message-driven-bean implementation methods)

Specifies an EJB method that is idempotent. This allows the failover handler to retry a failed call without knowing whether the call actually completed 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 Jakarta Enterprise Beans Using Deployment Descriptors*.

Attributes

The following table summarizes the attributes.

Table A-3 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Description
- Attributes

Description

Target: Class (message-driven EJBs only)

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 Jakarta Enterprise Beans Using Deployment Descriptors*.

Attributes

The following table summarizes the attributes.

Table A-4 Attributes of the weblogic.javaee.JMSClientID

Name	Description	Data Type	Required?
value	Client ID.	String	Yes
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.	Boolean	No Defaults to `false`.

weblogic.javaee.JNDIName

The following sections describe the annotation in more detail.

- Description
- Attributes

Description

Target: Type (a stateful, stateless, or singleton session-bean class, or one of its business interfaces)

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Jakarta Enterprise Beans Using Deployment Descriptors*.

Attributes

The following table summarizes the attributes.

Table A-5 Attributes of the weblogic.javaee.JNDIName

Name	Description	Data Type	Required?
className	Client-view class. Defaults to `Object.class`; it may be omitted only when the bean exposes exactly one client view.	Class<?>	No
value	JNDI name of the client view.	String	Yes

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-6 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 Jakarta Enterprise Beans Using Deployment Descriptors*.

Attributes

The following table summarizes the attributes.

Table A-7 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 an empty string. When it is empty, the annotation processor does not set provider-url.	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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> This annotation is overridden by the `transaction-isolation` element in the `weblogic-ejb-jar.xml` deployment descriptor. A matching method entry in the descriptor takes precedence. See weblogic-ejb-jar.xml Deployment Descriptor Reference in *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.

Attributes

The following table summarizes the attributes.

Table A-8 Attributes of the weblogic.javaee.TransactionIsolation

Name	Description	Data Type	Required?
value	Isolation level. Valid values include: • READ_COMMITTED—Transaction can view only committed updates from other transactions. • READ_UNCOMIMTTED—Transactions can view uncommitted 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.	int	Yes

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-9 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

Copyright © 2007, 2026, Oracle and/or its affiliates.