Developer's Guide to Enterprise JavaBeans Technology

Sun ONE Application Server

Version 7, Update 1

817-2175-10
March 2003

Contents

About This Guide

This *Developer's Guide to Enterprise Java Beans Technology* describes how to create and implement Java 2 Platform, Enterprise Edition (J2EE) applications that follow the Enterprise JavaBeans™ (EJB™) specification in the Sun™ Open Network Environment (ONE) Application Server 7 environment. In addition to briefly describing EJB programming concepts and tasks, this guide offers sample code, implementation tips, and reference material.

This preface addresses the following topics:

- Who Should Use This Guide
- Using the Documentation
- How This Guide Is Organized
- Related Information
- Documentation Conventions
- Product Support

Who Should Use This Guide

The intended audience for this guide is the person who develops, assembles, and deploys beans in a corporate enterprise.

This guide assumes you are familiar with the following topics:

- Java programming
- Java APIs as defined in the EJB, Java Server Page (JSP), and Java Database Connectivity (JDBC) specifications
- The SQL structured database query languages

- Relational database concepts
- Software development processes, including debugging and source code control

Using the Documentation

The Sun ONE Application Server manuals are available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML) formats, at:

`http://docs.sun.com/`

The following table lists tasks and concepts described in the Sun ONE Application Server manuals. The left column lists the tasks and concepts, and the right column lists the corresponding manuals.

Sun ONE Application Server Documentation Roadmap

For information about	See the following
Late-breaking information about the software and the documentation	*Release Notes*
Supported platforms and environments	*Platform Summary*
Introduction to the application server, including new features, evaluation installation information, and architectural overview.	*Getting Started Guide*
Installing Sun ONE Application Server and its various components (sample applications, Administration interface, Sun ONE Message Queue).	*Installation Guide*
Creating and implementing J2EE applications that follow the open Java standards model on the Sun ONE Application Server 7. Includes general information about application design, developer tools, security, assembly, deployment, debugging, and creating lifecycle modules.	*Developer's Guide*
Creating and implementing J2EE applications that follow the open Java standards model for web applications on the Sun ONE Application Server 7. Discusses web application programming concepts and tasks, and provides sample code, implementation tips, and reference material.	*Developer's Guide to Web Applications*

Sun ONE Application Server Documentation Roadmap *(Continued)*

For information about	See the following
Creating and implementing J2EE applications that follow the open Java standards model for enterprise beans on the Sun ONE Application Server 7. Discusses EJB programming concepts and tasks, and provides sample code, implementation tips, and reference material.	*Developer's Guide to Enterprise JavaBeans Technology*
Creating Web Services, RMI-IIOP, or other clients that access J2EE applications on the Sun ONE Application Server 7	*Developer's Guide to Clients*
J2EE features such as JDBC, JNDI, JTS, JMS, JavaMail, resources, and connectors	*Developer's Guide to J2EE Features and Services*
Creating custom NSAPI plugins	*Developer's Guide to NSAPI*
Performing the following administration tasks:	*Administrator's Guide*
• Using the Administration interface and the command line interface	
• Configuring server preferences	
• Using administrative domains	
• Using server instances	
• Monitoring and logging server activity	
• Configuring the web server plugin	
• Configuring the Java Messaging Service	
• Using J2EE features	
• Configuring support for CORBA-based clients	
• Configuring database connectivity	
• Configuring transaction management	
• Configuring the web container	
• Deploying applications	
• Managing virtual servers	
Editing server configuration files	*Administrator's Configuration File Reference*
Configuring and administering security for the Sun ONE Application Server 7 operational environment. Includes information on general security, certificates, and SSL/TLS encryption. Web-core-based security is also addressed.	*Administrator's Guide to Security*

Sun ONE Application Server Documentation Roadmap *(Continued)*

For information about	See the following
Configuring and administering service provider implementation for J2EE CA connectors for the Sun ONE Application Server 7. Includes information about the Administration Tool, DTDs and provides sample XML files.	*J2EE CA Service Provider Implementation Administrator's Guide*
Migrating your applications to the new Sun ONE Application Server 7 programming model from the Netscape Application Server version 2.1, including a sample migration of an Online Bank application provided with Sun ONE Application Server	*Migration Guide*
Using Sun ONE Message Queue.	The Sun ONE Message Queue documentation at: http://docs.sun.com/?p=/coll/S1_MessageQueue_30

How This Guide Is Organized

This guide contains the following documentation components:

- "Introducing the Sun ONE Application Server Enterprise JavaBeans Technology"
- "Using Session Beans"
- "Using Entity Beans"
- "Using Container-Managed Persistence for Entity Beans"
- "Using Message-Driven Beans"
- "Handling Transactions with Enterprise Beans"
- "Developing Secure Enterprise Beans"
- "Assembling and Deploying Enterprise Beans"
- "CMP Mapping with the Sun ONE Studio 4 Interface"
- "Elements Listings"

Related Information

In addition to the information in the Sun ONE Application Server documentation collection listed in "Using the Documentation," on page 12, the following resources may be helpful:

- J2EE Specifications

  `http://java.sun.com/products/`

- Enterprise JavaBeans Specification, Version 2.0

  `http://java.sun.com/products/ejb/docs.html#specs`

- General EJB product information:

  `http://java.sun.com/products/ejb`

- Java Software tutorials:

  `http://java.sun.com/j2ee/docs.html`

- *Enterprise JavaBeans*, by Richard Monson-Haefel, O'Reilly Publishing, ISBN 0-596-00226-2

  `http://www.oreilly.com/catalog/entjbeans3/`

- Enterprise Beans Technology book index

  http://developer.java.sun.com/developer/Books/ejbtechnology.html

- *Enterprise JavaBeans Design Patterns*, ISBN 0-471-20831-0

- *Core J2EE Patterns*, ISBN 0-13-064884-1

Documentation Conventions

This section describes the types of conventions used throughout this guide:

- General Conventions
- Conventions Referring to Directories

General Conventions

The following general conventions are used in this guide:

- **File and directory paths** are given in UNIX® format (with forward slashes separating directory names). For Windows versions, the directory paths are the same, except that backslashes are used to separate directories.
- **URLs** are given in the format:

  http://*server.domain*/*path*/*file*.html

  In these URLs, *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server's directory structure; and *file* is an individual filename. Italic items in URLs are placeholders.
- **Font conventions** include:
  - The `monospace` font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.
  - *Italic* type is used for code variables.
  - *Italic* type is also used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
  - **Bold** type is used as either a paragraph lead-in or to indicate words used in the literal sense.
- **Installation root directories** for most platforms are indicated by *install_dir* in this document. Exceptions are noted in "Conventions Referring to Directories." on page 17.

  By default, the location of *install_dir* on **most** platforms is:
  - Solaris 8 non-package-based Evaluation installations:

    user's home directory`/sun/appserver7`
  - Solaris unbundled, non-evaluation installations:

    `/opt/SUNWappserver7`
  - Windows, all installations:

    `C:\Sun\AppServer7`

  For the platforms listed above, *default_config_dir* and *install_config_dir* are identical to *install_dir*. See "Conventions Referring to Directories" on page 17 for exceptions and additional information.
- **Instance root directories** are indicated by *instance_dir* in this document, which is an abbreviation for the following:

  default_config_dir`/domains/`*domain*`/`*instance*

- **UNIX-specific descriptions** throughout this manual apply to the Linux operating system as well, except where Linux is specifically mentioned.

Conventions Referring to Directories

By default, when using the Solaris 8 and 9 package-based installation and the Solaris 9 bundled installation, the application server files are spread across several root directories. These directories are described in this section.

- **For Solaris 9 bundled installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
  - *install_dir* refers to `/usr/appserver/`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
  - *default_config_dir* refers to `/var/appserver/domains`, which is the default location for any domains that are created.
  - *install_config_dir* refers to `/etc/appserver/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.
- **For Solaris 8 and 9 package-based, non-evaluation, unbundled installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
  - *install_dir* refers to `/opt/SUNWappserver7`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
  - *default_config_dir* refers to `/var/opt/SUNWappserver7/domains`which is the default location for any domains that are created.
  - *install_config_dir* refers to `/etc/opt/SUNWappserver7/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.

NOTE Forte for Java 4.0 has been renamed to Sun ONE Studio 4 throughout this manual.

Product Support

If you have problems with your system, contact customer support using one of the following mechanisms:

- The online support web site at:

  `http://www.sun.com/supporttraining/`

- The telephone dispatch number associated with your maintenance contract

Please have the following information available prior to contacting support. This helps to ensure that our support staff can best assist you in resolving problems:

- Description of the problem, including the situation where the problem occurs and its impact on your operation
- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem
- Detailed steps on the methods you have used to reproduce the problem
- Any error logs or core dumps

Chapter 1

Introducing the Sun ONE Application Server Enterprise JavaBeans Technology

This section provides an overview of how the Java Enterprise Edition (J2EE) Enterprise JavaBeans™ (EJB™) technology works in the application programming model of the Sun™ ONE Application Server 7.

NOTE If you are unfamiliar with the EJB technology, refer to the Java Software tutorials:

`http://java.sun.com/j2ee/docs.html`

and the J2EE specifications:

`http://java.sun.com/products/`

Overview material on the Sun ONE Application Server is contained in the *Sun ONE Application Server Product Introduction*.

This section addresses the following topics:

- Summary of EJB 2.0 Changes
- EJB Architecture
- Value Added Features
- About Enterprise JavaBeans
- About Developing an Effective Application
- About EJB Assembly and Deployment

Relevant files supplied with the Sun ONE Application Server are contained in the following locations:

- Sun ONE Application Server DTD files:

  install_dir`/apperv/lib/dtds`

- Sun ONE Application Server sample applications:

  `i`*nstall_dir*`/apperv/samples`

Summary of EJB 2.0 Changes

Sun ONE Application Server supports the Sun Microsystems Enterprise JavaBeans (EJBs) architecture as defined by the Enterprise JavaBeans Specification, v2.0 and is compliant with the Enterprise JavaBeans Specification, v1.1.

NOTE You can deploy existing 1.1 beans in the Sun ONE Application Server, but we recommend that new beans be developed as 2.0 enterprise bean.

This section summarizes the changes in the Enterprise JavaBeans Specification, v2.0 that impact enterprise beans in the Sun ONE Application Server environment:

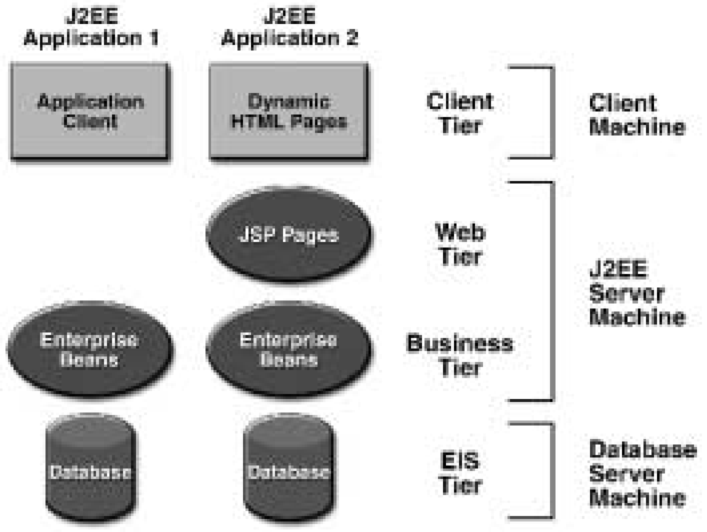
- Container-managed persistence—Provides a new way of handling container-managed persistence. See "Using Container-Managed Persistence for Entity Beans," on page 79."
- Container-managed relationships—Allows you to define relationships between entity beans. See "Assembling and Deploying Enterprise Beans," on page 169.
- Message-driven beans—This new type of enterprise bean is a Java Message Service consumer. "Using Message-Driven Beans," on page 129.
- Local interfaces—Session and entity beans can implement a local interface. Container-managed EJB relationships are now based on the local interface. See "Creating a Local Interface," on page 65.
- Additional methods on the home interface—Allow you to implement business logic that is independent of a specific entity bean instance. See "Creating the Remote Home Interface," on page 61.

- New query language (EJB QL)—The new EJB Query Language (EJB QL) provides for navigation across a network of entity beans defined by container-managed relationships. See "Using EJB QL," on page 105.

EJB Architecture

The Sun ONE Application Server reduces the complexity of developing middleware by providing automatic support for middleware services such as transactions, security, database connectivity, and more.

The following figure illustrates where enterprise beans fit in the J2EE environment. In this figure the client machine is running a web browser or application client, the J2EE server machine is running (or hosting) the Sun ONE Application Server, and the database server machine hosts the databases, such as Oracle and LDAP. Enterprise beans reside in the business tier, with JSPs (and servlets) providing the interface to the client tier, and the Sun ONE Application Server managing the relationships between the client and database machines.

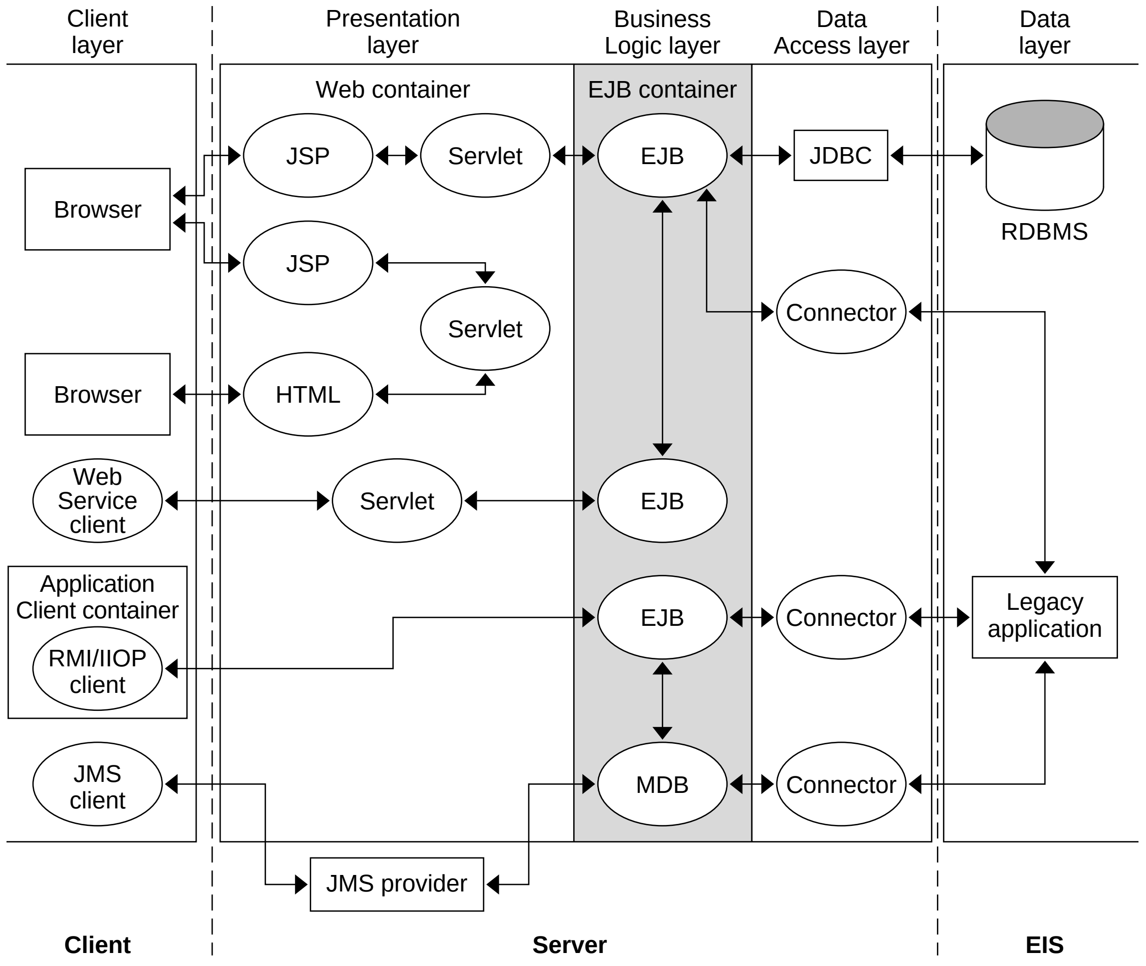


The Sun ONE Application Server is responsible for providing the base of the EJB execution systems, which include:

- A standard set of EJB services

- Distributed transaction management services
- A means of data store access or backend system connection
- An EJB container to implement the management and control services for the EJB classes

The following figure illustrates further details of the J2EE environment. The business logic layer shows the EJB flow.



Value Added Features

The Sun ONE Application Server provides a number of value additions that relate to EJB development. These capabilities are discussed in the following sections (references to more in-depth material are included):

- Read-Only Beans
- pass-by-reference
- Pooling and Caching Features
- Monitoring
- Integration with Sun ONE Studio 4
- Dynamic Deployment and Reloading

Read-Only Beans

Another feature that the Sun ONE Application Server provides is the *read-only bean*, an entity bean that is never modified by an EJB client. Read-0nly beans avoid database updates completely.

A read-only bean can be used to cache a database entry that is frequently accessed but rarely updated (externally by other beans). When the data that is cached by a read-only bean is updated by another bean, the read-only bean can be notified to refresh its cached data.

The Sun ONE Application Server provides a number of ways by which a read-only bean's state can be refreshed. By setting the `refresh-period-in-seconds` element and the transaction attribute of the bean, it is easy to configure a read-only bean that is (a) always refreshed, (b) periodically refreshed, (c) never refreshed, or (d) programatically refreshed.

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. For further information and usage guidelines, see "Read-Only Beans," on page 59.

pass-by-reference

The `pass-by-reference` element in the `sun-ejb-jar.xml` file allows you to specify the passing method / argument type used by enterprise beans. This is an opportunity to improve performance. See "pass-by-reference," on page 186.

Pooling and Caching Features

The Sun ONE Application Server provides a highly configurable bean pooling mechanism that allows the deployer to configure bean pools according to the needs of the enterprise.

In addition, the Sun ONE Application Server supports a number of tunable parameters that can be used to control the number of beans cached as well as the duration they are cached. Multiple bean instances that refer to the same database row in a table can be cached.

Refer to “Pooling and Caching,” on page 31 for information on this functionality.

Monitoring

The Sun ONE Application Server supports monitoring of many aspects of the runtime environment, including various elements of the EJB container which can be useful for debugging your application's correctness as well as tuning its performance.

See the *Sun ONE Application Server Administrator’s Guide* (Monitoring and Managing Sun One Application Server section) and the *Performance, Tuning, and Sizing Guide* for more information on monitoring.

Integration with Sun ONE Studio 4

Sun ONE Studio 4, Enterprise Edition for Java (formerly Forte for Java (FFJ), Enterprise Edition), is an integrated development environment (IDE) that allows you to create, assemble, deploy, and debug code in the Sun ONE Application Server from a single, easy-to-use interface. Behind the scenes, a plugin integrates the Sun ONE Studio 4 IDE with the Sun ONE Application Server.

For more information about using the Sun ONE Studio 4, see the Sun ONE Studio 4, Enterprise Edition tutorial and “CMP Mapping with the Sun ONE Studio 4 Interface,” on page 217.

Dynamic Deployment and Reloading

You can deploy, redeploy, and undeploy an application or standalone module. If this is done while the server is running, it is considered *dynamic*. The following dynamic processes are available in Sun ONE Application Server:

- Dynamic reloading—Enables reloading the classes that constitute an application when they change on disk.
- Dynamic redeployment (for the developer community)—Enables redeploying an existing application without restarting the server. You can ALSO disable and enable an application or module without undeploying it.

For more information on dynamic deployment, refer to the Sun ONE Application Server *Developer's Guide* and *Administrator's Guide*.

About Enterprise JavaBeans

If you are already familiar with enterprise beans and how they work, you may prefer to proceed to "About Developing an Effective Application," on page 34

The following topics are discussed in this section:

- What Is an Enterprise JavaBean?
- Types of Beans
- EJB Flow
- The EJB Container
- Interfaces
- Pooling and Caching
- How Enterprise Beans Access Resources
- Transaction Management
- How Application Security Works

What Is an Enterprise JavaBean?

An *enterprise bean*, or Enterprise JavaBean (EJB), is a self-contained, reusable component that has data members, properties, and methods. Each enterprise bean encapsulates one or more application tasks or objects, including data structures and operation methods.

- Enterprise bean methods can take parameters and send back return values.
- Enterprise bean creation and management is handled at runtime by the container.

- Client access mediation is handled by the container and the server where the bean is deployed.
- Enterprise beans are restricted to using standard container services defined by the Enterprise JavaBeans Specification, v2.0. This guarantees that the bean is portable and deployable in any EJB-compliant container.
- Enterprise beans are components that can be assembled, without recompiling, into a composite application.
- A client's bean definition view is controlled entirely by the bean developer. The view is not affected by the container in which the bean runs or the server where the bean is deployed.

For several reasons, enterprise beans simplify the development of large, distributed applications.

- Container-provided services—Because the EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container—not the bean developer—is responsible for system-level services such as transaction management and security authorization.
- Remote clients—Because the enterprise beans, not the clients, contain the application's business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.
- Bean reusability—Because enterprise beans are portable components, the application assembler can build new applications from existing beans. These applications can run on any compliant J2EE server.

Types of Beans

There are three distinct types of enterprise beans:

- **Session bean**, stateful or stateless
    - A *stateful* session bean is intended to represent objects and processes that maintain state across invocations, such as a document copy for editing, or specialized business objects for individual clients.

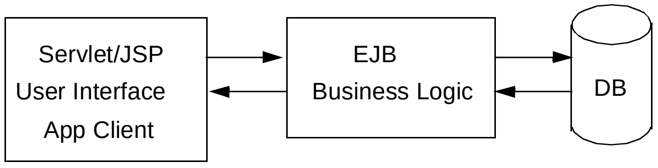
- A *stateless session bean* encapsulates a transient or temporary piece of business logic needed by a specific client that does not maintain state across invocations.
- Refer to “Using Session Beans,” on page 41,” for information on developing session beans.

- **Entity bean**—An *entity bean* commonly represents persistent data which is maintained directly in a database or accessed through an Enterprise Information System (EIS) application as an object.
  - *Bean-managed persistence*—The bean is responsible for its own persistence. The entity bean code that you write contains the calls that access the database. For information on developing entity beans in general and bean-managed persistence in particular, refer to “Using Entity Beans,” on page 55.
  - *Container-managed persistence*—The enterprise bean container handles all database access required by the entity bean by interacting through the persistence manager. For information on container-managed persistence, refer to “Using Container-Managed Persistence for Entity Beans,” on page 79.
- **Message-driven bean**—A *message-driven bean* represents a stateless service; it is essentially an asynchronous message consumer, invoked by JMS, that is completely anonymous and has no client-visible identity.

  Refer to “Using Message-Driven Beans,” on page 129, for information on developing message-driven beans.

EJB Flow

When a user invokes a Sun ONE Application Server servlet from a browser, the servlet may invoke one or more enterprise beans. For example, the servlet may load a JavaServer Page (JSP) to the user’s browser to request a user name and password, then pass the user input to a session bean to validate the input.



After a valid user name and password combination is accepted, the servlet might instantiate one or more entity and session beans to run the application's business logic, then terminate. The beans themselves might instantiate other entity or session beans to do further business logic and data processing.

Sample Scenario

A servlet invokes a session bean that gives a customer service representative access to an order database. This access might include the ability to:

- Browse the database
- Queue items for purchase
- Place customer orders
- Permanently reduce number of parts in the database
- Bill the customer
- Reorder parts when the stock is low or depleted.

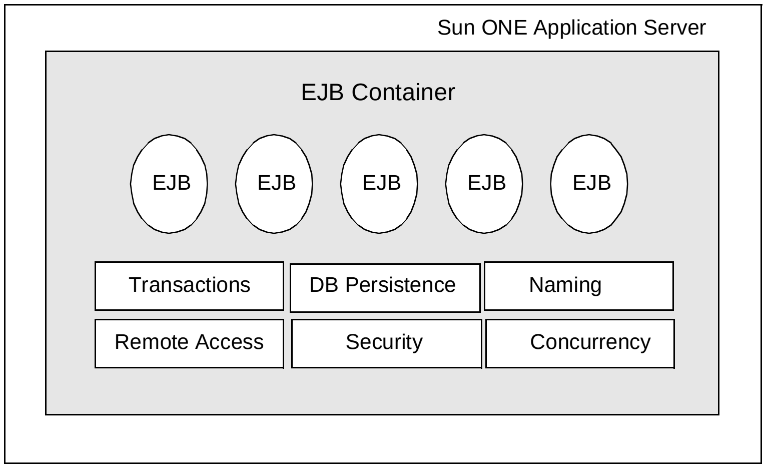
As part of the customer order process, a servlet creates a session bean that manages a shopping cart to keep temporary track of items as a customer selects them. When the order completes, the shopping cart data transfers to the order database and the shopping cart session bean is freed.

The EJB Container

Enterprise beans always work within the context of a container. The container serves as a link between the enterprise beans and the hosting server. The EJB container enables distributed application building using your own components and components from other suppliers.

Through the container, the Sun ONE Application Server provides high-level transaction management, security management, state management (persistence), multithreading, and resource pooling wrappers, thereby shielding you from having to know the low-level API details. By handling concurrency, the container shields you from worry about entities (hence, threads) simultaneously accessing an enterprise bean. This container provides all standard container services denoted by the Enterprise JavaBeans Specification, v2.0, and also provides additional services specific to the Sun ONE Application Server.

The Sun ONE Application Server services include remote access, naming service, security service, concurrency, transaction control, and database access. The following figure illustrates the EJB container provided by the Sun ONE Application Server.



Interfaces

A client never accesses session or entity EJB instances directly. Instead, a client uses the bean's remote interface to access a bean instance. The EJB object class that implements a bean's remote interface is provided by the container.

Home Interface

The *home interface* provides a mechanism for clients to create and destroy and find EJBs. The EJB supplies a home interface for the container that extends the `javax.ejb.EJBHome` interface defined in the EJB specification. At its most basic, the home interface defines zero or more `create` methods for each way to create a bean.

Entity beans must define finder methods for each way that can be used to look up a bean or a collection of beans.

Remote Interface

A *remote interface* (and remote home interface) provides a mechanism for remote clients to access session or entity beans. A remote client can be another EJB deployed in the same or a different container, or a Java program, such as an application, applet, or servlet. The remote client view of an EJB is location independent and can be mapped to non-Java client environments.

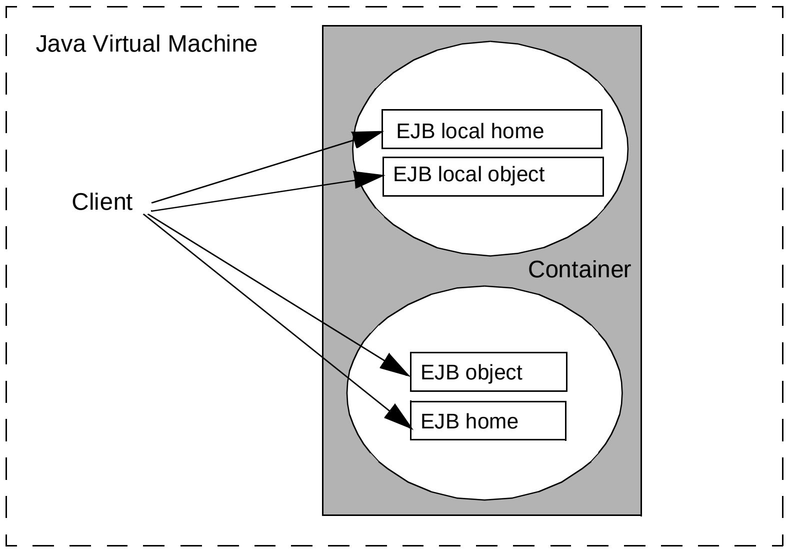
The remote home interface is defined by the EJB developer and implemented by the EJB container.

Local Interface

A *local interface* (and local home interface) provides a mechanism for a client that is located in the same Java Virtual Machine (JVM) with the session or entity bean to access that bean.This provides the *local client view*. A local client may be tightly coupled to the associated bean; session and entity beans can have many local clients.

The container provides the class that implements the local home interface and local interface. The objects that implement these interfaces are local Java objects. The local client view of an EJB is *not* location independent.

The following diagram shows a local client connecting through the local interfaces within the two enterprise beans in the container.



The local interface may be defined for a bean during development, to allow streamlined calls to the bean if a caller is in the same container.

Pooling and Caching

The EJB container of the Sun ONE Application Server pools anonymous instances (message-driven beans, stateless session beans, and entity beans) to reduce the overhead of creating and destroying objects. The EJB container maintains the free pool for each bean that is deployed. Bean instances in the free pool have no identity (that is, no primary key associated) and are used to serve the method calls of the home interface. The free beans are also used to serve all methods for stateless session beans.

Bean instances in the free pool transition from a Pooled state to a Cached state after `ejbCreate` and the business methods run. The size and behavior of each pool can be controlled using the pool-related properties in the `server.xml` and `sun-ejb-jar.xml` files.

The EJB container caches "stateful" instances (stateful session beans and entity beans) in memory to improve performance. The EJB container maintains a cache for each bean that is deployed.

To achieve scalability, the container will selectively evicts some bean instances from the cache, usually when cache overflows. These evicted bean instances return to the free bean pool. The size and behavior of each cache can be controlled using the cache-related properties in the `server.xml` and `sun-ejb-jar.xml` files.

Pooling and caching parameters for the `sun-ejb-jar.xml` file are discussed in "Pooling and Caching Elements," on page 203.

Pooling Parameters

One of the most important parameters of Sun ONE Application Server pooling is `steady-pool-size`. When `steady-pool-size` is set to greater than 0, the container not only pre-populates the bean pool with the specified number of beans, but also attempts to ensure that there is always this many beans in the free pool. This ensures that there are enough beans in the ready to serve state to process user requests.

Another parameter, `pool-idle-timeout-in-seconds`, allows the administrator to specify, through the amount of time a bean instance can be idle in the pool. When `pool-idle-timeout-in-seconds` is set to greater than 0, the container removes/destroys any bean instance that is idle for this specified duration.

Caching Parameters

Sun ONE Application Server provides a way that completely avoids caching of entity beans, using commit-c option. Commit-c option is particularly useful if beans are accessed in large number but very rarely reused. For additional information, refer to "Commit Options," on page 147.

The Sun ONE Application Server caches can be either bounded or unbounded. *Bounded caches* have limits on the number of beans that they can hold beyond which beans are passivated. For stateful session beans, there are three ways (LRU, NRU and FIFO) of picking victim beans when cache overflow occurs. Caches can also be configured to passivate beans that were idle (not accessed for a specified duration) to be passivated.

How Enterprise Beans Access Resources

Enterprise beans can access a wide variety of resources, including databases, JavaMail sessions, JMS objects, and URLs. The J2EE platform provides mechanisms that allow you to access all of these resources in a similar manner.

This section discusses the following:

- JNDI Connection
- Database Connection
- URL Connections

JNDI Connection

J2EE components locate the objects they need to access by invoking the `lookup` method of the Java Naming and Directory Interface (JNDI) API. The value returned by this call represents the object that the caller wants to access. In the case of an enterprise beans, the `lookup` call returns an object reference to the home interface of the bean. This reference may be used for all future invocations on the EJB home interface.

```
Context initial = new InitialContext();
      Object objref
initial.lookup("java:comp/env/ejb/CompString");
```

A J2EE component on the server (a JSP, servlet, or enterprise bean) that wants to access a deployed enterprise bean, uses an EJB reference element in its deployment descriptor to specify this access. The EJB reference is mapped at deployment time to the JNDI name corresponding to the enterprise bean that the component wishes to access. This mapping serves to decouple components accessing enterprise beans from the JNDI names of the beans being accessed. Thus, the JNDI name to which an EJB's home is bound may be changed at deployment time without requiring the caller's code to change.

Database Connection

The persistence type of an enterprise bean determines whether or not you will code the connection routine for accessing a database.

- For beans that access a database and do not use container-managed persistence—You are responsible for writing persistence code. Such beans include entity beans that use bean-managed persistence and session beans.
- For beans that use container-managed persistence—Connection routines are generated for you at deployment. Applies only to entity beans.

URL Connections

A Uniform Resource Locator (URL) specifies the location of a resource on the web, such as web pages. These URLs then can be mapped to JNDI names so that developers can lookup the URLs.

Transaction Management

By dividing the application’s work into units called transactions, you are freed from dealing with the complex issues of database failure recovery and maintaining database integrity.

As a developer, you can choose between using programmatic transaction demarcation in the EJB code (bean-managed) or declarative demarcation (container-managed). Regardless of whether an enterprise bean uses bean-managed or container-managed transaction demarcation, the burden of implementing transaction management is on the EJB container and theSun ONE Application Server. The container and the server implement the necessary low-level transaction protocols, such as the two-phase commit protocol, between a transaction manager and a database system or Sun ONE Message Queue provider.

For information on transaction handling, refer to “Handling Transactions with Enterprise Beans,” on page 143.

How Application Security Works

The J2EE application programming model insulates developers from mechanism-specific implementation details of application security. For the most part, the container provides the implementation of the security infrastructure. J2EE provides this insulation in a way that enhances the portability of applications, allowing them to be deployed in diverse security environments with no additional coding.

The declarative security mechanisms used in an application are expressed in the deployment descriptor. The deployer then uses specific Sun ONE Application Server tools to map the application requirements that are in a deployment descriptor to the security mechanisms that are implemented by the container.

Refer to “Developing Secure Enterprise Beans,” on page 161 for further information. For information on security realms, refer to the *Sun ONE Application Server Developer’s Guide*.

About Developing an Effective Application

Partitioning a Sun ONE Application Server application’s business logic and data processing into the most effective set of servlets, JSPs, session beans, entity beans, and message-driven beans is the crux of your job as a developer. There are no specific rules for object-oriented design with enterprise beans, other than that entity bean instances tend to be long lived, persistent, and shared among clients, while session bean instances tend to be short lived and used only by a single client; message-driven beans are in their own category as the only asynchronous receivers of JMS messages.

In general, your goal is to create a Sun ONE Application Server application that effectively balances the need for execution speed with the need for sharing enterprise beans (among applications and clients) and easily deploying applications across servers.

High-level information and guidelines which can help you develop enterprise beans in the Sun ONE Application Server environment are addressed in the following sections:

- General Process for Creating Enterprise Beans
- Bean Usage Guidelines
- Client View Guidelines
- Remote or Local Interface Guidelines

- Accessing Sun ONE Application Server Functionality

General Process for Creating Enterprise Beans

The procedure in this section outlines the general process of creating an enterprise bean. Specific instructions on creating the various types of enterprise beans are contained in the sections referenced in the following steps.

To create an enterprise bean:

1. Create a directory for all the enterprise bean's files.
2. Decide on the type of enterprise bean you are creating:
   - Session bean (Refer to "Developing Session Beans," on page 44.)
     - Stateful
     - Stateless
   - Entity bean (Refer to "Developing Entity Beans," on page 59.)
     - With bean-managed persistence
     - With container-managed persistence (Refer to "Using Container-Managed Persistence," on page 86.)
   - Message-driven bean (Refer to "Developing Message-Driven Beans," on page 132.)
3. Write the code for the enterprise bean according to the EJB specification, including:
   - A local and / or remote home interface
   - A local and / or remote interface
   - An implementation class (for a message-driven bean, this is all you need)
4. Compile the interfaces and classes.
5. Create the `META-INF` directory and the other structural requirements of an enterprise bean.

6. Create the deployment descriptor files, `ejb-jar.xml` and `sun-ejb-jar.xml`. (Refer to “Assembling and Deploying Enterprise Beans,” on page 169.)

   If the bean is an entity bean with container-managed persistence, you must also create a `sun-cmp-mappings.xml` file and a `.dbschema` file. (Refer to “Using Container-Managed Persistence,” on page 86.)

7. Package the class and the XML files to a JAR file, if desired. If you are using directory deployment, this is optional.

8. Deploy the bean by itself or include it in a J2EE application. (Refer to the Sun ONE Application Server *Developer’s Guide*.)

It’s a good idea to verify the structure of these files using the verifier tool as described in the *Sun ONE Application Server Developer’s Guide*.

Bean Usage Guidelines

Deciding which parts of an application are candidates for entity beans and which are candidates for session beans (stateful or stateless) or message-driven beans will have a significant impact on the effectiveness of your application. In general:

- Use a stateful bean to store non-shared data that corresponds to the user conversational state, that is, a state specific to a single user.
- Use a stateless session bean to access data or perform transactional operations.
- Create session beans that are small, generic, and narrowly task focused. Ideally, these enterprise beans encapsulate behavior that is used in many applications.
- Ask the application assembler to co-locate enterprise beans with your presentation logic (servlets and JSPs) on the same server. This reduces the number of Remote Procedure Calls (RPCs) when the application runs.
- The applications should explicitly remove the beans using the `ejbRemove` method when they are no longer required, thereby reducing the overhead on the container (by eliminating the passivation process).
- Unique naming is optional across enterprise beans in different applications, although applications do need to be named uniquely within the context of a single application server instance. That is, enterprise beans within an application cannot have the same name.

For further information on EJB developer guidelines, refer to “Using Session Beans,” on page 41, “Using Entity Beans,” on page 55, and “Using Message-Driven Beans,” on page 129.

Client View Guidelines

The choice between the use of local and remote interfaces is a design decision that you, the developer, make when developing an enterprise bean. The following facts should be taken into account in determining whether the local or remote programming model should be used:

- The remote programming model provides location independence and flexibility with regard to deployment. The client and enterprise bean are loosely coupled.
- Remote calls involve pass-by-value, providing a layer of isolation between caller and callee. This protects against inadvertent modification of data.
- For local objects, pass-by-reference is optional and is not mandated by the J2EE specification
- Remote calls are potentially expensive.
- Remote calls require that objects that are passed as parameters be serializable.
- Narrowing remote types requires the use of `javax.rmi.PortableRemote.Object.narrow` rather than Java language casts.
- Remote calls involve error cases that are not expected in local calls. The client has to explicitly program handlers for these remote exceptions.
- Because of the overhead of remote programming, it is typically used for relatively coarse-grained component access.
- Local calls can optionally involve pass-by-reference. The client and the bean may be programmed to rely on pass-by-reference semantics. Locals calls imply that the local client and the enterprise bean must be co-located.
- Because local programming provides lighter-weight access to a component, it better supports more fine-grained component access.
- Be aware of the potential sharing of objects passed through the local interface.

For additional information, refer to the Enterprise JavaBeans Specification, v2.0.

Remote or Local Interface Guidelines

With all object-oriented development, you must determine the granularity level needed for your business logic and data processing. *Granularity level* refers to how many pieces to divide an application into.

- A low level of granularity (a low number of beans and bean method invocations)—A more monolithic application is developed, creating an application that is not as likely to promote sharing and reuse, but usually executes more quickly.
- A high level of granularity (a high number of beans and bean method invocations)—An application is divided into many, smaller, more narrowly defined enterprise beans. This creates an application that may promote greater sharing and reuse of enterprise beans among different applications at your site.
- Dividing a distributed application into a moderate to large number of separate beans degrade performance degradation and more overhead. Enterprise beans are not simply Java objects; they are higher-level entities with remote call interface semantics, security semantics, transaction semantics, and properties. This complexity creates overhead.

Accessing Sun ONE Application Server Functionality

You can develop entity beans that adhere strictly to the Enterprise JavaBeans Specification, v2.0, or you can develop entity beans that take advantage of both the specification and additional, value-added Sun ONE Application Server features.

The Sun ONE Application Server offers several features available only in the Sun ONE Application Server container. The Sun ONE Application Server APIs enable applications to take programmatic advantage of specific Sun ONE Application Server environment features.

NOTE Use these APIs only if you plan on using those beans exclusively in a Sun ONE Application Server environment.

About EJB Assembly and Deployment

The process of assembling modules and applications in Sun ONE Application Server conforms to all the customary J2EE-defined specifications, however, you can include the Sun ONE Application Server-specific deployment descriptors that enhance the functionality of the Sun ONE Application Server beyond the J2EE specifications.

A J2EE module is a collection of one or more J2EE components with two deployment descriptors of that type. One descriptor is J2EE standard, the other is specific to Sun ONE Application Server. For enterprise beans, the following deployment descriptor files apply:

- `ejb-jar.xml`—J2EE standard file
- `sun-ejb-jar.xml`—Sun ONE Application Server-specific file
- `sun-cmp-mappings.xml`—Sun ONE Application Server-specific file used for container-managed persistence mapping

Information on the EJB DTDs and XML files is contained in “Assembling and Deploying Enterprise Beans,” on page 169.

An alphabetical list of all EJB-related elements is contained in “Elements Listings,” on page 229.

General information on assembly and deployment is contained in the *Sun ONE Application Server Developer’s Guide*.

Deployment procedures are contained in the Sun ONE Application Server *Administrator’s Guide* and the Administration interface online help.

Chapter 2

Using Session Beans

This section provides guidelines for creating session beans in the Sun ONE Application Server 7 environment.

NOTE If you are unfamiliar with session beans or the EJB technology, refer to the Java Software tutorials:

```
http://java.sun.com/j2ee/docs.html
```

Extensive information on session beans is contained in chapters 6, 7, and 8 of the Enterprise JavaBeans Specification, v2.0.

Overview material on the Sun ONE Application Server is contained in "Introducing the Sun ONE Application Server Enterprise JavaBeans Technology," on page 19 and the *Sun ONE Application Server Product Introduction*.

This section addresses the following topics:

- About Session Beans
- Developing Session Beans
- Restrictions and Optimizations

Extensive information on session beans is contained in the chapters 6, 7, and 8 of the Enterprise JavaBeans Specification, v2.0.

About Session Beans

This section provides an overview of what you need to be aware of about session beans in order to develop effective models for your business processes.

This section addresses the following topics:

- Session Bean Characteristics
- The Container

Session Bean Characteristics

The defining characteristics of a session bean have to do with its non-persistent, independent status within an application. One way to think of a session bean is as a temporary, logical extension of a client application that runs on the Sun ONE Application Server. Generally, a session bean does not represent shared data in a database, but obtains a data snapshot. However, a session bean can update data.

Session beans have the following characteristics:

- Execute for a single client.
- Can be transaction aware.
- Do not represent directly shared data in an underlying database, although they may access and update this data.
- Are short lived.
- Are not persisted in a database.
- Are removed if the container crashes; the client has to establish a new session.

Much of a standard, distributed application consists of logical code units that perform repetitive, time-bound, and user-dependent tasks. These tasks can be simple or complex, and are often needed in different applications. For example, banking applications must verify a user's account ID and balances before performing any transaction. Such discrete tasks, transient by nature, are candidates for session beans.

Sample Scenario

The shopping cart employed by many web-based, online shopping applications is a typical use for a session bean. It is created by the online shopping application only when an item is selected by the user. When selection is completed, the item prices in the cart are calculated, the order is placed, and the shopping cart object is released, or freed. A user can continue browsing merchandise in the online catalog, and if the user decides to place another order, a new shopping cart is created.

Often, a session bean has no dependencies on or connections to other application objects. For example, a shopping cart bean might have a data list member for storing item information, a data member for storing the total cost of items currently in the cart, and methods for adding, subtracting, reporting, and totaling items. On the other hand, the shopping cart might not have a live connection to the database at all.

The Container

Like an entity bean, a session bean can access a database through JDBC calls. A session bean can also provide transaction settings. These transaction settings and JDBC calls are referenced by the session bean's container, allowing it to participate in transaction managed by the container.

A container managing stateless session beans has a different charter from a container managing stateful session beans.

Stateless Container

The *stateless container* manages the stateless session beans, which, by definition, do not carry client-specific states. Therefore, all session beans (of a particular type) are considered equal.

A stateless session bean container uses a bean pool to service requests. The Sun ONE Application Server-specific XML file contains the properties that define the pool:

- `steady-pool-size`
- `resize-quantity`
- `max-pool-size`
- `pool-idle-timeout-in-seconds`

These properties are defined for the deployment descriptor in "Elements in the sun-ejb-jar.xml File," on page 176."

Stateful Container

The *stateful container* manages the stateful session beans, which, by definition, carry the client-specific state. There is a one-to-one relationship between the client and the stateful session beans. At creation, each stateful session bean is given a unique session ID that is used to access the session bean so that an instance of a stateful session bean is accessed by a single client only.

Stateful session beans are managed using cache. The size and behavior of stateful session beans cache can be controlled by specifying the following parameters:

- `max-cache-size`
- `resize-quantity`
- `cache-idle-timeout-in-seconds`
- `removal-timeout-in-seconds`
- `victim-selection-policy`

The `max-cache-size` element specifies the maximum number of session beans that are held in cache. If the cache overflows (when the number of beans exceeds `max-cache-size`), the container then passivates some beans or writes out the serialized state of the bean into a file. The directory in which the file is created is obtained from the `server.xml` file using the configuration APIs.

These properties are defined in the deployment descriptor. See “Elements in the sun-ejb-jar.xml File,” on page 176 for more information.

The passivated beans are stored on the file system.The `session-store` attribute in the `server` element in the `server.xml` file allows the administrator to specify the directory where passivated beans are stored. By default, passivated stateful session beans are stored in application-specific subdirectories created under *instance_dir*/`session-store`.

Developing Session Beans

When a client is done with the session bean, it is released, or freed. When designing an application, you should designate each temporary, single client object as a potential session bean.

The following sections discuss how to develop effective session beans:

- Development Requirements
- Determining Session Bean Usage

- Providing Interfaces
- Creating the Bean Class Definition

Development Requirements

When developing a session bean, you must provide the following:

- Session bean's remote interface and remote home interface, if the session bean provides a remote client view
- Session bean's local interface and local home interface, if the session bean provides a local client view
- Bean class implementation
- Assembly and deployment data

Requirements of a session bean implementation class:

- Implements the `javax.ejb` SessionBean interface.
- The class is defined as public, and cannot be defined as abstract or final.
- Implements one `ejbCreate` method that takes no arguments.
- Implements the business methods.
- Contains a public constructor with no parameters.
- Must not define the `finalize` method.

Determining Session Bean Usage

This section provides some guidelines for determining whether to implement stateful or stateless session beans.

- Stateful Session Bean Considerations
- Stateless Session Bean Considerations

Stateful Session Bean Considerations

Stateful session beans are appropriate if any of the following conditions are true:

- The bean's state represents the interaction between the bean and a specific client.

- The bean needs to hold information about, or on behalf of, the client user conversational state across method invocations.
- The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
- Behind the scenes, the bean manages the work flow of several enterprise beans.

Because stateful session beans are private to a client, their demand on server resources increases as the number of users accessing an application increases. The beans remain in the container until they are explicitly removed by the client, or are removed by the container when they timeout.

The container needs to passivate stateful session beans to secondary storage as its cache fills up and the beans in the cache timeout. If the client subsequently accesses the bean, the container is responsible for activating the bean. This passivation/activation process imposes a performance overhead on the server.

Stateless Session Bean Considerations

You might choose a stateless session bean if any of these conditions exist:

- The bean's state has no data for a specific client, that is, user conversational state does not have to be retained across method invocations on the bean.
- In a single method invocation, the bean performs a generic task for all clients.
- The bean fetches a set of read-only data (from a database) that is often used by clients. Such a bean, for example, could retrieve the table rows that represent the products that are on sale this month.

Use a stateless session bean to access data or perform transactional operations. Stateless session beans provide high scalability because a small number of such beans managed by the container in a stateless bean pool) can help serve a large number of clients. This is possible because stateless beans have no association with the clients. When a request for a service provided by a stateless session bean is received, the container is free to dispatch the request to any bean instance in the pool.

- The `create` method of the remote home interface must return the session bean’s remote interface.
- The `create` method of the local interface must return the session bean’s local interface.
- There can be no other `create` methods in the home interface.
- A stateless session bean must not implement the `javax.ejb.SessionSynchronization` interface.

Providing Interfaces

As the developer, you are responsible for providing interfaces for the bean. If you implement a remote view for your bean, provide a remote component interface and a remote home interface. If you implement a local view, provide a local component interface and a local home interface.

To use interfaces safely, you need to carefully consider potential deployment scenarios, then decide which interfaces can be local and which remote, and finally, develop the application code with these choices in mind.

The following sections discuss creating interfaces:

- Creating a Remote Interface
- Creating a Local Interface
- Creating the Local Home Interface
- Creating the Remote Home Interface

Creating a Remote Interface

A session bean's remote interface defines a user's access to a bean's methods. All remote interfaces extend `javax.ejb.EJBObject`. For example:

```
import javax.ejb.*;
import java.rmi.*;
public interface MySession extends EJBObject {
// define business method methods here....
public String getACcountname() throws RemoteException;
}
```

The remote interface defines the session bean's business methods that a client calls. For each method you define in the remote interface, you must supply a corresponding method in the bean class itself. The corresponding method in the bean class must have the same signature, the same parameter types and return type. The name of the method has ejb preprended to it. For example, the implementation class for `MySession` includes the method:

```
String ejbgetAccountname() throws RemoteException
{
method implementation
}
```

Creating a Local Interface

The local interface may be defined for a bean during development to allow streamlined calls to the bean if a caller is in the same container, that is, running in the same address space or Java Virtual Machine (JVM). This improves the performance of applications in which co-location is planned.

However, the calling semantics of local interfaces are different from those of remote interfaces. For example, remote interfaces pass parameters using pass-by-value semantics, while local interfaces use pass-by-reference. As a developer, you must be aware of the potential sharing of objects passed through the local interface. In particular, be careful that the state of one enterprise bean is not assigned to the state of another. You must also exercise caution in determining which objects to pass across the local interface, particularly in the case where there is a change in transaction or security content.

The local interface extends the `javax.ejb.EJBLocalObject` interface, and is allowed to have super interfaces. The `throws` clause of a method defined in the local interface must not include `java.rmi.RemoteException`. For example:

```
import javax.ejb.*;
   public interface MyLocalSession extends EJBLocalObject {
// define business method methods here....
}
```

For each method defined in the local interface, there must be a matching method in the session bean's class. The matching method must have the same name, the same number and types of arguments, and the same return type. All exceptions defined in the `throws` clause of the matching method of the session bean class must be defined in the `throws` clause of the method of the local interface. The methods should not throw `java.rmi.RemoteException`.

Creating the Local Home Interface

The home interface defines the methods that enable a client using the application to create and remove session beans. An enterprise bean's local home interface defines the methods that allow local clients to create, find, and remove EJB objects, as well as home business methods that are not specific to a bean instance (session beans do not have finders and home business methods). The local home interface is defined by you and implemented by the container. A client locates a session bean's home interface using JNDI.

The local home interface allows a local client to:

- Create a new session object
- Remove a session object

A local home interface always extends `javax.ejb.EJBLocalHome`. For example:

```
import javax.ejb.*;
import java.rmi.*;

public interface MySessionLocalBeanHome extends EJBLocalHome {
   MySessionLocalBean create() throws CreateException;
}
```

Create Methods

As this example illustrates, a session bean's home interface defines one or more `create` methods. Each method must be named `create`, and must correspond in number and argument types to an `ejbCreate` method defined in the session bean class. The return type for each `create` method, however, does not match its corresponding `ejbCreate` method's return type. Instead, it must return the session bean's local interface type.

All exceptions defined in the `throws` clause of an `ejbCreate` method must be defined in the `throws` clause of the matching `create` method in the remote interface. In addition, the `throws` clause in the home interface must always include `javax.ejb.CreateException`.

Remove Methods

A remote client may remove a session object using the `remove` method on the `javax.ejb.EJBObject` interface, or the `remove(Handle handle)` method of the `javax.ejb.EJBHome` interface.

Because session objects do not have primary keys that are accessible to clients, invoking the `javax.ejb.EBJHome.remove(Object primaryKey)` method on a session results in `javax.ejbRemoveException`.

Creating the Remote Home Interface

The container provides the implementation of the remote home interface for each session bean that defines a remote home interface that is deployed in the container. The object that implements this is called a session `EJBHome` object. The remote home interface allows a client to do the following:

- Create a new session object
- Remove a session object
- Get the `javax.ejb.EJBMetaData` interface for the session bean
- Obtain a handle for the remote home interface

The remote home interface must extend the `javax.ejb.EJBHome` interface, and is allowed to have super interfaces. The methods defined in the interface must follow the rules for RMI/IIOP.

The remote home interface must define one or more `create<METHOD>(...)` methods.

A remote home interface always extends `javax.ejb.EJBHome`. For example:

```
import javax.ejb.*;
import java.rmi.*;

public interface MySessionHome extends EJBHome {
   MySession create() throws CreateException, RemoteException;
}
```

As this example illustrates, a session bean's home interface defines one or more `create` methods. The return type for each `create` method, however, does not match its corresponding `ejbCreate` method's return type. Instead, it must return the session bean's remote interface type.

All exceptions defined in the `throws` clause of an `ejbCreate` method must be defined in the `throws` clause of the matching `create` method in the remote interface. In addition, the throws clause in the home interface must always include `javax.ejb.CreateException` and `java.rmi.RemoteException`.

NOTE For stateless session beans, the home interface must have exactly one `create` method and the bean must have exactly one `ejbCreate` method. Both methods take no arguments.

Creating the Bean Class Definition

For a session bean, the bean class must be defined as `public`, must not be `final`, and cannot be `abstract`. The bean class must implement the `javax.ejb.SessionBean` interface.

```
import java.rmi.*;
import java.util.*;
import javax.ejb.*;
public class MySessionBean implements SessionBean {
   // Session Bean implementation. These methods must always
included.
   public void ejbActivate() {
   }
   public void ejbPassivate() {
}
public void ejbRemove() {
}
public void setSessionContext(SessionContext ctx) {
}

// other code omitted here....
}
```

The session bean must implement one or more `ejbCreate(...)` methods. There must be one method for each way a client invokes the bean. For example:

```
public void ejbCreate() {
   string[] userinfo = {"User Name", "Encrypted Password"} ;
}
```

Each `ejbCreate(...)` method must be declared as `public`, return `void`, and be named `ejbCreate`. Arguments must be legal Java RMI types. The `throws` clause may define application specific exceptions and `java.ejb.CreateException`.

Session beans also implement one or more business methods. These methods are usually unique to each bean and represent its particular functionality. For example, if a session bean manages user logins, it might include a unique function called `validateLogin`.

Business method names can be anything, but must not conflict with the method names defined in the EJB interfaces. Business methods must be declared as `public`. Method arguments and return value types must be legal for Java RMI. The `throws` clause may define application specific exceptions.

Session Synchronization

There is one interface implementation permitted in a stateful session bean class definition, particularly `javax.ejb.SessionSynchronization`, that enables a session bean instance to be notified of transaction boundaries and synchronize its state with those transactions.

The `javax.ejb.SessionSynchronization` interface allows a stateful session bean instance to be notified by its container of transaction boundaries. A session bean class is optional to implement this interface. A session bean class should implement this interface only if you want to synchronize its state with the transactions. For example, a stateful session bean that implements this interface will get callbacks after a new transaction begins, but before a transaction commits, and after commitment.

For more information about this interface, see the Enterprise JavaBeans Specification, v2.0.

NOTE The container will only invoke the session synchronization interface methods for stateful session beans that use container-managed transactions.

Abstract Methods

Besides the business methods you define in the remote interface, the `EJBObject` interface defines several abstract methods that enable you to:

- Retrieve the bean's home interface
- Retrieve the bean's handle (a unique identifier)
- Compare the bean to another bean to see if it is identical
- Free or remove the bean when it is no longer needed.

For more information about these built-in methods and how they can be used, see the Enterprise JavaBeans Specification, v2.0.

The deployment tools provided by the container are responsible for the generation of additional classes when the session bean is deployed.

Restrictions and Optimizations

This section discusses restrictions on developing session beans and provides some optimization guidelines:

- Optimizing Session Bean Performance
- Restricting Transactions

Optimizing Session Bean Performance

For stateful session beans, co-locating the stateful beans with their clients so that the client and bean are executing in the same process address space will improve performance.

Restricting Transactions

The following restrictions on transactions are enforced by the container and must be observed as you develop session beans:

- A session bean can participate in, at most, a single transaction at a time.
- If a session bean is participating in a transaction, a client cannot invoke a method on the bean such that the transaction attribute in the deployment descriptor would cause the container to execute the method in a different or unspecified transaction context or an exception is thrown.
- If a session bean instance is participating in a transaction, a client cannot invoke the `remove` method on the session object's home or component interface object or an exception is thrown.

Chapter 3

Using Entity Beans

This section describes entity beans and explains the requirements for creating them in the Sun ONE Application Server 7 environment.

NOTE If you are unfamiliar with entity beans or the EJB technology, refer to the Java Software tutorials:

```
http://java.sun.com/j2ee/docs.html
```

Extensive information on entity beans is contained in chapters 9, 10, 12, 13, and 14 of the Enterprise JavaBeans Specification, v2.0.

Overview material on the Sun ONE Application Server is contained in “Introducing the Sun ONE Application Server Enterprise JavaBeans Technology,” on page 19 and the *Sun ONE Application Server Product Introduction*.

This section addresses the following topics:

- About Entity Beans
- Developing Entity Beans
- Using Read-Only Beans
- Handling Synchronization of Concurrent Access

NOTE If you are already familiar with entity beans and are only concerned with container-managed persistence, go to “Using Container-Managed Persistence for Entity Beans,” on page 79.

About Entity Beans

An entity bean implements an object view of an entity stored in an underlying database, or an entity implemented by an existing enterprise application (for example, by a mainframe program or by an ERP application). Some examples of business objects are customers, orders, and products. The data access protocol for transferring the state of the entity between the entity bean instances and the underlying database is referred to as *object persistence.*

The following topics are discussed in this section:

- Entity Bean Characteristics
- The Container
- Persistence
- Read-Only Beans

Entity Bean Characteristics

Entity beans differ from session beans in several ways. Entity beans are persistent, can be accessed simultaneously by multiple clients, have primary keys, and may participate in relationships with other entity beans.

Entity beans have the following characteristics:

- Provide an object view of data in a database.
- Allow shared access by multiple users.
- Persist for as long as needed by all clients, using either bean-managed persistence or container-managed persistence.
- Transparently survive server crashes.
- Represent shared data in a database.

A good situation for using entity beans includes a well encapsulated, transactional, and persistent interaction with databases, documents, and other business objects.

The Container

Entity beans rely on the enterprise bean container to manage security, concurrency, transactions, and other container-specific services for the entity objects it manages. Multiple clients can access an entity object at the same time, while the container transparently handles simultaneous accesses through transactions.

Each entity has a unique object identifier. A customer entity bean, for example, might be identified by a customer number. This unique identifier, or *primary key*, enables the client to locate a particular entity bean.

Like a session bean, an entity bean can access a database through JDBC calls inside methods whose transaction attributes can be set using deployment descriptors.The container supports both bean-managed and container-managed persistence as described in the following section.

Persistence

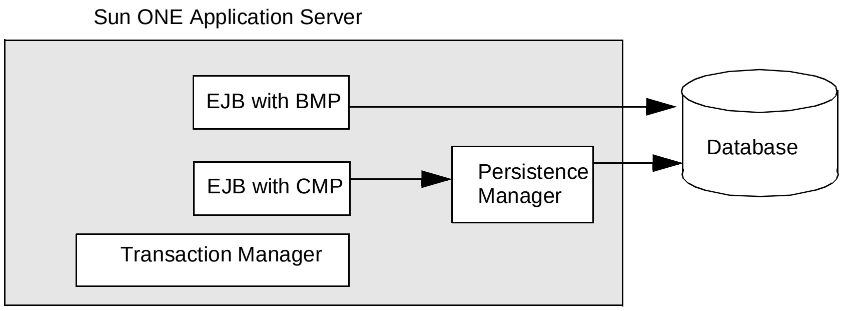
Because the state of an entity bean is saved in a some durable storage, it is persistent. *Persistence* means that the entity bean's state exists beyond the lifetime of the application or the server process.

Persistence of entity beans may done explicitly by the bean and programmed by the bean developer. This is known as *bean-managed persistence* (BMP).

Persistence management can also be delegated to the container, leveraging the Sun ONE Application Server and the persistence management APIs of the enterprise beans. This approach is called container-managed persistence (CMP). In the CMP mechanism, a persistence manager, integrated with the Sun ONE Application Server, is required to ensure reliable persistence. Refer to "Using Container-Managed Persistence for Entity Beans," on page 79 for additional information on container-managed persistence.

The following figure illustrates how persistence works in the Sun ONE Application Server environment.

Entity Bean Flow



Guidelines for selecting the most appropriate persistence method for your applications are contained in "Determining Entity Bean Usage," on page 60.

The following topics are addressed in this section:

- Bean-Managed Persistence
- Container-Managed Persistence

Bean-Managed Persistence

In *bean-managed persistence*, the bean is responsible for its own persistence. The entity bean code that you write contains the calls that access the database.

You code a bean-managed entity bean by providing database access calls—through JDBC and SQL—directly in the bean class methods. Database access calls must be in the `ejbCreate`, `ejbRemove`, `ejbFind`*XXX*, `ejbLoad`, and `ejbStore` methods. The advantage of this approach is that these beans can be deployed to the application server without requiring much effort. The disadvantage is that database access is expensive and, in some cases, the application server can do a better job of optimizing database access than the application programmer can. Also, bean-managed persistence requires the developer to write JDBC code.

For details about using JDBC to work with data, see the Sun ONE Application Server *Developer's Guide to J2EE Features and Services*.

Container-Managed Persistence

In *container-managed persistence*, the enterprise bean container handles all database access required by the entity bean by interacting through the persistence manager. The bean's code contains no database access (JCBC) calls. As a result, the bean's code is not tied to a specific persistent storage mechanism (database). Because of this flexibility, even if you redeploy the same entity bean on a different database, you won't need to modify the bean's code. In short, your entity beans are more portable.

The bean developer provides abstract bean classes. Typically, the container-managed persistence runtime generates concrete implementation classes that know how to load and save the bean state (in the `ejbLoad` and `ejbStore` methods).

To generate the data access calls, the container needs information that you provide in the entity bean's abstract schema. Additional information on the abstract schema is contained in "Abstract Schema," on page 84.

Read-Only Beans

A *read-only bean* is an entity bean that is never modified by an EJB client. The data that a read-only bean represents may be updated externally by other enterprise beans, or by other means, such as direct database updates.

NOTE For this release of the Sun ONE Application Server, only entity beans that use bean-managed persistence can be designated as read-only.

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. Instructions for creating read-only beans are contained in "Using Read-Only Beans," on page 74.

Developing Entity Beans

When creating an entity bean, you must provide a number of class files. The tasks required are discussed in the following topics:

- Determining Entity Bean Usage
- Responsibilities of the Bean Developer

- Defining the Primary Key Class
- Defining Remote Interfaces
- Defining Local Interfaces
- Creating the Bean Class Definition (for Bean-Managed Persistence)

Determining Entity Bean Usage

You should probably use an entity bean when the bean represents a business entity, not a procedure, and / or the bean's state must be persistent (the bean's state still exists in the database if the server is shut down).

Unlike session beans, entity bean instances can be accessed simultaneously by multiple clients. The container is responsible for synchronizing the instance state using transactions. Because this responsibility is delegated to the container, you do not need to consider concurrent access methods from multiple transactions.

Your choice of persistence method also has an impact:

- Bean-managed persistence—When you implement an entity bean to manage its own persistence, you implement persistence code (such as JDBC calls) directly in the EJB class methods. The downside is portability loss (that is, the risk of associating the bean with a specific database).
- Container-managed persistence—When entity bean persistence is managed by the container, the container transparently manages the persistence state. You do not need to implement any data access code in the bean methods. Not only is this method simpler to implement, but it makes the bean portable to different databases. Refer to "Using Container-Managed Persistence for Entity Beans," on page 79" for on implementation guidelines.

Responsibilities of the Bean Developer

This section describes what you need to do to ensure that an entity bean with bean-managed persistence can be deployed on the Sun One Application Server.

The entity bean developer is responsible for providing the following class files:

- Primary key class
- Entity bean remote interface and remote home interface, if the entity bean provides a remote client view

- Entity bean local interface and local home interface, if the entity bean provides a local client view
- Entity bean class

Defining the Primary Key Class

The EJB architecture allows a primary key class to be any class that is a legal Value Type in RMI-IIOP. The class must provide suitable implementation of the `hashCode` and equals (Object other) methods. The primary key class may be specific to an entity bean class, that is, each entity bean class may define a different class for its primary key, but it is possible for multiple entity beans to use the same primary key class.

You must specify a primary key class in the deployment descriptor.

Defining Remote Interfaces

This section discusses the following topics:

- Creating the Remote Home Interface
- Creating a Remote Interface

Creating the Remote Home Interface

As a bean developer, you must provide the bean's remote home interface (if it is applicable). The home interface defines the methods that enable a client accessing an application to create, find, and remove entity objects. You must create a remote home interface that meets the following requirements:

- The interface must extend the `javax.ejb.EJBHome` interface.
- The methods defined in this interface must follow the rules for RMI-IIOP. This means that their argument and return types are of valid types for RMI-IIOP, and that their throws clauses include `java.rmi.RemoteException`.
- Each method defined in the remote home interface must be one of the following:
    - A create method.
    - The remote home interface must always include the `findByPrimaryKey` method, which is always a single-object finder. The method must declare the primary key class as the method argument.

- A finder method.
- A home method. Home methods can have arbitrary names, provided they do not clash with the create, find, and remove method names. The matching `ejbHome` method specified in the entity bean class must have the same number and types of arguments, and must return the same type as the home method specified in the remote home interface of the bean.

Remote Create Methods

- Each create method must be named createXXX, where XXX is a unique method name continuation that matches one of the `ejbCreateXXX` methods defined in the enterprise bean class. For example, `createEmployee(...)`, `createLargeOrder(....)`.
- The matching `ejbCreateXXX` in the bean must have the same number and types of its arguments. However, the return type is different.
- The return type for a createXXX method must be the entity bean remote interface type.
- All the exceptions defined in the throws clause of the matching `ejbCreateXXX` and `ejbPostCreateXXX` methods of the enterprise bean class must be included in the throws clause of the matching create method of the remote home interface (that is, the set of exceptions defined for the create method must be a superset of the union of exceptions defined for the `ejbCreateXXX` and `ejbPostCreateXXX` methods).
- The throws clause of a create method must include `javax.ejb.CreateException`.

Remote Find Methods

- A home interface can define one or more find methods. Each method must be named findXXX, where XXX is a unique method name continuation. For example, `findApplesAndOranges`.
- Each finder method must correspond to one of the finder methods defined in the entity bean class definition.
- The number and argument types must also correspond to the finder method definitions in the bean class.
- The return type for a find `<METHOD>` method must be the entity bean's remote interface type (for a single-object finder), or a collection thereof (for a multi-object finder).

- All the exceptions defined in the throws clause of an `ejbFind` method of the entity bean class must be included in the throws clause of the matching find method of the remote home interface.
- The throws clause of a finder method must include `javax.ejb.FinderException`.

findByPrimaryKey Method

- Every remote home interface must always include the `findByPrimaryKey` method, which is always a single-object finder.
- The method must declare the primary key class as the method argument.
- All the exceptions defined in the throws clause of an `ejbFindByPrimaryKey` method of the entity bean class must be included in the throws clause of the matching find method of the remote home interface.
- The throws clause of a `findByPrimaryKey` method must include `javax.ejb.FinderException`.

Remote Remove Methods

All home interfaces automatically (by extending `javax.ejb.EJBHome`) define two `remove` methods for destroying an enterprise bean when it is no longer needed:

```
public void remove(java.lang.Object primaryKey)
throws java.rmi.RemoteException,   RemoveException
```

```
public void remove(Handle handle)
throws java.rmi.RemoteException,  RemoveException
```

NOTE Do not override these remove methods.

Example of a Remote Home interface

```
import javax.ejb.*;
import java.rmi.*;


public interface MyEntityBeanLocalHome
   extends EJBHome
{
   /**
      * Create an Employee
      * @param empName Employee name
      * @exception CreateException If the employee cannot be
```

```
          created
          * @return The remote interface of the bean
       */
    public MyEntity create(String empName)
       throws CreateException;
    /**
       * Find an Employee
       * @param empName Employee name
       * @exception FinderException if the empName is not found
       * @return The remote interface of the bean
       */
    public MyEntity findByPrimaryKey(String empName)
       throws FinderException;
}
```

Defining Local Interfaces

To build an enterprise bean that allows local access, you must code the local interface and the local home interface. The local interface defines the bean's business methods; the local home interface defines its life cycle (create/remove) and finder methods.

This section addresses the following topics:

- Creating the Local Home Interface
- Creating a Local Interface

Creating the Local Home Interface

The home interface defines the methods that enable a client using the application to create and remove entity beans. A bean's local home interface defines the methods that allow local clients to create, find, and remove EJB objects, as well as home business methods that are not specific to a bean instance (session beans do not have finders and home business methods). The local home interface is defined by you and implemented by the container. A client locates a bean's home using JNDI.

The local home interface allows a local client to:

- Create new entity objects within the home
- find existing entity objects within the home
- Remove an entity object from the home
- Execute a home business method

A local home interface always extends `javax.ejb.EJBLocalHome`. For example:

```
import javax.ejb.*;
public interface MyEntityLocalBeanHome extends EJBLocalHome {
   MyEntityLocalBean create() throws CreateException;
}
```

Creating a Local Interface

If an entity bean is the target of a container-managed relationship, it must have local interfaces. The direction of the relationship determines whether or not a bean is a target. Because they require local access, entity beans that participate in a container-managed relationship must reside in the same EJB JAR file. The primary benefit of this locality is improved performance—local calls are faster than remote calls.

Since local interfaces follow pass by reference semantics, you must be aware of the potential sharing of objects passed through the local interface. In particular, be careful that the state of one enterprise bean is not assigned as the state of another. You must also exercise caution in determining which objects to pass across the local interface, particularly in the case where there is a change in transaction or security content.

- The interface must extend the `javax.ejb.EJBLocalHome` interface.
- The throws clause of a method on the local home interface must not include the `java.rmi.RemoteException`.
- Each method defined in the local home interface must be one of the following:
  - A create method
  - A finder method
  - A home method

(Local) Create Methods

- Each create method must be named createXXX, where XXX is a unique method name continuation, and it must match one of the `ejbCreateXXX` methods defined in the enterprise bean class. For example, `createEmployee(...)`, `createLargeOrder(....)`.
- The matching `ejbCreateXXX` in the bean must have the same number and types of its arguments. (Note that the return type is different.)
- The return type for a createXXX method must be the entity bean's local interface type.

- All the exceptions defined in the throws clause of the matching `ejbCreateXXX` and `ejbPostCreateXXX` methods of the enterprise bean class must be included in the throws clause of the matching create method of the remote home interface (that is, the set of exceptions defined for the create method must be a superset of the union of exceptions defined for the `ejbCreateXXX` and `ejbPostCreateXXX` methods).
- The throws clause of a create method must include `javax.ejb.CreateException`.

(Local) Find Methods

- A home interface can define one or more find methods. Each method must be named findXXX, where XXX is a unique method name continuation. For example, `findApplesAndOranges`.
- Each finder method must correspond to one of the finder methods defined in the entity bean class definition.
- The number and argument types must also correspond to the finder method definitions in the bean class.
- The return type for a find `<METHOD>` method must be the entity bean's local interface type (for a single-object finder), or a collection thereof (for a multi-object finder).
- All the exceptions defined in the throws clause of an `ejbFind` method of the entity bean class must be included in the throws clause of the matching find method of the remote home interface.
- The throws clause of a finder method must include the javax.ejb.FinderException.

findByPrimaryKey Method

- Every local home interface must always include the `findByPrimaryKey` method, which is always a single-object finder.
- The method must declare the primary key class as the method argument.
- All the exceptions defined in the throws clause of an `ejbFindByPrimaryKey` method of the entity bean class must be included in the throws clause of the matching find method of the remote home interface.
- The throws clause of a `findByPrimaryKey` method must include `javax.ejb.FinderException`.

(Local) home Methods

- Home methods can have arbitrary names, provided that they do not clash with create, find, and remove method names.
- The matching `ejbHome` method specified in the entity bean class must have the same number and types of arguments and must return the same type as the home method as specified in the local home interface of the bean.

Creating a Remote Interface

Besides the business methods you define in the remote interface, the EJBObject interface defines several abstract methods that enable you to:

- Retrieve the bean's home interface
- Retrieve the bean's handle-to retrieve the bean's primary key which uniquely identifies the bean's instance
- Compare the bean to another bean to see if it is identical
- Remove the bean when it is no longer needed

For more information about these built-in methods and how they are used, see the Enterprise JavaBeans Specification, v2.0.

NOTE The Enterprise JavaBeans Specification, v2.0 permits the bean class to implement the remote interface's methods, but recommends against this practice to avoid inadvertently passing a direct reference (through this) to a client in violation of the client-container-EJB protocol intended by the Enterprise JavaBeans Specification, v2.0.

- An entity bean's remote interface defines a user's access to a bean's methods.
- The interface must extend the `javax.ejb.EJBObject` interface.
- The methods defined in the remote interface must follow the rules for RMI-IIOP.
- This means that their argument and return value types must be valid types for RMI-IIOP, and their throws clauses must include the java.rmi.RemoteException.
- For each method defined in the remote interface, there must be a matching method in the entity bean s class. The matching method must have The same name. The same number and types of its arguments, and the same return type.

- All the exceptions defined in the throws clause of the matching method of the enterprise bean class must be defined in the throws clause of the method of the remote interface.
- The remote interface methods must not expose local interface types, local home interface types, or the managed collection classes that are used for entity beans with container-managed persistence as arguments or results.

Example of a Remote Interface

The following fragment is an example of a remote interface

```
import javax.ejb.*;
import java.rmi.*;

public interface MyEntity
   extends EJBObject
   {
   public String getAddress() throws RemoteException;
   public void setAddress(String addr) throws RemoteException;
}
```

Creating the Bean Class Definition (for Bean-Managed Persistence)

For an entity bean that uses bean-managed persistence, the bean class must be defined as `public` and cannot be `abstract`. The bean class must implement the `javax.ejb.EntityBean` interface. For example:

```
import java.rmi.*;
import java.util.*;
import javax.ejb.*;
public class MyEntityBean implements EntityBean {
   // Entity Bean implementation. These methods must always be
   included.
   public void ejbActivate() {
   }
   public void ejbLoad() {
   }
   public void ejbPassivate() {
   }
   public void ejbRemove() {
   }
   public void ejbStore() t{
   }
   public void setEntityContext(EntityContext ctx) {
    }
    public void unsetEntityContext() {
    }
    // other code omitted here....
    }
```

In addition to these methods, the entity bean class must also define one or more `ejbCreate` methods and the `ejbFindByPrimaryKey` finder method. Optionally, it may define one `ejbPostCreate` method for each `ejbCreate` method. It may provide additional, developer-defined finder methods that take the form `ejbFind`*XXX*, where *XXX* represents a unique method name continuation (for example, `ejbFindApplesAndOranges`) that does not duplicate any other method names.

Entity beans typically implement one or more business methods. These methods are usually unique to each bean and represent its particular functionality. Business method names can be anything, but must not conflict with the method names used in the EJB architecture. Business methods must be declared as `public`. Method arguments and return value types must be Java RMI legal. The `throws` clause may define application-specific exceptions and may include `java.rmi.RemoteException`.

There are two business method types to implement in an entity bean:

- Internal methods—Used by other business methods in the bean, but never accessed outside the bean itself.
- External methods—referenced by the entity bean's remote interface.

The following sections address the various methods in an entity bean's class definition:

- Using ejbCreate
- Using ejbActivate and ejbPassivate
- Using ejbLoad and ejbStore
- Using setEntityContext and unsetEntityContext
- Using ejbRemove
- Using Finder Methods

Using ejbCreate

The entity bean must implement one or more `ejbCreate` methods. There must be one method for each way a client is allowed to invoke the bean. For example:

```
public String ejbCreate(String orderId, String customerId,
   String status, double totalPrice)
   throws CreateException {
      try {
      InitialContext ic = new InitialContext();
      DataSource ds = (DataSource) ic.lookup(dbName);
      con =  ds.getConnection();
      String insertStatement =
         "insert into orders values ( ? , ? , ? , ? )";
      PreparedStatement prepStmt =
         con.prepareStatement(insertStatement);
      prepStmt.setString(1, orderId);
      prepStmt.setString(2, customerId);
      prepStmt.setDouble(3, totalPrice);
      prepStmt.setString(4, status);
      prepStmt.executeUpdate();
      prepStmt.close();
   } catch (Exception ex) {
      throw new CreateException("ejbCreate: "
         +ex.getMessage());
   }
}

public String ejbPostCreate(String orderId, String customerId,String
status, double totalPrice)
   throws CreateException

{
......
......
}
```

Each `ejbCreate` method must be declared as `public`, return the primary key type, and be named `ejbCreate`. The return type can be any legal Java RMI type that converts to a number for key purposes. All arguments must be legal Java RMI types. The `throws` clause may define application-specific exceptions, and may include `java.ejb.CreateException`.

This is the method in which relationships are established. For each `ejbCreate` method, the entity bean class may define a corresponding `ejbPostCreate` method to handle entity services immediately following creation. Each `ejbPostCreate` method must be declared as `public`, must return void, and be named `ejbPostCreate`. The method arguments, if any, must match in number and argument type its corresponding `ejbCreate` method. The `throws` clause *may* define application-specific exceptions, and may include `java.ejb.CreateException`.

Using ejbActivate and ejbPassivate

When an entity bean instance is needed by a server application, the bean's container invokes `ejbActivate` to ready a bean instance for use. Similarly, when an instance is no longer needed, the bean's container invokes `ejbPassivate` to disassociate the bean from the application.

If specific application tasks need to be performed when a bean is first made ready for an application, or when a bean is no longer needed, you should program those operations within the `ejbActivate` and `ejbPassivate` methods. For example, you may release references to database and backend resources during `ejbPassivate` and regain them during `ejbActivate`.

Using ejbLoad and ejbStore

An entity bean can collaborate with the container to store the bean state information in a database, for synchronization purposes. In the case of bean-managed persistence, you are responsible for coding `ejbLoad` and `ejbStore`. The container ensures that the state of the bean is synchronized with the database by calling `ejbLoad` at the beginning of a transaction and calling `ejbStore` when the transaction completes successfully.

Use your implementation of `ejbStore` to store state information in the database, and use your implementation of `ejbLoad` to retrieve state information from the database.

The following example shows `ejbLoad` and `ejbStore` method definitions that store and retrieve active data.

```
public void ejbLoad()
      throws java.rmi.RemoteException
{
   String itemId;
   javax.sql.Connection dc = null;
   java.sql.Statement stmt = null;
   java.sql.ResultSet rs = null;

   itemId = (String) m_ctx.getPrimaryKey();

   System.out.println("myBean: Loading state for item " + itemId);

   String query =
      "SELECT s.totalSold, s.quantity " +
      " FROM Item s " +
      " WHERE s.item_id = " + itemId;

   dc = new DatabaseConnection();
```

```
        dc.createConnection(DatabaseConnection.GLOBALTX);
        stmt = dc.createStatement();
        rs = stmt.executeQuery(query);

        if (rs != null) {
            rs.next();
            m_totalSold = rs.getInt(1);
            m_quantity = rs.getInt(2);
        }
    }
    public void ejbStore()
            throws java.rmi.RemoteException
    {
        String itemId;
        itemId = (String) m_ctx.getPrimaryKey();
        DatabaseConnection dc = null;
        java.sql.Statement stmt1 = null;
        java.sql.Statement stmt2 = null;

        System.out.println("myBean: Saving state for item = " + itemId);

        String upd1 =
            "UPDATE Item " +
            " SET quantity = " + m_quantity +
            " WHERE item_id = " + itemId;

        String upd2 =
            "UPDATE Item " +
            " SET totalSold = " + m_totalSold +
            " WHERE item_id = " + itemId;

        dc = new DatabaseConnection();
        dc.createConnection(DatabaseConnection.GLOBALTX);
        stmt1 = dc.createStatement();
        stmt1.executeUpdate(upd1);
        stmt1.close();
        stmt2 = dc.createStatement();

        stmt2.executeUpdate(upd2);
        stmt2.close();
    }
```

For more information about bean isolation levels that access transactions concurrently with other beans, see "Handling Synchronization of Concurrent Access," on page 78.

Using setEntityContext and unsetEntityContext

A container calls `setEntityContext` after it creates an entity bean instance to provide the bean with an interface to the container. Implement this method to store the entity context passed by the container. You can later use this reference to get the primary key of the instance, and so on.

```
public void setEntityContext(javax.ejb.EntityContext ctx)
{
m_ctx = ctx;
}
```

Similarly, a container calls `unsetEntityContext` to remove the container reference from the instance. This is the last bean class method a container calls before the bean instance becomes a candidate for removal. After this call, the Java garbage collection mechanism eventually calls `finalize` on the instance to clean it up and dispose of it.

```
public void unsetEntityContext()
{
m_ctx = null;
}
```

Using ejbRemove

The client can invoke the remove methods on the entity bean's home or component interface to remove the associated record from the database. The container invokes the `ejbRemove` method on an entity bean instance in response to a client invocation on the entity bean's home or component interface, or as the result of a cascade-delete operation.

Using Finder Methods

Because entity beans are persistent, shared among clients, and may have more than one instance instantiated at the same time, an entity bean must implement at least one `ejbFindByPrimaryKey` method. This enables the client and the container to locate a specific bean instance. All entity beans must provide a unique primary key as an identifying signature. Implement the `ejbFindByPrimaryKey` method in the bean's class to enable a bean to return its primary key to the container.

The following example shows a definition for `FindByPrimaryKey`:

```
public String ejbFindByPrimaryKey(String key)
        throws java.rmi.RemoteException,
            javax.ejb.FinderException
```

In some cases, you find a specific entity bean instance based on what the enterprise bean does, based on certain values the instance is working with, or based on other criteria. These implementation-specific finder method names take the form `ejbFindXXX`, where `XXX` represents a unique continuation of a method name (for example, `ejbFindApplesAndOranges`) that does not duplicate any other method names.

Finder methods must be declared as `public`, and their arguments, and return values must be legal Java RMI types. Each finder method return type must be the entity bean's primary key type or a collection of objects of the same primary key type. If the return type is a collection, the return type must be one of the following:

- JDK 1.1 `java.util.Enumeration` interface
- Java 2 `java.util.Collection` interface

The `throws` clause of a finder method is an application-specific exception, and may include `java.rmi.RemoteException` and/or `java.ejb.FinderException`.

Using Read-Only Beans

A *read-only bean* is an entity bean that is never modified by an EJB client. The data that a read-only bean represents may be updated externally by other enterprise beans, or by other means, such as direct database updates.

NOTE For this release of Sun ONE Application Server, only entity beans that use bean-managed persistence can be designated as read-only.

Read-only beans are specific to Sun ONE application server and are not part of the Enterprise JavaBeans Specification, v2.0.

The following topics are addressed in this section:

- Read-Only Bean Characteristics and Life Cycle
- Read-Only Bean Good Practices
- Refreshing Read-Only Beans
- Deploying Read Only Beans

Read-Only Bean Characteristics and Life Cycle

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. For example, a read-only bean can be used to represent a stock quote for a particular company, which is updated externally. In such a case, using a regular entity bean may incur the burden of calling `ejbStore`, which can be avoided by using a read-only bean.

Read-only beans have the following characteristics:

- Only entity beans can be read-only beans.
- Only bean-managed persistence is allowed.
- Only container-managed transactions are allowed; read-only beans cannot start their own transactions.
- Read-only beans don't update any bean state.
- `ejbStore` is never called by the container.
- `ejbLoad` will be called only when a transactional method is called or when the bean is initially created (in the cache), or at regular intervals controlled by the bean's `refresh-period-in-seconds`.
- The home interface can have any number of find methods. The return type of the find methods must be the primary key for the same bean type (or a collection of primary keys).
- If the data that the bean represents can change, then `refresh-period-in-seconds` must be set to refresh the beans at regular intervals. `ejbLoad` is called at this regular interval.

A read-only bean comes into existence using the appropriate find methods.

Read-only beans are cached and have the same cache properties as entity beans. When a read-only bean is selected as a victim to make room in the cache, `ejbPassivate` is called and the bean is returned to the free pool. When in the free pool, the bean has no identity and will be used only to serve any finder requests.

Read-only beans are bound to the naming service like regular read-write entity beans, and clients can look up read-only beans the same way read-write entity beans are looked up.

Read-Only Bean Good Practices

- Avoid having any `create` or `remove` methods in the home interface

- Use any of the valid EJB 2.0 transaction attributes for the transaction attribute for methods

  The reason for having `TX_SUPPORTED` is to allow reading uncommitted data in the same transaction. Also, the TX attributes can be used to force `ejbLoad`.

Refreshing Read-Only Beans

There are several ways of refreshing read-only beans as addressed in the following sections:

- Invoking a Transactional Method
- Refreshing Periodically
- Refreshing Programmatically

Invoking a Transactional Method

Invoking any transactional method will invoke `ejbLoad`.

Refreshing Periodically

Read-only beans can be refreshed periodically by specifying the `refresh-period-in-seconds` element in the Sun ONE Application Server-specific XML file.

- If the value specified in `refresh-period-in-seconds` is zero, the bean is never refreshed (unless a transactional method is accessed).
- If the value is greater than zero, the bean is refreshed at the rate specified.

NOTE This is the only way to refresh the bean state if the data can be modified external to the Sun ONE Application Server.

Refreshing Programmatically

Typically, beans that update any data that is cached by read-only beans need to notify the read-only beans to refresh their state. You can use `ReadOnlyBeanNotifier` to force the refresh of read-only beans. To do this, invoke the following methods on the `ReadOnlyBeanNotifier` bean:

```
public interface ReadOnlyBeanNotifier
    extends java.rmi.Remote
{
    refresh(Object PrimaryKey)
        throws RemoteException;
}
```

The implementation of the `ReadOnlyBeanNotifier` interface is provided by the container. The user can look up `ReadOnlyBeanNotifier` using the following fragment of code:

```
com.sun.ejb.ReadOnlyBeanNotifier notifier =
com.sun.ejb.containers.ReadOnlyBeanHelper.getReadOnlyBeanNotifier
    (<ejb-name-of -the-target>);
    notifier.refresh(<PrimaryKey>);
```

Beans that update any data that is cached by read-only beans need to call the `refresh` methods. The next (non-transactional) call to the read-only bean will invoke `ejbLoad`.

Deploying Read Only Beans

Read-only beans are deployed in the same manner as other entity beans. However, in the entry for the bean in the Sun ONE Application Server-specific XML file, the `is-read-only-bean` element must be set to true. That is:

```
    <is-read-only-bean>true</is-read-only-bean>
```

Also, the `refresh-period-in-seconds` element may be set to some value that specifies the rate at which the bean is refreshed. If this element is missing, a default of 600 (seconds) is assumed.

All requests with the same transaction context are routed to the same read-only bean instance. The deployer can specify if such multiple requests have to be serialized by setting the `allow-concurrent-access` element to either true (to allow concurrent accesses) or false (to serialize concurrent access to the same read-only bean). The default is false.

For further information on these elements, refer to the *Sun ONE Application Server Administrator's Configuration File Reference.*

Handling Synchronization of Concurrent Access

As an entity bean developer, you generally do not have to be concerned about concurrent access to an entity bean from multiple transactions. The bean's container automatically provides synchronization in these cases. In the Sun ONE Application Server, the container activates one entity bean instance for each simultaneously occurring transaction that uses the bean.

Transaction synchronization is performed automatically by the underlying database during database access calls. You typically perform this synchronization in conjunction with the underlying database or resource. One approach would be to acquire the corresponding database locks in the `ejbLoad` method, for example by choosing an appropriate isolation level or by using a `select for update` clause. The specifics vary depending on the database being used.

For more information, see the Enterprise JavaBeans Specification, v2.0 as it relates to concurrent access.

Chapter 4

Using Container-Managed Persistence for Entity Beans

This section contains information on how container-managed persistence works in the Sun ONE Application Server 7 environment. Implementation procedures are included.

NOTE To implement container-managed persistence, you should already be familiar with entity beans, which are discussed in "Using Entity Beans," on page 55.

This section addresses the following topics:

- Sun ONE Application Server Support
- About Container-Managed Persistence
- Using Container-Managed Persistence
- Third-Party Pluggable Persistence Manager API
- Restrictions and Optimizations
- Elements in the sun-cmp-mappings.xml File
- Examples

Extensive information on container-managed persistence is contained in chapters 10, 11, and 14 of the Enterprise JavaBeans Specification, v2.0.

Sun ONE Application Server Support

Sun ONE Application Server support for container-managed persistence includes:

- Full support for the J2EE v 1.3 specification’s container-managed persistence model.
    - Support for commit options B and C for transactions as defined in the Enterprise JavaBeans Specification, v2.0. Refer to “Commit Options,” on page 147 for further information.
    - The primary key class must be a subclass of `java.lang.Object`. This ensures portability, and is noted because some vendors allow primitive types (such as `int`) to be listed as the primary key class.
- The Sun ONE Application Server container-managed persistence implementation which provides:
    - An Object / Relational (O / R) mapping tool (part of the Sun ONE Application Server Assembly Tool) that creates XML deployment descriptors for EJB JAR files that contain beans that use container-managed persistence
    - Support for compound (multi-column) primary keys
    - Support for sophisticated custom finder methods
    - Standards-based query language (EJB QL)
    - Container-managed persistence runtime, which supports the following JDBC drivers / databases:
        - Oracle 8i, Oracle 9
        - Sybase 12
        - Microsoft SQLServer 2000
        - Pointbase 4.2 (not available in the pre-installed version of the Sun ONE Application Server with Solaris)
- Support for third-party object-to-relational (O / R) mapping tools. An explanation of the third-party API is contained in “Third-Party Pluggable Persistence Manager API,” on page 110.

About Container-Managed Persistence

An entity bean using container-managed persistence delegates the management of its state (or persistence) to the Sun ONE Application Server container. Rather than write the JDBC code that is needed to implement bean-managed persistence, a developer implementing container-managed persistence uses tools to create the bean's deployment descriptors. The deployment descriptors then provide the information that the container uses to map bean fields to columns in a relational database.

An EJB container needs two things to support container-managed persistence:

- Mapping—Information on how to map an entity bean to a resource, such as a table in a relational database
- Runtime environment—A container-managed persistence runtime environment that uses the mapping information to perform persistence operations on each bean

This section addresses the following container-managed persistence topics:

- CMP Components
- Relationships
- Abstract Schema
- Deployment Descriptors
- Persistence Manager

CMP Components

Unlike bean-managed persistence, container-managed persistence does not require you to write database access calls in the methods of the entity bean class. Because persistence is handled by the container at runtime, you must specify in the deployment descriptor those persistence fields and relationships for which the container must handle data access. You access persistent data using the accessor methods that are defined for the abstract persistence schema.

An entity bean that uses container-managed persistence consists of several components that interoperate:

- The abstract bean class, written by you.
- The remote interface, written by you.

- The local interface, written by you.
- The deployment descriptor, written by you.
- An optional primary key class, written by you.
- The concrete bean class, generated by the container-managed persistence implementation.

  This class inherits from the abstract bean class and uses information from the deployment descriptor. Accessor (read) and mutator (write) methods in the bean class are implemented here to the concrete state class.
- The concrete remote bean implementation class, generated by the container-managed persistence implementation.
- The EJBObject (skeleton), generated by the container-managed persistence implementation.
- The remote stub, generated by the container-managed persistence implementation.

The following classes are used for container-managed persistence:

- Generation class—Called from the `ejbc` compile utility; generates the concrete classes.
- Generated classes—Use container-managed persistence to effect persistence behavior at server runtime.
- Management classes—Collect and report statistics at server runtime.

Relationships

NOTE This section applies only if you are using container-managed persistence 2.0 beans.

A *relationship* allows you to navigate from an object to its related objects. Relationships can be either bidirectional or unidirectional.

- Bidirectional—Each entity bean has a relationship field that refers to the other bean. Through the relationship field, an entity bean's code can access its related object. If an entity bean has a relationship field, we often say it "knows" about its related object.

- Unidirectional—Only one entity bean has a relationship field that refers to the other.

NOTE Even if a relationship is unidirectional, if you make a change to that relationship, other enterprise beans will be affected if they are associated with that relationship.

A *container-managed relationship* (CMR) between fields in a pair of classes allows operations on one side of the relationship to affect the other side. At runtime, if a field in one instance is modified to refer to another instance, the referred instance will have its relationship field modified to reflect the change in relationship.

NOTE No warning is given if you delete one object in a managed relationship. Container-managed persistence automatically nullifies the relationship on the foreign key side and deletes the object without asking for confirmation.

In the Java code, relationships are represented by object reference (either collections or fields that are typed to an EJB local interface), depending on the relationship cardinality. A relationship can be one-to-one, one-to-many, or many-to-many, depending on the number of instances of each class in the relationship. In the database, this might be represented by foreign key columns and, in the case of many-to-many relationships, join tables.

The following sections describe the various types of relationships:

- One-to-One Relationships
- One-to-Many Relationships
- Many-to-Many Relationships

One-to-One Relationships

With one-to-one relationships, there is a single-valued field in each class whose type is the local interface of the other bean type. Any change to the field on either side of the relationship is handled as a relationship change. If the field on one side is changed from null to non-null, then the field on the other side is changed to refer to this instance. If the field on the other side had been non-null, that other relationship is made null before the change is made.

One-to-Many Relationships

With one-to-many relationships, there is a single-valued field on the many side and a multi-valued field (collection) on the one side.

If an instance is added to the collection field, the field in the new instance is updated to reference the instance containing the collection field. If an instance is deleted from the collection, the field on the instance is nullified.

Any change, addition or removal of a field on the many side is handled as a relationship change. If the field on the many side is changed from null to non-null, this instance is added to the collection-valued field on the one side. If the field on the many side is changed from non-null to null, then this instance is removed from the collection-valued field on the one side.

Many-to-Many Relationships

With many-to-many relationships, there are multi-valued, or *collection*, fields on both sides of the relationship. Any change to the contents of the collection on either side of the relationship is handled as a relationship change. If an instance is added to the collection on this side, then this instance is added to the collection on the other side. If an instance is removed from a collection on this side, then this instance is removed from the collection on the other side.

Abstract Schema

Part of an entity bean's deployment descriptor, the *abstract schema* defines the bean's persistent fields and relationships. The term abstract distinguishes this schema from the physical schema of the underlying data store.

You specify the name of an abstract schema in the deployment descriptor. This name is referenced by queries written in the EJB Query Language (EJB QL). For an entity bean using container-managed persistence, you must define an EJB-QL query for every finder method (except `findByPrimaryKey`). The EJB-QL query determines the query that is executed by the EJB container when the finder method is invoked.

Example

```
<ejb-relation>
   <ejb-relation-name>OrderLineItem</ejb-relation-name>
   <ejb-relationship-role>
      <ejb-relationship-role-name>
         OrderHasLineItems
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
         <ejb-name>Order</ejb-name>
      </relationship-role-source>
      <cmr-field>
         <cmr-field-name>lineItems</cmr-field-name>
         <cmr-field-type>java.util.Collection</cmr-field-type>
       </cmr-field>
   </ejb-relationship-role>
<ejb-relationship-role>
   <ejb-relationship-role-name>
      LineItemInOrder
   </ejb-relationship-role-name>
   <multiplicity>Many</multiplicity>
   <relationship-role-source>
         <ejb-name>LineItemEJB</ejb-name>
      </relationship-role-source>
   </ejb-relationship-role>
</ejb-relation>
```

Deployment Descriptors

If your container-managed fields are to be mapped to database fields, you must provide mapping information to the deployer. Each module with container-managed persistence beans must have the following files for the deployment process

- `ejb-jar.xml`—Contains information such as the transactional attributes of the beans and the fields of a bean that are going to be container-managed.
- `sun-ejb-jar.xml`—The standard file for assembling enterprise beans. Refer to "Elements in the sun-ejb-jar.xml File," on page 176 and "Sample EJB XML Files," on page 213 for information.
- `sun-cmp-mappings.xml`—The file for mapping container-managed persistence. Refer to "Elements in the sun-cmp-mappings.xml File," on page 112 and "Sample Schema Definition," on page 121 for information.

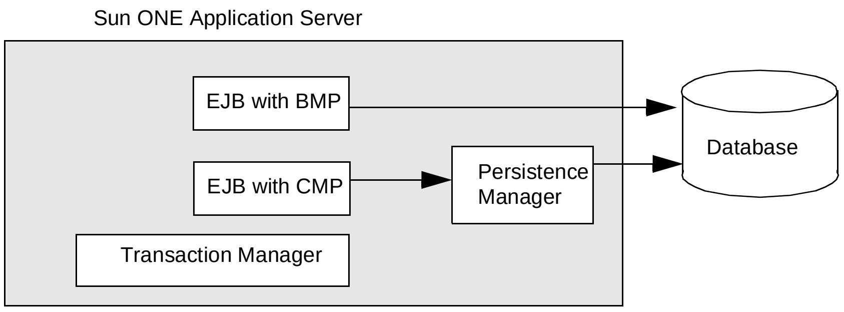
Persistence Manager

In the Sun ONE Application Server, the container-managed persistence model is based on the Pluggable Persistence Manager API which provides the role of the persistence manager in defining and supporting the mapping between an entity bean and the persistence store.

The *persistence manager* is the component responsible for the persistence of the entity beans installed in the container. The classes provided by the persistence manager vendor are responsible for managing the relationships between the entity beans, and for managing access to their persistent state. The persistence manager vendor is also responsible for providing the implementation of the Java classes that are used in maintaining the container-managed relationships. The persistence manager uses the data source registry provided by the container to access data sources.

The following figure illustrates how persistence works in the Sun ONE Application Server environment.

Entity Bean Flow



It is also possible to write custom persistence managers to support legacy systems, or to implement caching strategies that improve performance for your container-managed persistence solution.

Using Container-Managed Persistence

Implementation for entity beans that use container-managed persistence is mostly a matter of mapping and assembly/deployment.

NOTE Java types assigned to the container-managed fields must be restricted to the following: Java primitive types, Java serializable types, and references to EJB remote or remote home interfaces.

This section addresses the following topics:

- Process Overview

- Mapping Capabilities
- Supported Data Types for Mapping
- BLOB Support
- Using the capture-schema Utility
- Mapping Fields and Relationships
- Configuring the Resource Manager
- Using EJB QL
- Configuring Queries for 1.1 Finders

Process Overview

The container-managed persistence process consists of three operations: mapping, deploying, and running. These operations are accomplished as described in the following phases:

- Phase 1. Creating the mapping deployment descriptor file
- Phase 2. Generating and compiling concrete beans and delegates
- Phase 3. Running in the Sun ONE Application Server runtime

Phase 1. Creating the mapping deployment descriptor file

NOTE The Sun ONE Studio IDE will create this descriptor automatically for deployment.

This phase can be done concurrent with development of the container-managed persistence beans in the Sun ONE Studio 4 IDE, or after development while preparing for deployment.

During this phase, you map CMP fields and CMR fields (relationships) to the database. A primary table is selected for each container-managed persistence bean, and optionally, multiple secondary tables. CMP fields are mapped to columns in either the primary or secondary table(s). CMR fields are mapped to pairs of column lists (normally, column lists are the list of columns associated with pairs of primary and foreign keys).

- The mapping is saved in a file which conforms to the `sun-cmp-mapping_1_0.dtd`. The resulting XML file is packaged with the user-defined bean classes in an EJB JAR file and must be named `META-INF/sun-cmp-mappings.xml`.
- Errors are reported during the deployment process. Errors may be triggered from within the Sun ONE Studio 4 environment or at the command line.
- The mapping information is developed in conjunction with the database schema file. This file must be captured using the Sun ONE Studio 4 IDE ("Capturing a Schema," on page 217) or the capture-schema utility ("Using the capture-schema Utility," on page 93).
- If the database table structure is changed, you first capture the schema of the updated tables after the database administrator updates the tables. You then remap the CMP fields and relationships.

NOTE There is no automatic procedure for performing this remapping; you must do it manually.

Phase 2. Generating and compiling concrete beans and delegates

This phase is done during deployment of an EJB application to the Sun ONE Application Server. During this phase, deployment information is combined with the mapping information created during Phase 1.

The following files are generated:

- The concrete bean file, which extends the abstract bean written by you

  The concrete bean implements the EJB life cycle methods `ejbSetEntityContext`, `ejbUnsetEntityContext`, `ejbCreate`, `ejbRemove`, `ejbLoad`, `ejbStore`. It also contains implementation of `getXXX` and `setXXX` for each CMP field and the CMR field, `ejbFindByPrimaryKey`, other finder methods, and any selector methods defined by the user.
- The compiled EJB-QL for finder and selector methods, stored as a properties file

  This file contains the container-managed persistence query parameter list, the query filter, the query ordering expression, the query candidate class name, and the query result type.
- A generation log file that reports errors to you, including EJB-QL syntax and usage errors

- State and helper classes

Phase 3. Running in the Sun ONE Application Server runtime

At runtime, the information provided at deployment is used to service requests on entities implemented as enterprise beans.

Mapping Capabilities

Mapping refers to the ability to tie an object-oriented model to a relational model of data, usually the schema of a relational database. The container-managed persistence implementation provides the ability to tie a set of interrelated classes containing data and associated behaviors to the interrelated meta-data of the schema. You can then use this object representation of the database to form the basis of a Java application. You can also customize this mapping to optimize these underlying classes for the particular needs of an application.

The result is a single data model through which you can access both persistent database information and regular transient program data. You only need to understand the Java programming language objects; you do not need to know or understand the underlying database schema.

Information on the container-managed persistence DTD and XML file elements is contained in “Elements in the sun-cmp-mappings.xml File,” on page 112.

Mapping Features

The mapping capabilities provided by the Sun ONE Application Server include:

- Mapping a container-managed persistence bean to a single table
- Mapping a container-managed persistence bean to multiple tables
- Mapping container-managed persistence fields to columns
- Mapping container-managed persistence fields to different column types
- Mapping tables with compound primary keys
- Mapping container-managed persistence relationships to foreign key columns
- Mapping tables with overlapping primary and foreign keys

Mapping Tool

The mapping tool generates information that maps the entity bean's container-managed fields to a data source, such as a column in a relational database table. This mapping information is stored in an XML file.

The *meet-in-the-middle mapping* of the container-managed persistence implementation creates a custom mapping between an existing schema and existing Java classes, using the Mapping Tool.

Mapping Techniques

A container-managed persistence class should represent a data entity, such as an employee or a department. To model a specific data entity, you add persistent fields to the class that correspond to the columns in the data store.

The simplest kind of modeling is to have a persistence-capable class represent a single table in the data store, with a persistent field for each of the table's columns. An `Employee` class, for example, would have persistent fields for all the columns found in the `EMPLOYEE` table of the data store, such as `lastname`, `firstname`, `department`, and `salary`.

NOTE You can choose to have only a subset of the data store columns used as persistent fields, but if a field is persistent, it must be mapped.

Information on how to use Sun ONE Studio 4 to map container-managed persistence for enterprise beans is contained in the Sun ONE Application Server Integration Module for the Sun ONE Studio 4 online help.

Supported Data Types for Mapping

Container-managed persistence supports a set of JDBC 1.0 SQL data types that are used in mapping Java data fields to SQL types. Supported JDBC 1.0 SQL data types are listed in the following table.

Supported JDBC 1.0 SQL Data Types		
BIGINT	DOUBLE	SMALLINT
BIT	FLOAT	TIME
BLOB	INTEGER	TIMESTAMP
CHAR	LONGVARCHAR	TINYINT

Supported JDBC 1.0 SQL Data Types		
`DATE`	`NUMERIC`	`VARCHAR`
`DECIMAL`	`REAL`	

The following table contains suggested mappings.

Suggested Data Type Mappings

Java Type	JDBC Type	Nullability
`boolean`	`BIT`	`NON NULL`
`java.lang.Boolean`	`BIT`	`NULL`
`byte`	`TINYINT`	`NON NULL`
`java.lang.Byte`	`TINYINT`	`NULL`
`double`	`FLOAT`	`NON NULL`
`java.lang.Double`	`FLOAT`	`NULL`
`double`	`DOUBLE`	`NON NULL`
`java.lang.Double`	`DOUBLE`	`NULL`
`float`	`REAL`	`NON NULL`
`java.lang.Float`	`REAL`	`NULL`
`int`	`INTEGER`	`NON NULL`
`java.lang.Integer`	`INTEGER`	`NULL`
`long`	`BIGINT`	`NON NULL`
`java.lang.Long`	`BIGINT`	`NULL`
`long`	`DECIMAL (scale==0)`	`NON NULL`
`java.lang.Long`	`DECIMAL (scale==0)`	`NULL`
`long`	`NUMERIC (scale==0)`	`NON NULL`
`java.lang.Long`	`NUMERIC (scale==0)`	`NULL`
`short`	`SMALLINT`	`NON NULL`
`java.lang.Short`	`SMALLINT`	`NULL`
`java.math.BigDecimal`	`DECIMAL (scale!=0)`	`NON NULL`
`java.math.BigDecimal`	`DECIMAL (scale!=0)`	`NULL`

Suggested Data Type Mappings *(Continued)*

Java Type	**JDBC Type**	**Nullability**
`java.math.BigDecimal`	`NUMERIC`	`NULL`
`java.math.BigDecimal`	`NUMERIC`	`NON NULL`
`java.lang.String`	`CHAR`	`NON NULL`
`java.lang.String`	`CHAR`	`NULL`
`java.lang.String`	`VARCHAR`	`NON NULL`
`serializable`	`BLOB`	`NULL`

BLOB Support

Binary Large Object (BLOB) is a data type used to store and retrieve complex object fields. BLOBs are binary or serializable objects, such as pictures, that translate into large byte arrays which are then serialized into CMP fields.

NOTE On Oracle, using the Oracle thin driver (JDBC type 4), it is not possible to insert more than 2000 bytes of data into a column. To circumvent this problem, use the OCI driver (JDBC type 2).

To enable BLOB support in the Sun ONE Application Server environment:

1. Declare the variable in the bean class with a serializable type.
2. Edit the XML file by declaring the CMP mapping deployment descriptor in the `sun-cmp-mappings.xml` file.
3. Create the BLOB in the database.

NOTE Performance may be negatively impacted due to the size of the BLOB object.

Example

```
<cmp-field-mapping>
   <field-name>syllabus</field-name>
   <column-name>COURSE.SYLLABUS</column-name>
</cmp-field-mapping>
```

Example

```
/*****************************************************
Serializable class Syllabus : BLOB Testing
*****************************************************/

package collegeinfo
public class Syllabus implements java.io.Serializable
{
   public String author;
   public String syllabi;
}

Schema for Course:

table course
------------
courseId Number
deptId Number
courseName Varchar
syllabus BLOB
```

Using the capture-schema Utility

Mapping information is developed by first capturing the database schema. Use the `capture-schema` command to store the database metadata (schema) in a file for use in mapping and execution. You can also use the Sun ONE Studio (formerly Forte for Java) IDE to capture the database schema; refer to "Capturing a Schema," on page 217.

Syntax

`capture-schema -dburl` *url* `-username` *name* `-password` *password* `-driver` *ajdbcdriver* `[-schemaname` *name*`] [-table` *TableName*`]* [-out` *filename*`]`

Where:

`-dburl` *url*: Specifies the JDBC URL expected by the driver for accessing a database.

`-username` *name*: Specifies the user name for authenticating access to a database.

`-password` *password*: Specifies the password for accessing the selected database.

`-driver` *ajdbcdriver:* Specifies the JDBC driver class name. This class must be in your CLASSPATH.

`-schemaname` *name*: Specifies the name of the user schema being captured. If not specified, the default will capture metadata for all tables from all the schemas accessible to this user.

NOTE Specifying this parameter is highly recommended. If more than one schema is accessible for this user, more than one table with the same name might be captured, which will cause problems.

`-table` *TableName*: Specifies a table name. Multiple table names can be specified. If not specified, all the tables in the database schema will be captured.

`-out`: Specifies the output target. Defaults to `stdout`. To be able to use the output for the CMP mapping, the output file name must have the `.dbschema` suffix.

For container-managed persistence mapping, the `-out` parameter correlates to the `schema` subelement of the `sun-cmp-mapping` element in the `sun-cmp-mapping_1_0.dtd` file:

```
<!ELEMENT sun-cmp-mapping ( schema, entity-mapping+) >
```

In the `sun-cmp-mappings.xml` file, this element must be represented without the `.dbschema` suffix. For example:

```
<schema>RosterSchema</schema>
```

NOTE If no table flags are given, all the tables in the database are captured in the schema.

Example

```
capture-schema -dburl jdbc:pointbase:server://localhost:9092/sample
-username public -password public -driver
com.pointbase.jdbc.jdbcUniversalDriver -out RosterSchema.dbschema
```

Mapping Fields and Relationships

This section discusses how to map the fields and relationships of your entity beans by editing the `sun-cmp-mappings.xml` deployment descriptor. This can be done either manually (provided you are proficient in editing XML) or using the Sun ONE Application Server assembly and deployment tools.

A container-managed persistence bean has a name, a primary table, one or more fields, zero or more relationships, and zero or more secondary tables, plus flags for consistency checking. You will need to map the CMP fields and CMR fields to the database using the elements in the `sun-cmp-mappings.xml` file. CMP fields are mapped to columns in either the primary or secondary database table(s); CMR fields are mapped to pairs of column lists.

An alphabetic listing of the mapping elements in the container-managed persistence deployment descriptors is contained in "Elements in the sun-cmp-mappings.xml File," on page 112. A sample XML file is contained in "Sample Schema Definition," on page 121.

This section contains instructions for accomplishing the following mapping tasks:

- Specifying the Beans to Be Mapped
- Specifying the Mapping Components
- Specifying Field Mappings
- Specifying Relationships

Specifying the Beans to Be Mapped

You must start by using the following elements to specify the database schema and the container-managed persistence beans being mapped:

- sun-cmp-mappings
- sun-cmp-mapping
- schema
- entity-mapping

sun-cmp-mappings

Specifies the collection of subelements for all the beans that will be mapped in an EJB JAR collection.

Subelement is `sun-cmp-mapping`.

Example

Refer to "Sample Schema Definition," on page 121.

sun-cmp-mapping

Specifies beans mapped to a particular schema.

Subelements are `schema`, `entity-mapping`.

schema

Specifies the path to the schema file. Only one is required. For further information, refer to "Sample EJB QL Queries," on page 124 and "Capturing a Schema," on page 217.

Example

```
<schema>RosterSchema</schema>
```

entity-mapping

Specifies the mapping of beans to database columns.

Subelements are `ejb-name`, `table-name`, `cmp-field-mapping`, `cmr-field-mapping`, `secondary-table`, `consistency`.

Example

For an example, see "entity-mapping," on page 96.

Specifying the Mapping Components

The next step is to use the following elements to specify components that are part of the mapping, and to indicate how consistency checking will occur.

- entity-mapping
- ejb-name
- table-name
- secondary-table
- consistency

entity-mapping

Specifies the mapping of beans to database columns.

Subelements are `ejb-name`, `table-name`, `cmp-field-mapping`, `cmr-field-mapping`, `secondary-table`, `consistency`.

Example

```
<entity-mapping>
   <ejb-name>Player</ejb-name>
   <table-name>PLAYER</table-name>
   <cmp-field-mapping>
      <field-name>salary</field-name>
      <column-name>PLAYER.SALARY</column-name>
   </cmp-field-mapping>
   <cmp-field-mapping>
      <field-name>playerId</field-name>
      <column-name>PLAYER.PLAYER_ID</column-name>
   </cmp-field-mapping>
   <cmp-field-mapping>
      <field-name>position</field-name>
      <column-name>PLAYER.POSITION</column-name>
   </cmp-field-mapping>
      <field-name>name</field-name>
      <column-name>PLAYER.NAME</column-name>
   </cmp-field-mapping>
   <cmr-field-mapping>
      <cmr-field-name>teamId</cmr-field-name>
      <column-pair>
         <column-name>PLAYER.PLAYER_ID</column-name>
         <column-name>TEAMPLAYER.PLAYER_ID</column-name>
      </column-pair>
      <column-pair>
         <column-name>TEAMPLAYER.TEAM_ID</column-name>
         <column-name>TEAM.TEAM_ID</column-name>
      </column-pair>
   </cmr-field-mapping>
</entity-mapping>
```

ejb-name

Specifies the name of the entity bean in the `ejb-jar.xml` file to which the container-managed persistence beans relates. One is required.

Example

```
<ejb-name>Player</ejb-name>
```

table-name

Specifies the name of a database table. The table must be present in the database schema file. One is required.

Example

```
<table-name>PLAYER</table-name>
```

secondary-table

Specifies a bean's secondary table(s). Optional.

Subelements are `table-name`, `column-pair`.

Example

This secondary table example adds an email field in the `StudentEjb` class.

```
public abstract class StudentEJB implements EntityBean {

   /***************************************************
   Write ur set,get methods for Entity bean variables and
   business methods here
   ***************************************************/
   //Access methods for CMP fields
   public abstract Integer getStudentId();
   public abstract void setStudentId(Integer studentId);
   public abstract String getStudentName();
   public abstract void setStudentName(String studentName);

   public abstract void setEmail(String Email); <-----Column from
                                                 Secondary Table
```

The Student and the Email table should be related by a foreign key. The schema for the Email table may look like this:

```
Table Email:
------------
Student_id Number
email varchar

Table Student:
--------------
StudentId Number
StudentName varchar
deptId Number
AddressId Number
AccountId Varchar
```

When adding the secondary table, the tables will both apply to the same enterprise bean.

consistency

Specifies container behavior in guaranteeing transactional consistency of the data in the bean. Optional. If the consistency checking flag element is not present, none is assumed.

The following table describes the elements used for consistency checking.

Consistency Flags

Flag Element	Description
`check-all-at-commit`	This flag is not implemented for Sun ONE Application Server 7.
`check-modified-at-commit`	Checks modified instances at commit time.
`lock-when-loaded`	A lock is implemented when the data is loaded.
`lock-when-modified`	This flag is not implemented for Sun ONE Application Server 7.
`none`	No consistency checking occurs.

Specifying Field Mappings

Field mapping is done using the following elements:

- cmp-field-mapping
- field-name
- column-name
- read-only
- fetched-with
- level
- named-group
- none

cmp-field-mapping

The `cmp-field-mapping` element associates a field with one or more columns that it maps to. The column can be from a bean's primary table or any defined secondary table. If a field is mapped to multiple columns, the column listed first is used as a SOURCE for getting the value from the database. The columns are updated in the order they appear. There is one `cmp-field-mapping` element for each `cmp-field` element defined in the EJB JAR file.

A field can be marked as read-only.

Subelements are `field-name`, `column-name`, `read-only`, and `fetched-with`.

Example

```
<cmp-field-mapping>
   <field-name>name</field-name>
   <column-name>LEAGUE.NAME</column-name>
</cmp-field-mapping>
```

field-name

Specifies the Java identifier of a field. This identifier must match the value of the `field-name` subelement of the `cmp-field` that is being mapped. One is required.

Example

```
<field-name>name</field-name>
```

column-name

Specifies the name of a column from the primary table, or the table qualified name (TABLE.COLUMN) of a column from a secondary or related table. One or more is required.

NOTE When mapping multiple columns, any JAVA type can be used.

Example

```
<column-name>PLAYER.NAME</column-name>
```

Example

Use this with non-normalized tables where the same information appears in multiple places, and the information needs to be kept synchronized if it is updated.

```
public abstract class StudentEJB implements EntityBean {
.
.
.

public abstract String getInstallments();
```

The three columns from the student table can be mapped to a single installments column in the Student enterprise bean.

```
Table student:
.
.
.
installment1 Number
installment2 Number
installment3 Number
```

The same value will be written to all the columns in the database.

read-only

The `read-only` flag indicates that a field is read-only.

Example

```
<read-only>name</read-only>
```

fetched-with

Specifies the fetch group configuration for fields and relationships. A field may participate in a hierarchical or independent fetch group. Optional.

The fetched-with element has different default values based on its context.

- If there is no `fetched-with` sub-element of a `cmp-field-mapping`, the default value is assumed to be:

  ```
  <fetched-with><level>0</level></fetched-with>
  ```

- If there is no `fetched-with` sub-element of a `cmr-field-mapping`, the default value is assumed to be:

  ```
  <fetched-with><none/></fetched-with>
  ```

Subelements are `level`, `named-group`, or `none`.

level

Specifies the name of a hierarchical fetch group. The value must be an integer. Fields and relationships that belong to a hierarchical fetch group of equal (or lesser) value are fetched at the same time. The value of `level` must be greater than zero. Only one is allowed.

named-group

Specifies the name of an independent fetch group. All the fields and relationships that are part of a named group are fetched at the same time. Only one is allowed.

none

A consistency level flag that indicates that this field or relationship is fetched by itself.

Specifying Relationships

The following elements are used to specify the mapping for container-managed relationships:

- cmr-field-mapping
- cmr-field-name
- column-pair
- fetched-with

cmr-field-mapping

A container-managed relationship field has a name and one or more column pairs that define the relationship. There is one `cmr-field-mapping` element for each `cmr-field`. A relationship can also participate in a fetch group.

Subelements are `cmr-field-name`, `column-pair`, `fetched-with`.

Example

```
<cmr-field-mapping>
   <cmr-field-name>teamId</cmr-field-name>
   <column-pair>
      <column-name>PLAYER.PLAYER_ID</column-name>
      <column-name>TEAMPLAYER.PLAYER_ID</column-name>
   </column-pair>
   <column-pair>
      <column-name>TEAM.TEAM_ID</column-name>
      <column-name>TEAMPLAYER.TEAM_ID</column-name>
   </column-pair>
   <fetched-with>
      <none/>
   </fetched-with>
</cmr-field-mapping>
```

cmr-field-name

Specifies the Java identifier of a field. This must match the value of the `cmr-field-name` subelement of the `cmr-field` that is being mapped. One is required.

Example

```
<cmr-field-name>team</cmr-field-name>
```

column-pair

Specifies the pair of related columns in two database tables. One or more is required.

The columns names are specified in the `column-name` element.

Example

```
<column-pair>
   <column-name>PLAYER.PLAYER_ID</column-name>
   <column-name>TEAMPLAYER.PLAYER_ID</column-name>
</column-pair>
```

column-name

Specifies the name of a column from the primary table, or the table qualified name (TABLE.COLUMN) of a column from a secondary or related table. Two are required as subelements of a column-pair.

Example

```
<column-name>PLAYER.NAME</column-name>
```

fetched-with

Specifies the fetch group configuration for fields and relationships. A field may participate in a hierarchical or independent fetch group. Optional.

The fetched-with element has different default values based on its context.

- If there is no `fetched-with` sub-element of a `cmp-field-mapping`, the default value is assumed to be:

  `<fetched-with><level>0</level></fetched-with>`

- If there is no `fetched-with` sub-element of a `cmr-field-mapping`, the default value is assumed to be:

  `<fetched-with><none/></fetched-with>`

Subelements are `level`, `named-group`, or `none`.

Configuring the Resource Manager

The resource manager used by the container-managed persistence implementation is `PersistenceManagerFactory`, which is configured using the Sun ONE Application Server DTD file, `sun-server_7_0-0.dtd`.

Refer to the Sun ONE Application Server *Administrator's Guide* for information on creating a new persistence manager.

To deploy an EJB module that contains container-managed persistence beans, you need to add the following information to the `sun-ejb-jar.xml` deployment descriptor.

1. Specify the Persistence Manager to be used for the deployment:

```
<pm-descriptors>
   <pm-descriptor>
      <pm-identifier>SunONE</pm-identifier>
      <pm-version>1.0</pm-version>
<pm-class-generator>com.iplanet.ias.persistence.internal.ejb.ejbc.J
DOCodeGenerator
</pm-class-generator>
      <pm-mapping-factory>com.iplanet.ias.cmp.
         NullFactory</pm-mapping-factory>
   </pm-descriptor
   <pm-inuse>
      <pm-identifier>SunONE</pm-identifier>
      <pm-version>1.0</pm-version>
   </pm-inuse>
</pm-descriptors>
```

2. Specify the JNDI name of the Persistence Manager's resource (listed under `persistence-manager-factory-resource` entry in the `server.xml` file) and the JNDI name for `cmp-resource`. This name will be used at run time to manage persistent resources.

For example, if you have the following entry in the `server.xml` file:

```
<persistence-manager-factory-resource
   factory-class="com.sun.jdo.spi.persistence.support.
      sqlstore.impl.PersistenceMan
   gerFactoryImpl"
   enabled="true"
   jndi-name="jdo/pmf"
   jdbc-resource-jndi-name="jdo/pmfPM"
</persistence-manager-factory-resource>
```

Set the CMP resource as:

```
<cmp-resource>
   <jndi-name>jdo/pmf</jndi-name
</cmp-resource>
```

NOTE The Sun ONE Studio IDE will create `pm-descriptors` as part of this descriptor automatically for deployment. Information on how to set up the container-managed persistence resources is contained in the Sun ONE Application Server Integration Module for the Sun ONE Studio 4 online help.

Using EJB QL

The Enterprise JavaBeans Specification, v2.0 specifies a new query language (EJB QL) that can be used to define portable queries for the finder and select methods of CMP beans. These queries use a SQL-like syntax to select entity objects or field values based on the abstract schema types and relationships of CMP beans.

Finder methods are defined in the home and/or local home interfaces of the bean, and return instances of the same bean. Select methods are defined only in the abstract bean class, and can be used for selecting entity objects of any local or remote type as well as field values for beans from the same schema.

For more information, refer to the Chapter 11, "EJB QL: EJB Query Language for Container-Managed Persistence Query Methods" in the Enterprise JavaBeans Specification, v2.0.

Some EJB QL sample queries are contained in “Sample EJB QL Queries,” on page 124.

Configuring Queries for 1.1 Finders

The Enterprise JavaBeans Specification, v1.1 spec does not specify the format of the finder method description. The Sun ONE Application Server uses Java Data Objects Query Language (JDOQL) queries to implement finder and selector methods. For EJB 2.0, the container automatically maps an EJB QL query to JDOQL. For EJB 1.1, this mapping is partially done by the developer. You can specify the following elements of the underlying JDOQL query:

- Filter expression—A Java-like expression that specifies a condition that each object returned by the query must satisfy. Corresponds to the WHERE clause in EJB QL.
- Query parameter declaration—Specifies the name and the type of one or more query input parameters. Follows the syntax for formal parameters in the Java language.
- Query variable declaration—Specifies the name and type of one or more query variables. Follows the syntax for local variables in the Java language. Query variables might be used in the filter to implement joins.

The Sun ONE Application Server-specific deployment descriptor (`sun-ejb-jar.xml`) provides the following elements to store the EJB 1.1 finder method settings:

```
query-filter
query-params
query-variables
```

The Sun ONE Application Server constructs a JDOQL query using the persistence capable class of the EJB 1.1 entity bean as the candidate class. It adds the filter, parameter declarations, and variable declarations as specified by the developer to the JDOQL query. It executes the query and passes the parameters of the finder method to the `execute` call. The objects from the JDOQL query result set are converted into primary key instances to be returned by the EJB 1.1 `ejbFind` method.

The JDO specification (see JSR 12) provides a comprehensive description of JDOQL. The following information summarizes the elements used to define EJB 1.1 finders.

Query Filter Expression

The filter expression is a String containing a boolean expression evaluated for each instance of the candidate class. If the filter is not specified, it defaults to true. Rules for constructing valid expressions follow the Java language, with the following differences:

- Equality and ordering comparisons between primitives and instances of wrapper classes are valid.
- Equality and ordering comparisons of Date fields and Date parameters are valid.
- Equality and ordering comparisons of String fields and String parameters are valid.
- White space (non-printing characters space, tab, carriage return, and line feed) is a separator and is otherwise ignored.
- The following assignment operators are not supported:
    - =, +=, etc.
    - pre- and post-increment
    - pre- and post-decrement
- Methods, including object construction, are not supported, except for:

    ```
    Collection.contains(Object o)
    Collection.isEmpty()
    String.startsWith(String s)
    String.endsWith(String e)
    ```

    In addition, the Sun ONE Application Server supports the following non-standard JDOQL methods:

    ```
    String.like(String pattern)
    String.like(String pattern, char escape)
    String.substring(int start, int length)
    String.indexOf(String str), String.indexOf(String str, int
    start)
    String.length()
    Math.abs(numeric n), and Math.sqrt(double d)
    ```

- Navigation through a null-valued field, which would throw `NullPointerException`, is treated as if the subexpression returned false.

NOTE Comparisons between floating point values are by nature inexact. Therefore, equality comparisons (== and !=) with floating point values should be used with caution. Identifiers in the expression are considered to be in the name space of the candidate class, with the addition of declared parameters and variables. As in the Java language, this is a reserved word, and refers to the current instance being evaluated.

The following expressions are supported:

- Operators applied to all types where they are defined in the Java language:
  - relational operators (==, !=, >, <, >=, <=)
  - boolean operators (&, &&, |, ||, ~, !)
  - arithmetic operators (+, -, *, /)

  String concatenation is supported only for String + String.
- Parentheses to explicitly mark operator precedence
- Cast operator
- Promotion of numeric operands for comparisons and arithmetic operations. The rules for promotion follow the Java rules (see the numeric promotions of the Java language specification) extended by BigDecimal, BigInteger, and numeric wrapper classes.

Query Parameter

The parameter declaration is a String containing one or more parameter type declarations separated by commas. This follows the Java syntax for method signatures.

Query Variables

The type declarations follow the Java syntax for local variable declarations.

Example1

The following query returns all players called Michael. It defines a filter that compares the name field with a string literal:

```
"name == \"Michael\""
```

The `finder` element of the `sun-ejb-jar.xml` file would look like this:

```
<finder>
   <method-name>findPlayerByName</method-name>
   <query-filter>name == "Michael"</query-filter>
</finder>
```

Example 2

This query returns all products in a specified price range. It defines two query parameters which are the lower and upper bound for the price: double low, double high. The filter compares the query parameters with the price field:

```
   "low < price && price < high"
```

The `finder` element of the `sun-ejb-jar.xml` file would look like this:

```
<finder>
   <method-name>findInRange</method-name>
   <query-params>double low, double high</query-params>
   <query-filter>low &lt; price &amp;&amp; price &lt
   high</query-filter
</finder>
```

Example 3

This query returns all players having a higher salary than the player with the specified name. It defines a query parameter for the name `java.lang.String name`. Furthermore, it defines a variable for the player to compare with. It has the type of the persistence capable class that corresponds to the bean:

```
   mypackage.PlayerEJB_170160966_JDOState p
```

The filter compares the salary of the current player denoted by this keyword with the salary of the player with the specified name:

```
   (this.salary > p.salary) && (p.name == name)
```

The `finder` element of the `sun-ejb-jar.xml` file would look like:

```
<finder>
   <method-name>findByHigherSalary</method-name>
   <query-params>java.lang.String name</query-params>
   <query-filter>
      (this.salary &gt; p.salary) &amp;&amp;
      (p.name ==name)
   </query-filter>
   <query-variables></query-variables
</finder>
```

Third-Party Pluggable Persistence Manager API

Container-managed persistence in the EJB container can support persistence vendors integrating their runtimes into the Sun ONE Application Server using the Sun ONE Application Server Pluggable Persistence Manager API. The API describes integration requirements at deployment, at code-generation, and at runtime. It supports callouts to implement the concrete bean implementations when EJBs are compiled.

The Sun ONE Application Server enables the container-managed persistence implementation to use its startup framework to load classes and to register the persistence manager. The Pluggable Persistence Manager API also supports integration requirements with regard to transactions and dynamic deployment.

In general, the objective is that any third-party container-managed persistence solution that fully supports the Enterprise JavaBeans Specification, v2.0 can be made to work with the Sun ONE Application Server.

To use a third-party tool:

1. Build your enterprise beans using the third-party O/R mapping tool.
2. Deploy the beans using the Assembly Tool or the command-line interface.

Third-party persistence tools must use Java Database Connectivity (JDBC) resources or Java Connector API (JCA) resources at runtime to access relational data sources. This allows the pluggable persistence managers to automatically use the Connection Pooling, transaction handling, and security management features of the container. Third-party vendors will be able to plug in their concrete class generators and their mapping factory to generate a valid vendor-specific mapping object model.

The configuration requirements specify a number of properties which must be defined for a bean, including:

- The persistence mechanism
- The persistence vendor/version
- Additional information required by the persistence mechanism

Restrictions and Optimizations

This section discusses any restrictions and performance optimizations you should be aware of in implementing container-managed persistence for entity beans.

- Unique Database Schema Names in EAR File
- Limitations on Container-Managed Persistence Protocol
- Restrictions on Remote Interfaces

Unique Database Schema Names in EAR File

In a situation where there are multiple JAR files within an EAR file, for example `jar1` and `jar2`, any corresponding database schema files for `jar1` and `jar2` must have unique fully qualified names.

In other words, the database schema file names must be unique in a given EAR file.

Limitations on Container-Managed Persistence Protocol

- Data aliasing problems—If container-managed fields of multiple entity beans map to the same data item in the underlying database, the entity beans may see an inconsistent view of the data item if the multiple entity beans are invoked in the same transaction.
- Eager loading of state—The container loads the entire entity object state into the container-managed fields before invoking the `ejbLoad` method of the abstract bean. This approach may not be optimal for entity objects with large state if most business methods require access to only parts of the state. If this is an issue, use the `<fetched-with>` element for fields that are used infrequently.

Restrictions on Remote Interfaces

The following restrictions apply to the remote interface of an entity bean that uses container-managed persistence:

- Do not expose the `get` and `set` methods for container-managed relationship fields or the persistence Collection classes that are used in container-managed relationships through the remote interface of the bean.

  However, you are free to expose the `get` and `set` methods that correspond to the CMP fields of the entity bean through the bean's remote interface.
- Do not expose local interface types or local home interface types through the remote interface or remote home interface of the bean.

- Do not expose the container-managed collection classes that are used for relationships through the remote interface of the bean.

Dependent value classes can be exposed in the remote interface or remote home interface, and can be included in the client EJB JAR file.

Elements in the sun-cmp-mappings.xml File

"Assembling and Deploying Enterprise Beans," on page 169, provides general information and guidelines on assembling your enterprise beans for deployment. Additional deployment information and instructions are contained in the Sun ONE Application Server *Developer's Guide*.

"Persistence Elements," on page 196 provides information on the information on persistence-related elements in the `sun-ejb-jar.xml` file.

A sample XML file is contained in "Sample Schema Definition," on page 121.

This section describes the elements in the `sun-cmp-mappings.xml` file:

- check-all-at-commit
- check-modified-at-commit
- cmr-field-mapping
- cmr-field-name
- column-name
- column-pair
- consistency
- ejb-name
- entity-mapping
- fetched-with
- field-name
- level
- lock-when-loaded
- lock-when-modified
- named-group
- none

- read-only
- schema
- sun-cmp-mapping
- sun-cmp-mappings
- table-name

check-all-at-commit

This flag is not implemented for Sun ONE Application Server 7.

Subelements

none

check-modified-at-commit

A consistency level flag that indicates to check modified bean instances at commit time.

Subelements

none

cmp-field-mapping

The `cmp-field-mapping` element associates a field with one or more columns that it maps to. The column can be from a bean's primary table or any defined secondary table. If a field is mapped to multiple columns, the column listed first is used as a SOURCE for getting the value from the database. The columns are updated in the order they appear. There is one `cmp-field-mapping` element for each `cmp-field` element defined in the EJB JAR file.

A field can be marked as read-only.

A field may participate in a fetch group if the `fetched-with` element is not specified. The following is assumed:

```
<fetched-with><level>0</level></fetched-with>
```

Subelements

The following table describes subelements for the `cmp-field-mapping` element.

`cmp-field-mapping` Subelements

Subelement	Required	Description
`field-name`	only one	Specifies the Java identifier of a field. This identifier must match the value of the `field-name` subelement of the `cmp-field` that is being mapped. One is required.
`column-name`	one or more	Specifies the name of a column from the primary table, or the table qualified name (TABLE.COLUMN) of a column from a secondary or related table. One is required.
`read-only`	zero or one	Flag that indicates a field is read-only. Optional.
`fetched-with`	zero or one	Specifies the fetch group configuration for fields and relationships. Optional.

cmr-field-mapping

A container-managed relationship field has a name and one or more column pairs that define the relationship. There is one `cmr-field-mapping` element for each `cmr-field`. A relationship can also participate in a fetch group.

If the `fetched-with` element is not present, the following value is assumed: `<fetched-with><none/></fetched-with>`.

Subelements

The following table describes subelements for the `cmr-field-mapping` element.

`cmr-field-mapping` Subelements

Subelement	Required	Description
`cmr-field-name`	only one	Specifies the Java identifier of a field. Must match the value of the `cmr-field-name` subelement of the `cmr-field` that is being mapped.
`column-pair`	one or more	The name of the pair of columns in a database table.
`fetched-with`	zero or one	Specifies the fetch group configuration for fields and relationships. Optional.

cmr-field-name

Specifies the Java identifier of a field. Must match the value of the `cmr-field-name` subelement of the `cmr-field` that is being mapped.

Subelements

none

column-name

Specifies the name of a column from the primary table, or the table qualified name (TABLE.COLUMN) of a column from a secondary or related table. One is required.

Subelements

none

column-pair

The name of the pair of related columns in two database tables. One is required.

Subelements

The following table describes subelements for the `column-pair` element.

`column-pair` Subelements

Subelement	Required	Description
`column-name`	two	Specifies the name of a column from the primary table, or the table qualified name (TABLE.COLUMN) of a column from a secondary or related table.

consistency

Specifies container behavior in guaranteeing transactional consistency of the data in the bean. Optional. If the consistency checking flag element is not present, `none` is assumed.

Subelements

The following table describes the elements used for consistency checking.

Consistency Flags

Flag Element	Description
`check-all-at-commit`	Checks modified instances at commit time.
`check-modified-at-commit`	This flag is not implemented for Sun ONE Application Server 7.
`lock-when-loaded`	An exclusive lock is obtained when the data is loaded.
`lock-when-modified`	This flag is not implemented for Sun ONE Application Server 7.
`none`	No consistency checking occurs.

ejb-name

Specifies the name of the entity bean in the `ejb-jar.xml` file to which the container-managed persistence beans relates. One is required.

Subelements

none

entity-mapping

Specifies the mapping a bean to database columns.

Subelements

The following table describes subelements for the `entity-mapping` element.

`entity-mapping` Subelements

Subelement	Required	Description
`ejb-name`	only one	Specifies the name of the entity bean in the `ejb-jar.xml` file to which the container-managed persistence beans relates. One is required.
`table-name`	only one	Specifies the name of a database table. The table must be present in the database schema file.

`entity-mapping` Subelements *(Continued)*

Subelement	Required	Description
`cmp-field-mapping`	one or more	Associates a field with one or more columns that it maps to. The column can be from a bean's primary table or any defined secondary table. If a field is mapped to multiple columns, the column listed first is used as a SOURCE for getting the value from the database. The columns are updated in the order they appear. There is one `cmp-field-mapping` element for each `cmp-field` element defined in the EJB JAR file. A field can be marked as read-only.
`cmr-field-mapping`	zero or more	A container-managed relationship field has a name and one or more column pairs that define the relationship. There is one `cmr-field-mapping` element for each `cmr-field`. A relationship can also participate in a fetch group.
`secondary-table`	zero or more	Describes the relationship between a bean's primary and secondary table. Column pairs are used to describe this relationship.
`consistency`	zero or one	Specifies container behavior in guaranteeing transactional consistency of the data in the bean. If the consistency checking flag element is not present, `none` is assumed.

fetched-with

Specifies the fetch group configuration for fields and relationships. Optional.

A field may participate in a hierarchical or independent fetch group. If the `fetched-with` element is not present, the following value is assumed: `<fetched-with><none/></fetched-with>`.

Subelements

The following table describes subelements for the `fetched-with` element.

`fetched-with` Subelements

Subelement	Required	Description
`level`	exactly one of these elements is required	Specifies the name of a hierarchical fetch group. The value must be an integer. Fields and relationships that belong to a hierarchical fetch group of equal (or lesser) value are fetched at the same time. The value of `level` must be greater than zero.
`named-group`		Specifies the name of an independent fetch group. All the fields and relationships that are part of a named group are fetched at the same time.
`none`		A consistency level flag that indicates that this field or relationship is fetched by itself.

field-name

Specifies the Java identifier of a field. This identifier must match the value of the `field-name` subelement of the `cmp-field` that is being mapped. One is required.

Subelements

none

level

Specifies a hierarchical fetch group. The value of this element must be an integer. Fields and relationships that belong to a hierarchical fetch group of equal (or lesser) value are fetched at the same time. The value of level must be greater than zero. Only one is allowed.

Subelements

none

lock-when-loaded

A consistency level flag that indicates a lock will be implemented when the data is loaded.

Subelements

none

lock-when-modified

This flag is not implemented for Sun ONE Application Server 7.

Subelements

none

named-group

Specifies the name of an independent fetch group. All the fields and relationships that are part of a named group are fetched at the same time. One is allowed.

Subelements

none

none

A consistency level flag that indicates that this field or relationship is fetched with no other fields or relationships, or it specifies the `fetched-with` semantics.

Subelements

none

read-only

Flag that indicates a field is read-only.

Subelements

none

schema

Specifies the path to the schema file. Only one is required. For further information, refer to "Capturing a Schema," on page 217

Subelements

none

secondary-table

Specifies a bean's secondary table(s).

Subelements

The following table describes subelements for the `secondary-table` element.

`secondary table` Subelements

Subelement	Required	Description
`table-name`	only one	Specifies the name of a database table. The table must be present in the database schema file.
`column-pair`	one or more	The name of the pair of related columns in two database tables.

sun-cmp-mapping

Specifies beans mapped to a particular schema.

NOTE A bean cannot be related to a bean that maps to a different schema, even if the beans are deployed in the same EJB JAR file.

Subelements

The following table describes subelements for the `sun-cmp-mapping` element.

`sun-cmp-mapping` Subelements

Subelement	Required	Description
`schema`	only one	Specifies the path to the schema file.
`entity-mapping`	one or more	Specifies the mapping of beans to database columns.

sun-cmp-mappings

Specifies the collection of subelements for all the beans that will be mapped in an EJB JAR collection.

Subelements

The following table describes subelements for the `sun-cmp-mappings` element.

`sun-cmp-mappings` Subelements

Subelement	Required	Description
`sun-cmp-mapping`	one or more	Specifies beans mapped to a particular schema.

table-name

Specifies the name of a database table. The table must be present in the database schema file. One is required.

Subelements

none

Examples

The following examples are contained in this section:

- Sample Schema Definition
- Sample CMP Mapping XML File
- Sample EJB QL Queries

Sample Schema Definition

```
CREATE TABLE Player
(
   player_Id VARCHAR(255) PRIMARY KEY,
   name VARCHAR(255) ,
   position VARCHAR(255) ,
   salary DOUBLE PRECISION NOT NULL ,
   picture BLOB,
);

CREATE TABLE League
(
   league_Id VARCHAR(255) PRIMARY KEY,
   name VARCHAR(255) ,
   sport VARCHAR(255) ,
);
```

```
CREATE TABLE Team
(
   team_Id VARCHAR(255) PRIMARY KEY,
   city VARCHAR(255) ,
   name VARCHAR(255) ,
   league_Id VARCHAR(255) ,
   FOREIGN KEY (league_Id)   REFERENCES League (league_Id) ,
);

CREATE TABLE TeamPlayer
(
   player_Id VARCHAR(255) ,
   team_Id VARCHAR(255),
   CONSTRAINT pk_TeamPlayer PRIMARY KEY (player_Id , team_Id) ,
   FOREIGN KEY (team_Id)   REFERENCES Team (team_Id),
   FOREIGN KEY (player_Id)   REFERENCES Player (player_Id) ,
);
```

Sample CMP Mapping XML File

For information on these elements, refer to "Elements in the sun-cmp-mappings.xml File," on page 112.

The following sample mapping file would have the name `META-INF/sun-cmp-mappings.xml` in a deployable EJB JAR file:

```
<?xml version="1.0" encoding="UTF-8"?>
<sun-cmp-mappings>
   <sun-cmp-mapping>
      <schema>RosterSchema</schema>
      <entity-mapping>
         <ejb-name>League</ejb-name>
         <table-name>LEAGUE</table-name>
         <cmp-field-mapping>
            <field-name>name</field-name>
            <column-name>LEAGUE.NAME</column-name>
         </cmp-field-mapping>
         <cmp-field-mapping>
            <field-name>leagueId</field-name>
            <column-name>LEAGUE.LEAGUE_ID</column-name>
         </cmp-field-mapping>
         <cmp-field-mapping>
            <field-name>sport</field-name>
            <column-name>LEAGUE.SPORT</column-name>
         </cmp-field-mapping>
         <cmr-field-mapping>
```

```
            <cmr-field-name>team</cmr-field-name>
            <column-pair>
                <column-name>LEAGUE.LEAGUE_ID</column-name>
                <column-name>TEAM.LEAGUE_ID</column-name>
            </column-pair>
        </cmr-field-mapping>
    </entity-mapping>
    <entity-mapping>
        <ejb-name>Team</ejb-name>
        <table-name>TEAM</table-name>
        <cmp-field-mapping>
            <field-name>name</field-name>
            <column-name>TEAM.NAME</column-name>
        </cmp-field-mapping>
        <cmp-field-mapping>
            <field-name>city</field-name>
            <column-name>TEAM.CITY</column-name>
        </cmp-field-mapping>
        <cmp-field-mapping>
            <field-name>teamId</field-name>
            <column-name>TEAM.TEAM_ID</column-name>
        </cmp-field-mapping>
        <cmr-field-mapping>
            <cmr-field-name>playerId</cmr-field-name>
                <column-pair>
                <column-name>TEAM.TEAM_ID</column-name>
                <column-name>TEAMPLAYER.TEAM_ID</column-name>
            </column-pair>
            <column-pair>
                <column-name>TEAMPLAYER.PLAYER_ID</column-name>
                <column-name>PLAYER.PLAYER_ID</column-name>
            </column-pair>
            <fetched-with>
                <none/>
            </fetched-with>
        </cmr-field-mapping>
        <cmr-field-mapping>
            <cmr-field-name>leagueId</cmr-field-name>
            <column-pair>
                <column-name>TEAM.LEAGUE_ID</column-name>
                <column-name>LEAGUE.LEAGUE_ID</column-name>
            </column-pair>
            <fetched-with>
                <none/>
            </fetched-with>
        </cmr-field-mapping>
    </entity-mapping>
```

```
      <entity-mapping>
         <ejb-name>Player</ejb-name>
         <table-name>PLAYER</table-name>
         <cmp-field-mapping>
            <field-name>salary</field-name>
            <column-name>PLAYER.SALARY</column-name>
         </cmp-field-mapping>
         <cmp-field-mapping>
            <field-name>playerId</field-name>
            <column-name>PLAYER.PLAYER_ID</column-name>
         </cmp-field-mapping>
         <cmp-field-mapping>
            <field-name>position</field-name>
            <column-name>PLAYER.POSITION</column-name>
         </cmp-field-mapping>
         <cmp-field-mapping>
            <field-name>name</field-name>
            <column-name>PLAYER.NAME</column-name>
         </cmp-field-mapping>
         <cmr-field-mapping>
            <cmr-field-name>teamId</cmr-field-name>
            <column-pair>
               <column-name>PLAYER.PLAYER_ID</column-name>
               <column-name>TEAMPLAYER.PLAYER_ID</column-name>
            </column-pair>
            <column-pair>
               <column-name>TEAMPLAYER.TEAM_ID</column-name>
               <column-name>TEAM.TEAM_ID</column-name>
            </column-pair>
         </cmr-field-mapping>
      </entity-mapping>
   </sun-cmp-mapping>
</sun-cmp-mappings>
```

Sample EJB QL Queries

```
<query>
   <description></description>
   <query-method>
      <method-name>findAll</method-name>
      <method-params />
   </query-method>
   <ejb-ql>select object(l) from League l</ejb-ql>
</query>
```

```
<query>
   <description></description>
   <query-method>
      <method-name>findByName</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </query-method>
   <ejb-ql>select object(l) from League l where l.name = ?1</ejb-ql>
</query>

<query>
   <description></description>
   <query-method>
      <method-name>findByPosition</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </query-method>
   <ejb-ql>select distinct object(p) from Player p where p.position = ?1</ejb-ql>
</query>

<query>
   <description>Selector returning SET</description>
   <query-method>
      <method-name>ejbSelectTeamsCity</method-name>
      <method-params>
         <method-param>team.LocalLeague</method-param>
      </method-params>
   </query-method>
   <ejb-ql>select distinct t.city from Team t where t.league = ?1</ejb-ql>
</query>

<query>
   <description>Selector returning single object LocalInterface</description>
   <query-method>
      <method-name>ejbSelectTeamByCity</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </query-method>
   <result-type-mapping>Local</result-type-mapping>
   <ejb-ql>select distinct Object(t) from League l, in(l.teams) as t where t.city
= ?1</ejb-ql>
</query>
```

```
<query>
   <description>Selector returning single object String</description>
   <query-method>
      <method-name>ejbSelectTeamsNameByCity</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </query-method>
   <ejb-ql>select distinct t.name from League l, in(l.teams) as t where t.city =
?1</ejb-ql>
</query>

<query>
   <description>Selector returning Set using multiple collection
declarations</description>
   <query-method>
      <method-name>ejbSelectPlayersByLeague</method-name>
      <method-params>
         <method-param>team.LocalLeague</method-param>
      </method-params>
   </query-method>
   <result-type-mapping>Local</result-type-mapping>
   <ejb-ql>select Object(p) from League l, in(l.teams) as t, in(t.players) p
where l = ?1</ejb-ql>
</query>

<query>
   <description>Selector single object int</description>
   <query-method>
      <method-name>ejbSelectSalaryOfPlayerInTeam</method-name>
      <method-params>
         <method-param>team.LocalTeam</method-param>
         <method-param>java.lang.String</method-param>
      </method-params>
   </query-method>
   <ejb-ql>select p.salary from Team t, in(t.players) as p where t = ?1 and p.name
= ?2</ejb-ql>
</query>

<query>
   <description>Finder using the IN Expression</description>
   <query-method>
      <method-name>findByPositionsGoalkeeperOrDefender</method-name>
      <method-params/>
   </query-method>
   <ejb-ql>select object(p) from Player p where p.position IN ('goalkeeper',
'defender')</ejb-ql>
</query>
```

```
<query>
   <description>Finder using the LIKE Expression</description>
   <query-method>
      <method-name>findByNameEndingWithON</method-name>
      <method-params/>
   </query-method>
   <ejb-ql>select object(p) from Player p where p.name LIKE '%on'</ejb-ql>
</query>

<query>
   <description>Finder using the IS NULL Expression</description>
   <query-method>
      <method-name>findByNullName</method-name>
      <method-params/>
   </query-method>
   <ejb-ql>select object(p) from Player p where p.name IS NULL</ejb-ql>
</query>

<query>
   <description>Finder using the MEMBER OF Expression</description>
   <query-method>
      <method-name>findByTeam</method-name>
      <method-params>
         <method-param>team.LocalTeam</method-param>
      </method-params>
   </query-method>
   <ejb-ql>select object(p) from Player p where ?1 MEMBER p.teams</ejb-ql>
</query>

<query>
   <description>Finder using the ABS function</description>
   <query-method>
      <method-name>findBySalarayWithArithmeticFunctionABS</method-name>
      <method-params>
         <method-param>double</method-param>
      </method-params>
   </query-method>
   <ejb-ql>select object(p) from Player p where p.salary = ABS(?1)</ejb-ql>
</query>

<query>
   <description>Finder using the SQRT function</description>
   <query-method>
      <method-name>findBySalarayWithArithmeticFunctionSQRT</method-name>
      <method-params>
         <method-param>double</method-param>
      </method-params>
   </query-method>
   <ejb-ql>select object(p) from Player p where p.salary = SQRT(?1)</ejb-ql>
</query>
```

Chapter 5

Using Message-Driven Beans

This section describes message-driven beans and explains the requirements for creating them in the Sun ONE Application Server 7 environment.

NOTE If you are unfamiliar with message-driven beans or the EJB technology, refer to the Java Software tutorials:

`http://java.sun.com/j2ee/docs.html`

Extensive information on message-driven beans is contained in chapters 15 and 16 of the Enterprise JavaBeans Specification, v2.0.

Overview material on the Sun ONE Application Server is contained in "Introducing the Sun ONE Application Server Enterprise JavaBeans Technology," on page 19 and the *Sun ONE Application Server Product Introduction*.

This section contains the following topics:

- About Message-Driven Beans
- Developing Message-Driven Beans
- Restrictions and Optimizations
- Sample Message-Driven Bean XML Files

About Message-Driven Beans

A message-driven bean is an enterprise bean that allows J2EE applications to process messages asynchronously. It acts as message listener, which is similar to an event listener except that it receives messages instead of events. The messages may be sent by any J2EE component—an application client, another enterprise bean, or a web component—or by an application or system that does not use J2EE technology.

The following topics are addressed in this section:

- Message-Driven Beans Differences
- Message-Driven Bean Characteristics
- Transaction Management
- Concurrent Message Processing

Message-Driven Beans Differences

Session beans and entity beans allow you to send JMS messages and to receive them synchronously, but not asynchronously. To avoid tying up server resources, you may prefer to use asynchronous receives in a server-side component. To receive messages asynchronously, use a message-driven bean.

The most visible difference between message-driven beans and session and entity beans is that clients do not access message-driven beans through interfaces. Unlike a session or entity bean, a message-driven bean has only a bean class.

In several respects, a message-driven bean resembles a stateless session bean:

- A message-driven bean's instances retain no data or conversational state for a specific client.
- All instances of a message-driven bean are equal, allowing the container to pool these message-driven bean instances. This allows streams of messages to be processed concurrently.
- A single message-driven bean can process messages from multiple clients.

The instance variables of the message-driven bean instance can contain some state across the handling of client messages—for example, a JMS connection, an open database connection, or an object reference to an EJB object.

Message-Driven Bean Characteristics

A message-driven bean instance is an instance of a message-driven bean class. It has neither a home nor a remote interface; message-driven beans are anonymous. They have no client-visible identity.

A client accesses a message-driven bean through JMS by sending messages to the message destination for which the message-driven bean class is the `MessageListener`. A message-driven bean's Queue and Topic are assigned during deployment using the Sun ONE Application Server resources.

Message-driven beans have the following characteristics:

- Execute upon receipt of a single client message.
- Are asynchronously invoked.
- Are relatively short lived.
- Do not represent directly shared data in the database, but may access and update this data.
- Can be transaction-aware.
- Are stateless.

Transaction Management

Both container-managed and bean-managed transactions as defined in the Enterprise JavaBeans Specification, v2.0 are supported.

With container-managed transactions, a message may be delivered to a message-driven bean within a transaction context, so that all operations within the `onMessage` method are part of a single transaction. If message processing is rolled back, the message will be redelivered.

Refer to “Handling Transactions with Enterprise Beans,” on page 143 for additional information on transactions.

Concurrent Message Processing

A container allows many instances of a message-driven bean class to be running concurrently, thus allowing for the concurrent processing of a stream of messages. No guarantees are made as to the exact order in which messages are delivered to the instances of the message-driven bean class, although the container attempts to deliver messages in chronological order when this does not impair the concurrency of message processing.

Message-driven beans should, therefore, be prepared to handle messages that are out of sequence. For example, a message to cancel a reservation may be delivered before the message to make the reservation.

Developing Message-Driven Beans

The goal of the message-driven bean model is to make developing an enterprise bean that is asynchronously invoked to handle incoming messages as simple as developing the same functionality in any other JMS listener. A further goal is to allow for concurrent processing of a stream of messages by means of container-provided pooling of message-driven bean instances.

The following sections provide guidelines on creating message-driven beans:

- Creating the Bean Class Definition
- Configuration

Creating the Bean Class Definition

Unlike session and entity beans, message-driven beans do not have the remote or local interfaces that define client access. Client components do not locate message-driven beans and invoke methods directly on them.

Although message-driven beans do not have business methods, they may contain helper methods that are invoked internally by the `onMessage` method.

For message-driven beans, the class requirements are:

- The class must implement, directly or indirectly, the `javax.ejb.MessageDrivenBean` interface.
- The class must implement, directly or indirectly, the `javax.ejb.MessageListener` interface.

- The class must be defined as public and must *not* be defined as abstract or final.
- The class must have a public constructor that takes no arguments (used by the container to create instances of the message-driven bean class).
- The class must not define the `finalize` method.
- The class must implement the `onMessage` method.
- The class must implement one `ejbCreate` method, with no arguments.
- The class must implement one `ejbRemove` method with no arguments.

The following sections address the various methods in a message-driven bean’s class definition.

- Using ejbCreate
- Using setMessageDrivenContext
- Using onMessage
- Using ejbRemove

Using ejbCreate

The message-driven bean class defines one `ejbCreate` method whose signature must follow these rules:

- The method name must be `ejbCreate`.
- The method must be declared as `public` and must *not* be declared as `final` or `static`.
- The return type must be `void`.
- The method must have no arguments.
- The `throws` clause must not define any application exceptions.

Using setMessageDrivenContext

The container provides the message-driven bean instance with a `MesssageDrivenContext`. This gives the message-driven bean instance access to the instance's context maintained by the container.

Using onMessage

The `onMessage` method has a single argument: the incoming message.

The `onMessage` method is called by the bean's container when a message has arrived for the bean to service. This method contains the business logic that handles the processing of the message. It is the message-driven bean's responsibility to parse the message and perform the necessary business logic.

The message-driven bean class defines one `onMessage` method whose signature must follow these rules:

- The method must be declared as `public` and must *not* be declared as `final` or `static`.
- The return type must be `void`.
- The method must have a single argument of type `javax.jms.Message`.
- The `throws` clause must not define any application exceptions. Refer to "onMessage Runtime Exception," on page 139 for semantics on throwing an exception from `onMessage`.

The `onMessage` method is invoked in the scope of a transaction that is determined by the transaction attribute specified in the deployment descriptor.

NOTE If the bean is specified as using container-managed transaction demarcation, either the `Required` or `NotSupport` transaction attribute must be specified in its deployment descriptor.

Using ejbRemove

The message-driven bean class defines one `ejbRemove` method to free a bean when it is no longer needed. The signature must follow these rules:

- The method name must be `ejbRemove`.
- The method must be declared as `public` and must not be declared as `final` or `static`.
- The return type must be `void`.
- The method must have no arguments.
- The `throws` clause must not define any application exceptions.

NOTE You cannot assume that the container will always invoke the `ejbRemove` method on a message-driven bean instance.

The `ejbRemove` method is not called if the EJB container crashes, or if an exception is thrown from the instance's `onMessage` method to the container. If the message-driven bean instance allocates resources in the `ejbCreate` method, and/or the `onMessage` method, and releases the resources in the `ejbRemove` method, these resources will not be automatically released. Your application should provide a mechanism to periodically clean up the unreleased resources.

Configuration

This section addresses the following configuration topics:

- Connection Factory and Destination
- Message-Driven Bean Pool
- Server instance-wide Attributes
- Automatic Reconnection to JMS Provider

Connection Factory and Destination

A message-driven bean is a JMS client. Therefore, the message-driven bean container uses the JMS service integrated into the Sun ONE Application Server. JMS clients use JMS Connection Factory- and Destination-administered objects. A JMS Connection Factory administered object is a resource manager Connection Factory object that is used to create connections to the JMS provider.

The `mdb-connection-factory` element in the `sun-ejb-jar.xml`file for a message-driven bean can be used to specify the connection factory used by the container to create the container connection to the JMS provider. This element can be used to work with a third-party JMS provider.

If the `mdb-connection-factory` element is not specified, a default one created at server startup is used. This provides connection to the built-in Sun ONE Message Queue broker on the port that is specified in the `jms-service` element (if enabled) in the `server.xml` file, using the default user name/password (resource principal) of the Sun ONE Message Queue. Refer to the *Sun ONE Message Queue Developer's Guide* for more information.

The `jndi-name` element of the `ejb` element in `sun-ejb-jar.xml` file specifies the JNDI name of the administered object for the JMS Queue or Topic destination that is associated with the message-driven bean.

Message-Driven Bean Pool

The container manages a pool of message-driven beans for the concurrent processing of a stream of messages. The Sun ONE Application Server-specific bean deployment descriptor contains the elements that define the pool (that is, the `bean-pool` element):

- `steady-pool-size`
- `resize-quantity`
- `max-pool-size`
- `pool-idle-timeout-in-seconds`

For information on these elements, refer to "Pooling and Caching Elements," on page 203.

Server instance-wide Attributes

An administrator can control the following server instance-wide message-driven bean attributes for the `mdb-container` element in the `server.xml` file:

- `steady-pool-size`
- `pool-resize-quantity`
- `max-pool-size`
- `idle-timeout-in-seconds`
- `log-level`
- `monitoring-enabled`

For further explanation on these attributes, refer to "Pooling and Caching Elements," on page 203 and the Sun ONE Application Server *Administrator's Configuration File Reference*.

For information on monitoring message-driven beans, see the Sun ONE Application Server Administration interface online help and *Administrator's Guide*.

NOTE Running monitoring when it is not need may impact performance, so you may choose to turn monitoring off using the asadmin command or the Administration interface when it is not in use.

Automatic Reconnection to JMS Provider

When the Sun ONE Application Server is started, for each deployed message-driven bean, its container keeps a connection to the JMS provider. When the connection is broken, the container is not able to receive messages from the JMS provider and, therefore, is unable to deliver messages to its message-driven bean instances. When the auto reconnection feature is enabled, the container automatically tries to reconnect to the JMS provider if the connection is broken.

The `mdb-container` element in the `server.xml` file contains auto reconnection properties. By default, `reconnect-enabled` is set to true and `reconnect-delay-in-seconds` is set to 60 seconds. That is, there is a delay of 60 seconds before each attempt to reconnect, and `reconnect-max-retries` is set to 60.

The container logs messages for each reconnect attempt.

NOTE Depending on where the message processing stage is, if the connection is broken, the `onMessage` method may not be able to complete successfully, or the transaction may be rolled back due to a JMS exception. When the container reestablishes connection to the JMS provider, JMS message redelivery semantics apply.

Refer to the *Sun ONE Application Server Administrator's Configuration File Reference* for information on auto reconnect properties of the `mdb-container` element in the `server.xml` file.

Restrictions and Optimizations

This section discusses the following restrictions and performance optimizations that you should be aware of in developing message-driven beans:

- JMS Limitation
- Pool Tuning and Monitoring
- onMessage Runtime Exception

JMS Limitation

The Sun ONE Application Server supports JMS messaging through a built-in JMS service provided by Sun ONE Message Queue 3.0.1, Platform Edition. As a standalone product, Sun ONE Message Queue 3.0.1 supports the JMS 1.1 specification. However, Sun ONE Application Server 7 supports the J2EE 1.3 specification, which encompasses only the more limited JMS 1.02b specification. For this reason, the additional features embodied in JMS 1.1 are not available to applications running on the Sun ONE Application Server 7.

Developers of JMS messaging applications should, therefore, limit JMS client components that run in a Sun ONE Application Server environment to JMS 1.02b. For more information, see the *Sun ONE Message Queue Developer's Guide* or *Release Notes*.

Pool Tuning and Monitoring

The message-driven bean pool is also a pool of threads, with each message-driven bean instance in the pool associating with a server session, and each server session associating with a thread. Therefore, a large pool size also means a high number of threads, which will impact performance and server resources.

When configuring message-driven bean pool properties, you must consider factors such as message arrival rate and pattern, `onMessage` method processing time, overall server resources (threads, memory, and so on), and any concurrency requirements and limitations from other resources that the message-driven bean may access.

Performance and resource usage tuning should also consider potential JMS provider properties for the connection factory that is used by the container (`mdb-connection-factory` element in deployment descriptor). For example, the Sun ONE Message Queue flow control related properties for connection factory should be tuned in situations where the message incoming rate is much higher than `max-pool-size` can handle.

Refer to the *Sun ONE Application Server Administrator's Guide* for information on how to get message-driven bean pool statistics.

onMessage Runtime Exception

Message-driven beans, like other well-behaved JMS MessageListeners, should not, in general, throw runtime exceptions. If a message-driven bean's `onMessage` method encounters a system-level exception or error that does not allow the method to successfully complete, the Enterprise JavaBeans Specification, v2.0 provides the following guidelines:

- If the bean method encounters a runtime exception or error, it should simply propagate the error from the bean method to the container.
- If the bean method performs an operation that results in a checked exception that the bean method cannot recover, the bean method should throw the `javax.ejb.EJBException` that wraps the original exception.
- Any other unexpected error conditions should be reported using `javax.ejb.EJBException` (`javax.ejb.EJBException` is a subclass of `java.lang.RuntimeException`).

Under container-managed transaction demarcation, upon receiving a runtime exception from a message-driven bean's `onMessage` method, the container will roll back the container-started transaction and JMS message will be redelivered. This is because the message delivery itself is part of the container-started transaction. By default, the Sun ONE Application Server container closes the container's connection to the JMS provider when the first runtime exception is received from a message-driven bean instance's `onMessage` method. This avoids potential message redelivery looping and protects server resources if the message-driven bean's `onMessage` method continues misbehaving. This default container behavior can be changed using the `cmt-max-runtime-exceptions` property of the `mdb-container` element in the `server.xml` file.

The `cmt-max-runtime-exceptions` property specifies the maximum number of runtime exceptions allowed from a message-driven bean's `onMessage` method before the container starts to close the container's connection to the JMS provider. By default this value is 1; -1 disables this container protection.

A message-driven bean's `onMessage` method can use the `javax.jms.Message getJMSRedelivered` method to check whether a received message is a redelivered message.

NOTE The `cmt-max-runtime-exceptions` property may be deprecated in the future.

Sample Message-Driven Bean XML Files

This section includes the following sample files:

- Sample ejb-jar.xml File
- Sample sun-ejb-jar.xml File

For information on the elements associated with message-driven beans, refer to "Elements in the sun-ejb-jar.xml File," on page 176 and the *Sun ONE Application Server Developer's Guide*.

Sample ejb-jar.xml File

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

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
   <enterprise-beans>
      <message-driven>
         <ejb-name>MessageBean</ejb-name>
         <ejb-class>samples.mdb.ejb.MessageBean</ejb-class>
         <transaction-type>Container</transaction-type>
         <message-driven-destination>
            <destination-type>javax.jms.Queue</destination-type>
         </message-driven-destination>
         <resource-ref>
            <res-ref-name>jms/QueueConnectionFactory</res-ref-name>
            <res-type>javax.jms.QueueConnectionFactory</res-type>
            <res-auth>Container</res-auth>
         </resource-ref>
      </message-driven>
   </enterprise-beans>
      <assembly-descriptor>
         <container-transaction>
            <method>
               <ejb-name>MessageBean</ejb-name>
               <method-intf>Bean</method-intf>
               <method-name>onMessage</method-name>
               <method-params>
                  <method-param>javax.jms.Message</method-param>
               </method-params>
            </method>
            <trans-attribute>NotSupported</trans-attribute>
        </container-transaction>
    </assembly-descriptor
</ejb-jar>
```

Sample sun-ejb-jar.xml File

For information on these elements, refer to "Elements in the sun-ejb-jar.xml File," on page 176

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

<!DOCTYPE sun-ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Sun ONE Application
Server 7.0 EJB 2.0//EN'
'http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2_0-0.dtd'>

<sun-ejb-jar>
    <enterprise-beans>
        <ejb>
            <ejb-name>MessageBean</ejb-name>
            <jndi-name>jms/sample/Queue</jndi-name>
            <resource-ref>
                <res-ref-name>jms/QueueConnectionFactory</res-ref-name>
                <jndi-name>jms/sample/QueueConnectionFactory</jndi-name>
                <default-resource-principal>
                    <name>guest</name>
                    <password>guest</password>
                </default-resource-principal>
            </resource-ref>
            <mdb-connection-factory>
                <jndi-name>jms/sample/QueueConnectionFactory</jndi-name>
                <default-resource-principal>
                    <name>guest</name>
                    <password>guest</password>
                </default-resource-principal>
            </mdb-connection-factory>
        </ejb>
    </enterprise-beans>
</sun-ejb-jar>
```

Chapter 6

Handling Transactions with Enterprise Beans

This section describes the transaction support built into the Enterprise JavaBeans (EJBs) programming model for Sun ONE Application Server 7.

NOTE If you are unfamiliar with transaction handling in the EJB technology, refer to the Java Software tutorials:

`http://java.sun.com/j2ee/docs.html`

Extensive information on EJB transaction support is contained in Chapter 17, “Support for Transactions,” of the Enterprise JavaBeans Specification, v2.0.

Overview material on the Sun ONE Application Server is contained in “Introducing the Sun ONE Application Server Enterprise JavaBeans Technology,” on page 19 and the *Sun ONE Application Server Product Introduction*.

This section addresses the following topics:

- JTA and JTS Transaction Support
- Using Container-Managed Transactions
- Using Bean-Managed Transactions
- Setting Transaction Timeouts
- Handling Isolation Levels

JTA and JTS Transaction Support

J2EE includes support for distributed transactions through two specifications:

- Java Transaction API (JTA)
- Java Transaction Service (JTS)

The JTA is a high-level, implementation-independent protocol API that allows applications and application servers to access transactions.

JTS specifies the implementation of a transaction manager which supports the JTA and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the level below the API. JTS propagates transactions using the Internet Inter-ORB Protocol (IIOP).

The current transaction manager implementation supports JTS and the JTA. The EJB container itself uses the Java Transaction API interface to interact with JTS.

The J2EE transaction manager controls all EJB transactions, except for bean-managed Java Database Connectivity (JBDC) transactions, and allows an enterprise bean to update multiple databases within a transaction.

About Transaction Handling

As a developer, you can write an application that updates data in multiple databases which may be distributed across multiple sites. The site may use EJB servers from different vendors.

This section provides overview information on the following topics:

- Flat Transactions
- Global and Local Transactions
- Demarcation Models
- Commit Options
- Administration and Monitoring

Flat Transactions

The Enterprise JavaBeans Specification, v2.0 requires support for flat (as opposed to nested) transactions. In a flat transaction, each transaction is decoupled from and independent of other transactions in the system. You cannot start another transaction in the same thread until the current transaction ends.

Flat transactions are the most prevalent model and are supported by most commercial database systems. Although nested transactions offer a finer granularity of control over transactions, they are supported by far fewer commercial database systems.

Global and Local Transactions

Understanding the distinction between global and local transactions is crucial in understanding the Sun ONE Application Server support for transactions.

- Global transactions—Transactions that are managed and coordinated by a resource manager, and can span multiple databases and processes. The resource manager typically uses the XA two-phase commit protocol to interact with the Enterprise Information System (EIS) or database.

- Local transactions—Transactions that are native to a single EIS or database and are restricted within a single process. Local transactions do not involve multiple data sources.

Both local and global transactions are demarcated using the `javax.transaction.UserTransaction` interface, which the client must use. Local transactions bypass the transaction manager and are faster.

Initially, all transactions are local. If a non-XA data source connection is the first resource connection enlisted in a transaction scope, it will become a global transaction when a (second) XA data source connection joins it. If a second non-XA data source connection attempts to join, an exception is thrown.

The Sun ONE Application Server operates in *either* global or local transaction mode, but the two modes cannot be mixed in the same transaction.

NOTE	If your application uses global transactions, you must configure and enable the corresponding Sun ONE Application Server resource managers. For more information, see the Sun ONE Application Server Administration interface online help and the and *Administrator's Guide*.

Demarcation Models

As a developer, you can choose between using programmatic transaction demarcation in the EJB code (bean-managed) or declarative demarcation (container-managed). Regardless of whether an enterprise bean uses bean-managed or container-managed transaction demarcation, the burden of implementing transaction management is on the EJB container and the Sun ONE Application Server. The container and the server implement the necessary low-level transaction protocols, such as the two-phase commit protocol between a transaction manager and a dustbowls system or Sun ONE Message Queue provider, transaction context propagation, and distributed two-phase commit.

These demarcation models are addressed in the following sections:

- Container-Managed Transactions
- Bean-Managed Transactions

Container-Managed Transactions

One primary advantage of enterprise beans is the support they provide for container-managed transactions, also known as declarative transactions. In an enterprise bean with container-managed transactions, the EJB container sets the boundaries of the transactions.

NOTE You can use container-managed transactions with any type of enterprise bean (session, entity, or message-driven), but an entity bean can *only* use container-managed transactions.

Container-managed transactions simplify development because the EJB code does not explicitly mark the transaction's boundaries. That is, the code does not include statements that begin and end the transaction. The container is responsible for:

- Demarcating and transparently propagating the transactional context
- In conjunction with a transaction manager, ensuring that all participants in the transaction see a consistent outcome

Bean-Managed Transactions

The EJB specification supports bean-managed transaction demarcation, also known as programmer-demarcated transactions, using `javax.transaction.UserTransaction`. With bean-managed transactions, you must perform a Java Naming and Directory Interface (JNDI) lookup to obtain a `UserTransaction` object.

NOTE You can use bean-managed transactions with session or message-driven beans, but an entity bean must use container-managed transactions.

There are two types of bean-managed transactions:

- JDBC type—You delimit JDBC transactions with the commit and rollback methods of the connection interface.
- JTA type—You invoke the begin, commit, and rollback methods of the UserTransaction interface to demarcate JTA transactions.

Commit Options

The EBJ protocol is designed to give the container the flexibility to select the disposition of the instance state at the time a transaction is committed. This allows the container to best manage caching an entity object’s state and associating an entity object identity with the EJB instances.

There are three commit-time options:

- Option A—The container caches a ready instance between transactions. The container ensures that the instance has exclusive access to the state of the object in persistent storage.

  In this case, the container does *not* have to synchronize the instance’s state from the persistent storage at the beginning of the next transaction.

NOTE Commit option A is not supported for Sun ONE Application Server 7.

- Option B—The container caches a ready instance between transactions, but the container does *not* ensure that the instance has exclusive access to the state of the object in persistent storage. This is the default.

  In this case, the container must synchronize the instance's state by invoking `ejbLoad` from persistent storage at the beginning of the next transaction.

- Option C—The container does *not* cache a ready instance between transactions, but instead returns the instance to the pool of available instances after a transaction has completed.

  The life cycle for every business method invocation under commit option C looks like this:

  ```
  ejbActivate->
     ejbLoad ->
        business method ->
           ejbStore ->
              ejbPassivate
  ```

  If there is more than one transactional client concurrently accessing the same entity `EJBObject`, the first client gets the ready instance and subsequent concurrent clients get new instances from the pool.

The Sun ONE Application Server deployment descriptor has an element, `commit-option`, that specifies the commit option to be used. Based on the specified commit option, the appropriate handler is instantiated.

NOTE It is assumed that if commit option A is used, the developer is responsible for ensuring that only this application is updating the database. In other words, this is not the container's responsibility.

Administration and Monitoring

An administrator can control the following instance-wide transaction service attributes for the `transaction-service` element in the `server.xml` file:

- `automatic-recovery`
- `timeout-in-seconds`
- `tx-log-directory`
- `heuristic-decision`
- `keypoint-interval`

- `log-level`
- `monitoring-enabled`

For further explanation on these attributes, refer to the *Sun ONE Application Server Administrator's Configuration File Reference*.

In addition, the administrator can monitor transactions using statistics from the transaction manager that provide information on such activities as the number of transactions completed / rolled back / recovered since server startup, and transactions presently being processed.

For information on administering and monitoring transactions, see the Sun ONE Application Server Administration interface online help and the *Sun ONE Application Server Administrator's Guide*.

Using Container-Managed Transactions

Typically, the container begins a transaction immediately before an EJB method starts, and commits the transaction just before the method exits. Each method can be associated with a single transaction.

NOTE Nested or multiple transactions are not allowed within a method.

Container-managed transactions do not require all methods to be associated with transactions. When deploying an enterprise bean, you specify which of the bean's methods are associated with transactions by setting the transaction attributes.

Although beans with container-managed transactions require less coding, they have one limitation:

> When a method is executing, it can only be associated with either a single transaction or no transaction at all.

If this limitation will make coding your bean difficult, bean-managed transactions may be your best choice.

When a commit occurs, the transaction signals the container that the bean has completed its useful work and tells the container to synchronize its state with the underlying data source. The container permits the transaction to complete and then frees the bean. Result sets associated with a committed transaction are no longer valid. Subsequent requests for the same bean cause the container to issue a load to synchronize state with the underlying data source.

NOTE Transactions initiated by the container are implicitly committed.

Any participant can roll back a transaction.

The following sections are related to developing enterprise beans with container-managed transactions:

- Specifying Transaction Attributes
- Rolling Back a Container-Managed Transaction
- Synchronizing a Session Bean's Instance Variables
- Methods Not Allowed in Container-Managed Transactions

Specifying Transaction Attributes

A *transaction attribute* is a parameter that controls the scope of a transaction.

Because transaction attributes are stored in the deployment descriptor, they can be changed during several phases of J2EE application development: at EJB creation, at assembly (packaging), or at deployment. However, as an EJB developer, it is your responsibility to specify the attributes when creating the EJB. The attributes should be modified only when you (or whoever is assembling) are assembling components into larger applications.

NOTE Do not expect the person who is deploying the J2EE application to specify the transaction attributes.

You can specify the transaction attributes for the entire enterprise bean or for individual methods. If you've specified one attribute for a method and another for the bean, the attribute for the method takes precedence.

TIP If you're unsure about how to set up transactions in the EJB's deployment descriptor, specify container-managed transactions. Then, set the `Required` transaction attribute for the entire enterprise bean. This approach will work most of the time.

For more information, on the EJB deployment descriptor file, refer to “Creating Deployment Descriptors,” on page 170.

This section addresses the following topics:

- Differing Attribute Requirements
- Attribute Values

Differing Attribute Requirements

When specifying attributes for individual methods, the requirements differ with the type of bean.

- Session beans—Need the attributes defined for business methods, but do not allow them for the create methods.
- Entity beans—Require transaction attributes for the business, create, remove, and finder methods.
- Message-driven beans—Require transaction attributes (either `Required` or `NotSupported`) for the `onMessage` method.

Attribute Values

A transaction attribute may have one of the following values:

- Required
- RequiresNew
- Mandatory
- NotSupported
- Supports
- Never

Required

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.

TIP The `Required` attribute will work for most transactions. Therefore, you may want to use it as a default, at least in the early phases of development. Because transaction attributes are declarative, you can easily change them at a later time.

RequiresNew

If the client is running within a transaction and invokes the EJB's method, the container takes the following steps:

1. Suspends the client's transaction.
2. Starts a new transaction.
3. Delegates the call to the method.
4. Resumes the client's transaction after the method completes.

If the client is not associated with a transaction, the container starts a new transaction before running the method.

You should use the `RequiresNew` attribute when you want to ensure that the method always runs within a new transaction.

Mandatory

If the client is running within a transaction and invokes the EJB's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container throws a `TransactionRequiredException`.

Use the `Mandatory` attribute if the EJB's method must use the transaction of the client.

NotSupported

If the client is running within a transaction and invokes the EJB's method, the container suspends the client's transaction before invoking the method. After the method has completed, the container resumes the client's transaction.

If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Supports

If the client is running within a transaction and invokes the EJB's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

NOTE Because the transactional behavior of the method may vary, you should use the `Supports` attribute with caution.

Never

If the client is running within a transaction and invokes the enterprise bean's method, the container throws a `RemoteException`. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Use the `NotSupported` attribute for methods that don't need transactions. Because transactions involve overhead, this attribute may improve performance.

The following table summarizes the effects of the transaction attributes. The left column lists the transaction attribute, the middle column lists the type of client transaction, and the right column lists the transaction type of the business method. Transactions can be T1, T2, or None. (Both T1 and T2 transactions are controlled by the container.)

- T1 transaction—Is associated with the client that calls a method in the enterprise bean. In most cases, the client is another enterprise bean.
- T2 transaction—Is started by the container, just before the method executes.
- None—In the third column, the word None means that the business method does not execute within a transaction controlled by the container. However, the database calls in such a business method might be controlled by the transaction manager of the database.

Transaction Attributes and Scope

Transaction Attribute	Client's Transaction	Business Method's Transaction
`Required`	None	T2
	T1	T1
`RequiresNew`	None	T2
	T1	T2
`Mandatory`	None	Error
	T1	T1
`NotSupported`	None	None
	T1	None

Transaction Attributes and Scope *(Continued)*

Transaction Attribute	Client's Transaction	Business Method's Transaction
`Supports`	None	None
	T1	T1
`Never`	None	None
	Ti	Error

Rolling Back a Container-Managed Transaction

There are two ways to roll back a container-managed transaction:

- First, if a system exception is thrown, the container automatically rolls back the transaction.
- Second, by invoking the `setRollbackOnly` method of the `EJBContext` interface, the bean method instructs the container to roll back the transaction. If the bean throws an application exception, the rollback is not automatic, but may be initiated by a call to `setRollbackOnly`.

When the container rolls back a transaction, it always undoes the changes to data made by SQL calls within the transaction. However, only in entity beans will the container undo changes made to instance variables. (It does so by automatically invoking the entity bean's `ejbLoad` method, which loads the instance variables from the database.)

A session bean must explicitly reset any instance variables changed within the transaction when a rollback occurs. The easiest way to reset a session bean's instance variables is by implementing the `SessionSynchronization` interface.

Synchronizing a Session Bean's Instance Variables

The `SessionSynchronization` interface, which is optional in session beans, allows you to synchronize the instance variables with their corresponding values in the database. The container invokes the `SessionSynchronization` methods—`afterBegin`, `beforeCompletion`, and `afterCompletion`—at each of the main stages of a transaction.

- `afterBegin` method—Informs the instance that a new transaction has begun. The container invokes `afterBegin` immediately before it invokes the business method. The `afterBegin` method is a good place to load the instance variables from the database.
- `beforeCompletion` method—The container invokes `beforeCompletion` method after the business method has finished, but just before the transaction commits. The `beforeCompletion` method is the last opportunity for the session bean to roll back the transaction (by calling `setRollbackOnly`).

  If it hasn't already updated the database with the values of the instance variables, the session bean may do so in the `beforeCompletion` method.
- `afterCompletion` method—Indicates that the transaction has completed. It has a single boolean parameter, whose value is true if the transaction was committed, and false if it was rolled back.

  If a rollback occurred, the session bean can refresh its instance variables from the database in the `afterCompletion` method.

Methods Not Allowed in Container-Managed Transactions

For container-managed transactions, you should not invoke any method that might interfere with the transaction boundaries set by the container. Prohibited methods are:

- The `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection`
- The `getUserTransaction` method of `javax.ejb.EJBContext`
- Any method of `javax.transaction.UserTransaction`

You may, however, use these methods to set boundaries in bean-managed transactions.

Using Bean-Managed Transactions

In a bean-managed transaction, the code in the session or message-driven bean explicitly marks the boundaries of the transaction. By moving transaction management to the bean level, you gain the ability to place all the bean's activities—even those not directly tied to database access—under the same transaction control as your database calls. This guarantees that all application parts controlled by a bean run as part of the same transaction.

In a failure situation, either everything the bean undertakes is committed, or everything is rolled back.

The following sections are related to developing enterprise beans with bean-managed transactions:

- Choosing the Type of Transactions
- Returning Without Committing
- Methods Not Allowed in Bean-Managed Transactions

Choosing the Type of Transactions

When coding a bean-managed transaction for session or message-driven beans, you must decide whether to use JDBC or JTA transactions.

NOTE In a session bean with bean-managed transactions, it is possible to mix JDBC and JTA transactions. This practice is not recommended, however, because it could make your code difficult to debug and maintain.

The following sections discuss both types of transactions:

- JDBC Transactions
- JTA Transactions

JDBC Transactions

JDBC transaction is controlled by the transaction manager of the database. You may want to use JDBC transactions when wrapping legacy code inside a session bean.

To code a JDBC transaction, you invoke the commit and rollback methods of the `java.sql.Connection` interface. The beginning of a transaction is implicit. A transaction begins with the first SQL statement that follows the most recent commit, rollback, or connect statement. (This rule is generally true, but may vary with database vendor.)

For additional information on JDBC, refer to the *Sun ONE Application Server Developer's Guide to J2EE Features and Services*.

JTA Transactions

JTA allows you to demarcate transactions in a manner that is independent of the transaction manager implementation. The J2EE SDK implements the transaction manager with the JTS. But your code doesn't call the JTS methods directly. Instead, it invokes the JTA methods, which then call the lower-level JTS routines.

A JTA transaction is controlled by the J2EE transaction manager. You may want to use a JTA transaction because it can span updates to multiple databases from different vendors. A particular database's transaction manager may not work with heterogeneous databases.

The J2EE transaction manager does have one limitation—it does not support nested transactions. In other words, it cannot start a transaction for an instance until the previous transaction has ended.

For additional information on the JTA, refer to the *Sun ONE Application Server Developer's Guide to J2EE Features and Services*.

Returning Without Committing

A stateless session bean with bean-managed transactions that has begun a transaction in a business method must commit or roll back a transaction before returning. However, a stateful session bean does not have this restriction. In a stateful session bean with a JTA transaction—The association between the bean instance and the transaction is retained across multiple client calls.

Methods Not Allowed in Bean-Managed Transactions

For bean-managed transactions, do not invoke the `getRollbackOnly` and `setRollbackOnly` methods of the `EJBContext` interface. These methods should be used only in container-managed transactions.

NOTE For bean-managed transactions, invoke the `getStatus` and `rollback` methods of the `UserTransaction` interface.

Setting Transaction Timeouts

For container-managed transactions, you control the transaction timeout interval by setting the value of the `timeout-in-seconds` property in the `server.xml` file. For example, you would set the timeout value to 5 seconds as follows:

```
timeout-in-seconds=5
```

With this setting, if the transaction has not completed within 5 seconds, the EJB container rolls the transaction back.

NOTE Only enterprise beans using container-managed transactions are affected by the `timeout-in-seconds` property. For enterprise beans using bean-managed JTA transactions, you invoke the `setTransactionTimeout` method of the `UserTransaction` interface.

Handling Isolation Levels

Transactions not only ensure the full completion (or rollback) of the statements that they enclose, but also isolate the data modified by the statements. The *isolation level* describes the degree to which the data being updated is visible to other transactions.

If the transaction allows other programs to read uncommitted data, performance may improve because the other programs don't have to wait until the transaction ends. But this may also cause a problem—if the transaction subsequently rolls back, another program might read the wrong data.

For entity beans with bean-managed persistence and for all session beans, you can set the isolation level programmatically with the API provided by the underlying database. A database, for example, might allow you to permit uncommitted reads by invoking the `setTransactionIsolation` method.

For entity beans that use container-managed persistence, you can use the `consistency` element in the `sun-cmp-mapping.xml` file to set the isolation level.

CAUTION Do not change the isolation level in the middle of a transaction. Usually, such a change causes the database software to issue an implicit commit. Because the isolation levels offered by database vendors may vary, you should check the database documentation for more information. Isolation levels are not standardized for the J2EE platform.

Chapter 7

Developing Secure Enterprise Beans

This section describes how security management works in the EJB architecture and provides guidelines for developing secure enterprise beans for the Sun ONE Application Server 7 environment.

NOTE	If you are unfamiliar with the EJB technology, refer to the Java Software tutorials: `http://java.sun.com/j2ee/docs.html` Extensive information on EJB security is contained in Chapter 21, "Security Management," of the Enterprise JavaBeans Specification, v2.0. General information on application security is contained in the *Sun ONE Application Server Developer's Guide*.

This section addresses the following topics:

- About Secure Enterprise Beans
- Defining Security Roles
- Declaring Method Permissions
- Declaring Security Role References
- Specifying Security Identities
- Using Programmatic Security
- Handling Unprotected EJB-Tier Resources

General information on application security is contained in the *Sun ONE Application Server Developer's Guide*.

About Secure Enterprise Beans

Your main role as an EJB developer is to declare the security requirements of your applications in such a way that these requirements can be satisfied during application deployment. In most cases, the EJB's business methods should not contain any security-related logic.

The following topics are addressed in this section:

- Authorization and Authentication
- Security Roles
- Deployment

Authorization and Authentication

Authorization provides controlled access to protected resources; it is based on identification and authentication. *Identification* is the process that enables recognition of an entity by a system. A*uthentication* is the process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system.

Enterprise beans can be configured to permit access only to users with the appropriate authorization level. This is done by using the Sun ONE Application Server Administration interface to generate the deployment descriptor for the application EAR and EJB JAR files.

Security Roles

A *security role* is an application-specific logical grouping of users, classified by common trait, such as a customer profile or job title. When an application is deployed, roles are mapped to security identities, such as *principals* (identities assigned to users as a result of authentication) or groups, in the operational environment. Based on this, a user with a certain security role has associated access rights to an enterprise bean. The link is the actual name of the security role that is being referenced.

A group also represents a category of users, but its scope is different from the scope of a role.

- A *role* is a J2EE application-specific abstraction.

- A *group* is a set of environment-specific users from the current realm. Group membership is determined by the underlying realm implementation.

NOTE When defining method restrictions and role mappings, it is a common error to confuse realm groups and J2EE application roles. Such confusion can lead to unintended access consequences or inoperable application configurations. For information on realms, refer to the *Sun ONE Application Server Developer's Guide*.

Deployment

The *security role reference* defines a mapping between the name of a role that is called from an ENTERPRISE BEAN using `isCallerInRole` (String name) and the name of a security role that has been defined for the application. This security role reference allows an enterprise bean to reference an existing security role.

When an application is deployed, the deployer maps the roles to the security identities that exist in the operational environment. When you are developing enterprise beans, you should know the roles of your users, but you probably won't know exactly who the users will be. That's taken care of in the J2EE security architecture. After your component has been deployed, the system administrator maps the roles to the J2EE users (or groups) of the default realm (usually the file realm).

Defining Security Roles

To create a role for a J2EE application, you declare it for the EJB JAR file or for the WAR file that is contained in the application. The security roles defined by the `security-role` elements are scoped to the EJB JAR file level and apply to all enterprise beans in the EJB JAR files.

Example

The following example of a security role definition in a deployment descriptor specifies two role-name elements, `employee` and `admin`.

```
...
<assembly-descriptor>
   <security-role>
      <description>
         This role includes the employees of the enterprise who
```

```
            are allowed to access the employee self service
            application. This role is allowed to access only
            her/his information
         </desciption>
         <role-name>employee<role-name>
         </security-role>
         <security-role>
            <description>
               This role should be assigned to the personnel
               authorized to perform administrative functions
               for the employee self service application. This
               role does not have direct access to
               sensitive employee and payroll information
            </desciption>
         <role-name>admin<role-name>
         </security-role>
...
</assembly-descriptor>
```

Declaring Method Permissions

Method permissions indicate which roles are allowed to invoke which methods. The application assembler declares the method permission relationships in the deployment descriptor using the method permission elements as follows:

- Each `method-permission` element includes a list of one or more security roles and a list of one or more methods.

  All listed security roles are allowed to invoke all listed methods. Each security role in the list is identified by the `role-name` element, and each method (or set of methods, as described below) is identified by the method element. An optional description can be associated with a `method-permission` element using the description element.

- The method permissions relationship is defined as the union of all method permissions defined in the individual method permission elements.

- A security role or a method may appear in multiple `method-permission` elements.

Example

The following deployment descriptor example illustrates how security roles are assigned method permissions in the deployment descriptor. These are converted into security elements at deployment.

```
...
<method-permission>
   <role-name>employee</role-name>
      <method>
      <ejb-name>EmployeeService</ejb-name>
      <method-name>*</method-name>
   </method>
</method-permission>


<method-permission>
   <role-name>employee</role-name>
   <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>findByPrimaryKey</method-name>
   </method>
   <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>getEmployeeInfo</method-name>
   </method>
   <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>updateEmployeeInfo</method-name>
   </method
</method-permission>
...
```

Declaring Security Role References

As the EJB developer, you are responsible for declaring all security role names used in the enterprise bean in the `security-role-ref` elements of the deployment descriptor for roles which are used programmatically from within the respective enterprise beans.

- The application assembler is responsible for linking all security role references declared in the `security-role-ref` elements to the security roles defined in the `security-role` elements.
- The application assembler links each security role reference to a security role using the `role-link` element.

NOTE The `role-link` element value must be one of the security role names defined in a `security-role` element.

Example

The following deployment descriptor example shows how to link the security role reference named `payroll` to the security role named `payroll-department`.

```
<enterprise-beans>
   ...
   <entity>
      <ejb-name>AardvarkPayroll</ejb-name>
      <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
   ...
      <security-role-ref>
      <description> This role should be assigned to the payroll
      department's employees. Members of this role have access to
      anyone's payroll record. The role has been linked to the
      payroll-department role.
      </description>
      <role-name>payroll</role-name>
      <role-link>payroll-department</role-link>
      </security-role-ref>
   ....
   </entity>
   ...
</enterprise-beans>
```

This role should be assigned to the payroll department's employees. Members of this role have access to anyone's payroll record. The role has been linked to the `payroll-department` role.

Further information on security roles can be found in the *Sun ONE Application Server Developer's Guide*. More information on EJB access control configuration can be found in the Enterprise JavaBeans Specification, v2.0.

Specifying Security Identities

Optionally, the EJB assembler can specify whether the caller's identity should be used for executing the EJB methods or whether a specific run-as identity should be used. The `security-identity` element in the deployment descriptor is used for this purpose. The value of the `security-identity` element is `use-caller-identity` or `run-as`.

Unless specified, the caller identity is used by default.

The run-as Identity

The *run-as identity* establishes the identity the enterprise bean will use when it makes calls. It does not affect the identities of its callers, which are the identities tested for permission to access the methods of the enterprise bean.

The EJB assembler can use the `run-as` element to define a run-as identity for an enterprise bean in the deployment descriptor. The run-as identity applies to the enterprise bean as a whole, that is, to all methods of the EJB's home and component interface, or to the `onMessage` method of a message-driven bean, and all internal methods of the enterprise bean that might, in turn, be called.

Because the assembler does not generally know the security environment of the operational environment, the run-as identity is designated by a *logical* role-name which corresponds to one of the security roles defined in the deployment descriptor. The deployer must then assign a security principal (defined in the operational environment) to be used as the principal for the run-as identity. The security principal should be a principal that has been assigned to the security role as specified by the `role-name` element.

Using Programmatic Security

In general, security management should be enforced by the container in a manner that is transparent to the EJB's business methods.

NOTE Enterprise beans can use programmatic login just as servlets do. For more information, see the Sun ONE Application Server *Developer's Guide*.

Programmatic security in the EJB tier consists of the `getCallerPrincipal` and the `isCallerInRole` methods. You can use the `getCallerPrincipal` method to determine the caller of the enterprise bean, and the `isCallerInRole` method to determine the caller's role.

The `getCallerPrincipal` method of the `EJBContext` interface returns the `java.security.Principal` object that identifies the caller of the enterprise bean. (In this case, a principal is the same as a user.) In the following example, the `getUser` method of an enterprise bean returns the name of the J2EE user that invoked it:

```
public String getUser()
{
   return context.getCallerPrincipal().getName();
}
```

You can determine whether an EJB's caller belongs to a particular role by invoking the `isCallerInRole` method:

```
boolean result = context.isCallerInRole("Customer");
```

For details on how to implement programmatic security, refer to Chapter 21, Security Management," of the Enterprise JavaBeans Specification, v2.0.

Handling Unprotected EJB-Tier Resources

All users have the anonymous role. By default, the value of the anonymous role is ANYONE, which is configurable in the `server.xml` file. So, if a method permission specifies that the role required is ANYONE (or whatever the anonymous role is set to), then any user can access this method.

NOTE If a method permission covering a method does not exist, the method is accessible to all.

If a method permission exists, it is always enforced. For example, if a method permission is set so that the `updateEmployeeInfo` method can only be accessed by the `employee` role, then it is never possible to access this method without role `employee`. If the `employee` role is not mapped to any user or group, no one will be able to invoke the `updateEmployeeInfo` method.

Chapter 8

Assembling and Deploying Enterprise Beans

This section describes how enterprise beans are assembled and deployed in the Sun ONE Application Server 7 environment and provides information on the elements and subelements used to create the EJB XML files.

NOTE For general assembly and deployment information, see the Sun ONE Application Server *Developer's Guide*.You should already be familiar with that deployment material before proceeding with EJB assembly.

This section contains the following topics:

- EJB Structure
- Creating Deployment Descriptors
- Deploying Enterprise Beans
- The sun-ejb-jar_2_0-0.dtd File Structure
- Elements in the sun-ejb-jar.xml File
- Sample EJB XML Files

An alphabetical list of all EJB-related elements is contained in "Elements Listings," on page 229.

EJB Structure

The EJB Java ARchive (JAR) file is the standard format for assembling enterprise beans. This file contains the bean classes (home, remote, local, and implementation), all the utility classes, and the deployment descriptors (`ejb-jar.xml` and `sun-ejb-jar.xml`).

An EJB JAR file produced by a developer contains one or more enterprise beans and typically does not contain assembly instructions; an EJB JAR file produced by an assembler contains one or more enterprise beans plus application assembly instructions describing how the enterprise beans are combined into a single application deployment unit.

An EJB JAR file can stand alone without being part of an Enterprise ARchive (EAR) file, or be part of an EAR file.

Sample application files are located in *install_root*`/samples/j2ee/`.

Creating Deployment Descriptors

A J2EE module is a collection of one or more J2EE components of the same container type with two deployment descriptors of that type. One descriptor is J2EE standard, the other is specific to Sun ONE Application Server. For enterprise beans, two deployment descriptor files apply:

- `ejb-jar.xml`

  A J2EE standard file, described in the Enterprise JavaBeans Specification, v2.0.

- `sun-ejb-jar.xml`

  A Sun ONE Application Server-specific file described in this chapter.

- `sun-cmp-mappings.xml`

  A Sun ONE Application Server-specific file used if the deployed bean uses container-managed persistence.

NOTE For information on the XML file associated with container-managed persistence, refer to “Elements in the sun-cmp-mappings.xml File,” on page 112.”

The easiest way to create the deployment descriptor files is to deploy an EJB module using the Administration interface or Sun ONE Studio 4 IDE. For more information, see the *Sun ONE Application Server Developer's Guide*. For example EJB XML files, see "Sample EJB XML Files," on page 213.

After you have created these files, you can edit them using the Administration interface or a combination of an editor and command line utilities such as Ant to reassemble and redeploy the updated deployment descriptor information.

NOTE You can create the deployment descriptor manually if you prefer.

The J2EE standard deployment descriptors are described in the 1.3 J2EE Specification. For more information on EJB deployment descriptors, see Chapter 22 in the Enterprise JavaBeans Specification, v2.0. Our sample applications develop some ANT targets that help in assembly and deployment. Refer to the ANT information in the *Sun ONE Application Server Developer's Guide.*

Deploying Enterprise Beans

When you deploy, undeploy, or redeploy a enterprise bean, you do not need to restart the server.

NOTE Stubs and skeletons are generated during deployment. You can retrieve the client JAR file with the stubs and skeletons for use with a rich client.

This section addresses the following topics:

- Using the Administration Interface
- Using the Command-Line Interface
- Using the Sun ONE Studio 4 IDE
- Reloading Enterprise Beans

Using the Administration Interface

To deploy an EJB application using the Administration interface:

1. Open the Applications component under your server instance.
2. Go to the EJB Modules page.
3. Click Deploy.
4. Enter the full path to the JAR module (or click Browse to find it), then click OK.

Using the Command-Line Interface

To deploy an enterprise bean using the command line:

1. Edit the deployment descriptor files (`ejb-jar.xml` and `sun-ejb-jar.xml`) by hand.
2. Execute an Ant build command (such as `build jar`) to reassemble the JAR module.
3. Use the `asadmin deploy` command to deploy the JAR module. The syntax is as follows:

   ```
   asadmin deploy -type ejb [-name component-name] [-force=true]
   [-upload=true] -instance instancename filepath
   ```

   For example, the following command deploys an EJB application as a stand-alone module:

   ```
   asadmin deploy -type ejb -instance inst1 myEJB.jar
   ```

Using the Sun ONE Studio 4 IDE

You can use Sun ONE Studio 4 IDE, bundled with Sun ONE Application Server, to assemble and deploy enterprise beans. For information about using Sun ONE Studio 4, see the Sun ONE Studio 4, Enterprise Edition tutorial.

NOTE In Sun ONE Studio 4, deploying a web application is referred to as *executing* it.

Reloading Enterprise Beans

If you make code changes to an enterprise bean and dynamic reloading is enabled, you do not need to redeploy the enterprise bean or restart the server. You can simply drop the changed files into the application's deployed directory (such as, *instance-dir*/`applications`) and the changes will be picked up.

To enable dynamic reloading with the Administration interface:

1. In the Administration interface, select your server instance
2. Select Applications.

   The Application Properties page is displayed.
3. Check the Reload Enabled box to enable dynamic reloading.
4. Enter a number of seconds in the Reload Poll Interval field to set the interval at which applications and modules are checked for code changes and dynamically reloaded.
5. Click Save.

For details, see the *Sun ONE Application Server Administrator's Guide*.

In addition, to load new servlet files, reload EJB related changes, or reload deployment descriptor changes, you must do the following:

1. Create an empty file named `.reload` at the root of the deployed application:

   instance_dir`/applications/j2ee-apps/`*app_name*`/.reload`

   or individually deployed module:

   instance_dir`/applications/j2ee-modules/`*module_name*`/.reload`
2. Explicitly update the `.reload` file's timestamp (`touch .reload` in UNIX) each time you make changes to the bean or deployment descriptor.

   The reload monitor thread periodically looks at the timestamp of the `.reload` files to detect any changes. This interval is, by default, two seconds and can be modified by changing the value of `dynamic-reload-poll-interval-in-seconds` in the `server.xml` file.

You can deploy an EJB application in a number of ways:

- Using the Command-Line Interface
- Using the Administration Interface
- Using the Sun ONE Studio 4 IDE

For more detailed information about deployment, see the *Sun ONE Application Server Developer's Guide*.

The sun-ejb-jar_2_0-0.dtd File Structure

The `sun-ejb-jar_2_0-0.dtd` file defines the structure of the `sun-ejb-jar.xml` file, including the elements it can contain and the subelements and attributes these elements can have. The `sub-ejb-jar_2_0-0.dtd` file is located in the *install-dir*`/lib/dtds` directory.

NOTE Do not edit the `sun-ejb-jar_2_0-0.dtd` file; its contents change only with new versions of the Sun ONE Application Server.

For general information about DTD files and XML, see the XML specification at:

`http://www.w3.org/TR/REC-xml`

Each element defined in a DTD file (which may be present in the corresponding XML file) can contain the following:

- Subelements
- Data
- Attributes

An alphabetical list of all EJB-related elements is contained in "Elements Listings," on page 229.

Subelements

Elements can contain subelements. For example, the following file fragment defines the `cmp-resource` element:

```
<!ELEMENT cmp-resource (jndi-name, default-resource-principal?)>
```

This `ELEMENT` tag specifies that a resource element called `cmp-resource` can contain subelements called `jndi-name` and `default-resource-principal`, with the question mark (?) indicating that there can be zero or one of the `default-resource-principal` subelement.

Each subelement can be suffixed with an optional character to determine the number of times it can occur.

The following table shows how optional suffix characters of subelements determine the requirement rules, or number of allowed occurrences, for the subelements. The left column lists the subelement ending character, the right column list the corresponding requirement rule.

Requirement Rules for Subelement Suffixes

Suffix	**Number of Occurrences**
*element**	Can contain *zero or more* of this subelement.
?*element*	Can contain *zero or one* of this subelement.
element+	Must contain *one or more* of this subelement.
element (no suffix)	Must contain *only one* of this subelement.

If an element cannot contain other elements, you see `EMPTY` or `(#PCDATA)` instead of a list of element names in parentheses.

Data

Some elements contain character data instead of subelements. These elements have definitions of the following format:

`<!ELEMENT` *element-name* `(#PCDATA)>`

For example:

```
<!ELEMENT description (#PCDATA)>
```

In the Sun ONE Application Server XML files, white space is treated as part of the data in a data element. Therefore, there should be no extra white space before or after the data delimited by a data element. For example:

```
<description>class name of session manager</description>
```

```
<password>secret</password>
```

Attributes

Elements can contain attributes (name, value pairs). Attributes are defined in attributes lists using the ATTLIST tag.

None of the elements in the `sun-ejb-jar.xml` file contain attributes.

Elements in the sun-ejb-jar.xml File

An alphabetical list of all EJB-related elements is contained in "Elements Listings," on page 229.

NOTE For information on the DTD and XML file associated with container-managed persistence mapping, refer to "Elements in the sun-cmp-mappings.xml File," on page 112."

This section describes the XML elements in the `sun-ejb-jar_2_0-0.dtd` file. For your convenience, the elements are grouped as follows:

- General Elements
- Role Mapping Elements
- Reference Elements
- Security Elements
- Persistence Elements
- Pooling and Caching Elements
- Class Elements

NOTE If any configuration for an enterprise bean is not specified in the `sun-ejb-jar.xml` file, it can default to a corresponding value in the `ejb-container` element of the `server.xml` file if an equivalency exists. You can change the default values in the `server.xml` file; these changes will be reflected in any enterprise bean that does not have that value defined.

General Elements

General elements are as follows:

- ejb
- ejb-name
- enterprise-beans

- is-read-only-bean
- refresh-period-in-seconds
- sun-ejb-jar
- unique-id

ejb

Defines runtime properties for a single enterprise bean within the application. The subelements listed below apply to particular enterprise beans as follows:

- All types of beans—`ejb-name`, `ejb-ref`, `resource-ref`, `resource-env-ref`, `cmp`, `ior-security-config`, `gen-classes`, `jndi-name`
- Stateless session beans and message-driven beans—`bean-pool`
- Stateful session beans and entity beans—`bean-cache`
- Entity beans (BMP)—`is-read-only-bean`, `refresh-period-in-seconds`, `commit-option`, `bean-cache`
- Message-driven bean—`mdb-connection-factory`, `jms-durable-subscription-name`, `jms-max-messages-load`, `bean-pool`

Subelements

The following table describes subelements for the `ejb` element.

`ejb` Subelements

Subelement	Required	Description
`ejb-name`	only one	Matches the display name of the bean to which it refers.
`jndi-name`	zero or more	Specifies the absolute `jndi-name`. In the case of message-driven beans, this is the JNDI name of the Java Message Service Queue or Topic destination resource object associated with the message-driven bean class. Whether it is Queue or Topic type depends on the destination type in the message-driven deployment descriptor message-driven-destination. If no message-driven-destination deployment descriptor is specified, this defaults to Queue type.

`ejb` Subelements *(Continued)*

Subelement	Required	Description
`ejb-ref`	zero or more	Maps the absolute JNDI name to the `ejb-ref` element in the corresponding J2EE XML file.
`resource-ref`	zero or more	Maps the absolute JNDI name to the `resource-ref` in the corresponding J2EE XML file.
`resource-env-ref`	zero or more	Maps the absolute JNDI name to the `resource-env-ref` in the corresponding J2EE XML file.
`pass-by-reference`	zero or one	When a servlet or EJB calls another bean that is co-located within the same process, the Sun ONE Application Server does not automatically perform marshalling of all call parameters.
`cmp`	zero or one	Specifies runtime information for a container-managed persistence (CMP) EntityBean object for EJB1.1 and EJB2.0 beans. This is a pointer to a file that describes the mapping information of a bean.
`principal`	zero or one	Specifies the principal (user) name in an enterprise bean that has the `run-as` role specified.
`mdb-connection-factory`	zero or one	Specifies the connection factory associated with a message-driven bean.
`jms-durable-subscription-name`	zero or one	Contains data that specifies the durable subscription associated with a message-driven bean.
`jms-max-messages-load`	zero or one	Specifies the maximum number of messages to load into a Java Message Service session at one time for a message-driven bean to serve. The default is 1.
`ior-security-config`	zero or one	Specifies the security information for the IOR.
`is-read-only-bean`	zero or one	Flag specifying this bean is a read-only bean.

ejb Subelements *(Continued)*

Subelement	Required	Description
refresh-period-in-seconds	zero or one	Specifies the rate at which a read-only-bean must be refreshed from the data source. If this is less than or equal to zero, the bean is never refreshed; if greater than zero, the bean instances are refreshed at the specified interval. This rate is just a hint to the container. Default is 600.
commit-option	zero or one	Contains data that has valid values of A, B, or C. Default value is B.
gen-classes	zero or one	Specifies all the generated class names for a bean.
bean-pool	zero or one bean-pool	Specifies the bean pool properties. Used for stateless session beans, entity beans, and message-driven bean pools.
bean-cache	zero or one bean-pool	Specifies the bean cache properties. Used only for stateful session beans and entity beans

Example

```
<ejb>
       ejb-name>CustomerEJB</ejb-name>
       <jndi-name>customer</jndi-name>
       <resource-ref>
              <res-ref-name>jdbc/SimpleBank</res-ref-name>
              <jndi-name>jdbc/PointBase</jndi-name>
       </resource-ref>
       <is-read-only-bean>false</is-read-only-bean>
       <commit-option>B</commit-option>
       <bean-pool>
              <steady-pool-size>10</steady-pool-size>
              <resize-quantity>10</resize-quantity>
              <max-pool-size>100</max-pool-size>
              <pool-idle-timeout-in-seconds>
              600
              </pool-idle-timeout-in-seconds>
       </bean-pool>
       <bean-cache>
              <max-cache-size>100</max-cache-size>
              <resize-quantity>10</resize-quantity>
```

```
        <removal-timeout-in-seconds>3600</removal-timeout-in-seconds>
              <victim-selection-policy>LRU</victim-selection-policy>
        </bean-cache>
</ejb>
```

ejb-name

Matches the display name of the enterprise bean to which it refers. This name is assigned by the EJB JAR file producer to name the enterprise bean in the EJB JAR file's deployment descriptor. The name must be unique among the names of the enterprise beans in the same EJB JAR file.

There is no architected relationship between the `ejb-name` in the deployment descriptor and the JNDI name that the deployer will assign to the EJB's home.

Subelements

none

Example

```
<ejb-name>EmployeeService</ejb-name>
```

enterprise-beans

Specifies all the runtime properties for an EJB JAR file in the application.

Subelements

The following table describes subelements for the `enterprise-bean` element.

`enterprise-beans` Subelements

Subelement	**Required**	**Description**
`name`	zero or one	Specifies the name string.
`unique-id`	zero or one	Specifies a unique system identifier. This data is automatically generated and updated at deployment/redeployment.
`ejb`	zero or more	Defines runtime properties for a single enterprise bean within the application.
`pm-descriptors`	zero or one	Describes the persistence manager descriptors. One of them must be in use at a given time. This basically applies to Sun ONE Application Server pluggable persistence manager APIs.

`enterprise-beans` Subelements *(Continued)*

Subelement	Required	Description
`cmp-resource`	zero or one	Specifies the database to be used for storing container-managed persistence (CMP) beans in an EJB JAR file.

Example

```
<enterprise-beans>
   <ejb>
      ejb-name>CustomerEJB</ejb-name>
      <jndi-name>customer</jndi-name>
      <resource-ref>
         <res-ref-name>jdbc/SimpleBank</res-ref-name>
‘        <jndi-name>jdbc/PointBase</jndi-name>
      </resource-ref>
      <is-read-only-bean>false</is-read-only-bean>
      <commit-option>B</commit-option>
      <bean-pool>
         <steady-pool-size>10</steady-pool-size>
         <resize-quantity>10</resize-quantity>
         <max-pool-size>100</max-pool-size>
         <pool-idle-timeout-in-seconds>
         600
         </pool-idle-timeout-in-seconds>
      </bean-pool>
      <bean-cache>
         <max-cache-size>100</max-cache-size>
         <resize-quantity>10</resize-quantity>
         <removal-timeout-in-seconds>3600</removal-timeout-in-seconds>
         <victim-selection-policy>LRU</victim-selection-policy>
      </bean-cache>
   </ejb>
</enterprise-beans
```

is-read-only-bean

A flag specifying that this bean is a read-only bean.

Subelements

none

Example

```
<is-read-only-bean>false</is-read-only-bean>
```

refresh-period-in-seconds

Specifies the rate at which a read-only-bean must be refreshed from the data source. If the value is less than or equal to zero, the bean is never refreshed; if the value is greater than zero, the bean instances are refreshed at specified intervals. This rate is just a hint to the container. Default is 600.

Subelements

none

sun-ejb-jar

Defines the Sun ONE Application Server-specific configuration for an EJB JAR file in the application. This is the root element; there can only be one `sun-ejb-jar` element in an `sun-ejb-jar.xml` file.

Refer to “Sample sun-ejb-jar.xml File,” on page 214 for example of this file.

Subelements

The following table describes subelements for the `sun-ejb-jar` element.

`sun-ejb-jar` Subelements

Subelement	Required	Description
`security-role-mapping`	zero or more	Maps a role in the corresponding J2EE XML file to a user or group.
`enterprise-beans`	only one	Describes all the runtime properties for an EJB JAR file in the application.

unique-id

Specifies a unique system identifier. This data is automatically generated and updated at deployment/redeployment. Developers should not change these values after deployment.

Subelements

none

Role Mapping Elements

The role mapping element maps a role, as specified in the EJB JAR `role-name` entries, to a environment-specific user or group. If it maps to a user, it must be a concrete user which exists in the current realm who can log into the server using the current authentication method. If it maps to a group, the realm must support groups and it must be a concrete group which exists in the current realm. To be useful, there must be at least one user in that realm who belongs to that group.

Role mapping elements are as follows:

- group-name
- principal
- principal-name
- role-name
- security-role-mapping
- server-name

group-name

Specifies the group name.

Subelements

none

principal

Defines a node that specifies a user name on the platform.

Subelements

The following table describes subelements for the `principal` element.

`principal` Subelements

Subelement	Required	Description
`name`	only one	Specifies the name of the user.

principal-name

Specifies the principal (user) name in an enterprise bean that has the `run-as` role specified.

Subelements

none

role-name

Specifies the `role-name` in the `security-role` element of the `ejb-jar.xml` file.

Subelements

none

Example

```
<role-name>employee</role-name>
```

security-role-mapping

Maps roles to users and groups.

Subelements

The following table describes subelements for the `security-role-mapping` element.

`security-role-mapping` Subelements

Subelement	Required	Description
`role-name`	only one	Specifies the `role-name` from the `ejb-jar.xml` file being mapped.
`principal-name`	requires at least one `principal-name` or `group-name`	Specifies the principal (user) name in a bean that has the run-as role specified.
`group-name`	requires at least one `principal-name` or `group-name`	Specifies the group name.

server-name

Specifies the name of the server where the application is being deployed.

Subelements

none

Reference Elements

Reference elements are as follows:

- ejb-ref
- ejb-ref-name
- jndi-name
- pass-by-reference
- res-ref-name
- resource-env-ref
- resource-env-ref-name
- resource-ref

ejb-ref

Maps the absolute `jndi-name` name to the `ejb-ref` element in the corresponding J2EE XML file. The `ejb-ref` element is used for the declaration of a reference to an EJB's home.

Applies to session beans or entity beans.

Subelements

The following table describes subelements for the `ejb-ref` element.

`ejb-ref` Subelements

Subelement	Required	Description
`ejb-ref-name`	only one	Specifies the `ejb-ref-name` in the corresponding J2EE EJB JAR file `ejb-ref` entry.
`jndi-name`	only one	Specifies the absolute `jndi-name`.

ejb-ref-name

Specifies the `ejb-ref-name` in the corresponding J2EE XML file `ejb-ref` entry. The name must be unique within the enterprise bean. It is recommended that the name be prefixed with `ejb/`.

Subelements

none

Example

```
<ejb-ref-name>ejb/Payroll</ejb-ref-name>
```

jndi-name

Specifies the absolute `jndi-name`.

Applies to all enterprise beans.

Subelements

none

Example

```
<jndi-name>jdbc/PointBase</jndi-name>
```

pass-by-reference

Specifies the passing method used by a servlet or enterprise bean calling a remote interface method in another bean that is co-located within the same process. Default is false.

NOTE The `pass-by-reference` flag only applies for method calls to remote interfaces. As defined in the Enterprise JavaBeans Specification, v2.0, calls to local interfaces use pass by reference semantics.

- If false (the default if this element is not present), this application uses pass-by-value semantics, which the Enterprise JavaBeans Specification, v2.0 requires.
- If true, this application uses pass-by-reference semantics.

When a servlet or enterprise bean calls a remote interface method in another bean that is co-located within the same process, by default the Sun ONE Application Server makes copies of all the call parameters in order to preserve the pass-by-value semantics. This increases the call overhead and decreases performance.

However, if the calling method does not mutate the object being passed as a parameter, it is safe to pass the object itself without making a copy of it. To do this, set the pass-by-reference value to true.

To apply pass-by-reference semantics to an entire J2EE application containing multiple EJB modules, you can set the same element in the `sun-application.xml` file. If you want to use pass-by-reference in both the bean and application level, the bean level takes precedence over the application level.

For information on the `server.xml` file, see the *Sun ONE Application Server Developer's Guide* and *Administrator's Configuration File Reference*.

Subelements

none

res-ref-name

Specifies the `res-ref-name` in the corresponding J2EE `ejb-jar.xml` file `resource-ref` entry. The `res-ref-name` element specifies the name of a resource manager connection factory reference. The name is a JNDI name relative to the java:comp/env context. The name must be unique within an enterprise bean.

Subelements

none

Example

```
<res-ref-name>jdbc/SimpleBank</res-ref-name>
```

resource-env-ref

Maps the `resource-env-ref-name` in the corresponding J2EE `ejb-jar.xml` file `resource-env-ref` entry to an absolute `jndi-name` in the `resources` element in the `server.xml` file. The `resource-env-ref` element contains a declaration of an enterprise bean's reference to an administered object associated with a resource in the bean's environment.

Used in entity, message-driven, and session beans.

Subelements

The following table describes subelements for the `resource-env-ref` element.

`resource-env-ref` Subelements

Subelement	Required	Description
`resource-env-ref-name`	only one	Specifies the `resource-env-ref-name` in the corresponding J2EE `ejb-jar.xml` file `resource-env-ref` entry.
`jndi-name`	only one	Specifies the absolute `jndi-name`.

Example

```
<resource-env-ref>
   <resource-env-ref-name>
   jms/StockQueueName
   </resource-env-ref-name>
   <jndi-name>jms/StockQueue</jndi-name>
</resource-env-ref>
```

resource-env-ref-name

Specifies the `resource-ref-name` in the corresponding J2EE `ejb-jar.xml` file `resource-env-ref` entry. The `resource-env-ref-name` element specifies the name of a resource environment reference; its value is the environment entry name used in the EJB code. The name is a JNDI name relative to the java:comp/env context and must be unique within an enterprise bean.

Subelements

none

Example

```
<resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
```

resource-ref

Maps the `res-ref-name` in the corresponding J2EE `ejb-jar.xml` file `resource-ref` entry to the absolute `jndi-name` in the `resources` element in the `server.XML` file. The `resource-ref` element contains a declaration of an EJB's reference to an external resource.

NOTE Connections acquired from JMS connection factories are not shareable in the current release of the Sun ONE Application Server. The `res-sharing-scope` element in the `ejb-jar.xml` file `resource-ref` element is ignored for JMS connection factories.

Used in entity, message-driven, and session beans.

NOTE When `resource-ref` specifies a JMS connection factory for the Sun ONE Message Queue, the `default-resource-principal` (name/password) must exist in the Sun ONE Message Queue user repository. Refer to the Security Management chapter in the *Sun ONE Message Queue Administrator's Guide* for information on how to manage the Sun ONE Message Queue user repository.

Subelements

The following table describes subelements for the `resource-ref` element.

`resource-ref` Subelements

Subelement	Required	Description
`res-ref-name`	only one	Specifies the `res-ref-name` in the corresponding J2EE `ejb-jar.xml` file `resource-ref` entry.
`jndi-name`	only one	Specifies the absolute `jndi-name`.
`default-resource -principal`	zero or one	Specifies the default sign-on (name/password) to the resource manager.

Example

```
<resource-ref>
   <res-ref-name>jdbc/EmployeeDBName</res-ref-name>
   <jndi-name>jdbc/EmployeeDB</jndi-name>
</resource-ref>
```

Messaging Elements

This section contains the following elements associated with messaging:

- jms-durable-subscription-name
- jms-max-messages-load
- mdb-connection-factory

jms-durable-subscription-name

Specifies the durable subscription associated with a message-driven bean class. Only applies to the Java Message Service Topic Destination type, and only when the message-driven bean deployment descriptor subscription durability is Durable.

Subelements

none

jms-max-messages-load

Specifies the maximum number of messages to load into a Java Message Service session at one time for a message-driven bean to serve. The default is 1.

Subelements

none

mdb-connection-factory

Specifies the connection factory associated with a message-driven bean. Queue or Topic type must be consistent with the Java Message Service Destination type associated with the message-driven bean class.

Subelements

The following table describes subelements for the `mdb-connection-factory` element.

`mdb-connection-factory` Subelements

Subelement	Required	Description
`jndi-name`	only one	Specifies the absolute `jndi-name`.
`default-resource-principal`	zero or one	Specifies the default sign-on (name/password) to the resource manager.

Security Elements

This section describes the elements that are associated with authentication, authorization, and general security. The following elements are included:

- as-context
- auth-method
- caller-propagation
- confidentiality
- default-resource-principal
- establish-trust-in-client
- establish-trust-in-target
- integrity
- ior-security-config
- name
- password
- realm
- required
- sas-context
- transport-config

as-context

Specifies the authentication mechanism that will be used to authenticate the client. If specified, it will be `USERNAME_PASSWORD`.

Subelements

The following table describes subelements for the `as-context` element.

`as-context` Subelements

Subelement	Required	Description
`auth-method`	only one	Specifies the authentication method. The only supported value is `USERNAME_PASSWORD`.
`realm`	only one	Specifies the realm in which the user is authenticated.
`required`	only one	Specifies if the authentication method specified is required to be used for client authentication. If so, the EstablishTrustInClient bit will be set in the target_requires field of as-context. The value is either true or false.

auth-method

Specifies the authentication method. The only supported value is `USERNAME_PASSWORD`.

Subelements

none

caller-propagation

Specifies if the target will accept propagated caller identities. The values are NONE, SUPPORTED, or REQUIRED.

Subelements

none

confidentiality

Specifies if the target supports privacy-protected messages. The values are NONE, SUPPORTED, or REQUIRED.

Subelements

none

default-resource-principal

Specifies the default sign-on (name / password) to the resource manager.

Subelements

The following table describes subelements for the `default-resource-principal` element.

`default-resource-principal` Subelements

Subelement	Required	Description
`name`	only one	Specifies the default resource principal name used to sign on to a resource manager.
`password`	only on	Specifies password of the default resource principal.

establish-trust-in-client

Specifies if the target is capable of authenticating a client. The values are NONE, SUPPORTED, or REQUIRED.

Subelements

none

establish-trust-in-target

Specifies if the target is capable of authenticating *to* a client. The values are NONE, SUPPORTED, or REQUIRED.

Subelements

none

integrity

Specifies if the target supports integrity-protected messages. The values are NONE, SUPPORTED, or REQUIRED.

Subelements

none

ior-security-config

Specifies the security information for the input-output redirection (IOR).

Subelements

The following table describes subelements for the `ior-security-config` element.

`ior-security-config` Subelements

Subelement	Required	Description
`transport-config`	zero or one	Specifies the security information for transport.
`as-context`	zero or one	Describes the authentication mechanism that will be used to authenticate the client. If specified, it will be `USERNAME_PASSWORD`.
`sas-context`	zero or one	Describes the sas-context fields.

name

Specifies an identity.

Subelements

none

password

Specifies the password that security needs to complete authentication.

Subelements

none

realm

Specifies the realm in which the user is authenticated.

Subelements

none

required

Specifies if the authentication method specified is required to be used for client authentication. If so, the `EstablishTrustInClient` bit will be set in the `target_requires` field of `as-context`. The value is either true or false.

Subelements

none

sas-context

Describes the sas-context fields.

Subelements

The following table describes subelements for the `sas-context` element.

`sas-context` Subelements

Subelement	Required	Description
`caller-propagation`	only one	Specifies if the target will accept propagated caller identities. The values are NONE, SUPPORTED, or REQUIRED.

transport-config

Specifies the security transport information.

Subelements

The following table describes subelements for the `transport-config` element.

`transport-config` Subelements

Subelement	Required	Description
`integrity`	only one	Specifies if the target supports integrity-protected messages. The values are NONE, SUPPORTED, or REQUIRED.
`confidentiality`	only one	Specifies if the target supports privacy-protected messages. The values are NONE, SUPPORTED, or REQUIRED.

`transport-config` Subelements *(Continued)*

Subelement	Required	Description
`establish-trust-in-target`	only one	Specifies if the target is capable of authenticating *to* a client. The values are NONE, SUPPORTED, or REQUIRED.
`establish-trust-in-client`	only one	Specifies if the target is capable of authenticating a client. The values are NONE, SUPPORTED, or REQUIRED.

Persistence Elements

This section describes the elements associated with container-managed persistence (CMP), the persistence manager, and the persistence vendor. For information on using these elements, refer to "Using Container-Managed Persistence," on page 86.

The following elements are included:

- cmp
- cmp-resource
- concrete-impl
- finder
- is-one-one-cmp
- mapping-properties
- method-name
- one-one-finders
- pc-class
- pm-class-generator
- pm-config
- pm-descriptor
- pm-descriptors
- pm-identifier
- pm-inuse
- pm-mapping-factory

- pm-version
- query-filter
- query-params
- query-variables

cmp

Describes runtime information for a container-managed persistence (CMP) entity bean object for EJB1.1 and EJB2.0 beans. This is a pointer to a file that describes the mapping information of a bean.

Subelements

The following table describes subelements for the `cmp` element.

`cmp` Subelements

Subelement	Required	Description
`mapping-properties`	only one	Contains data that specifies the location of the persistence vendor's specific object-to-relational (O/R) database mapping file.
`concrete-impl`	only one	Contains data that specifies the location of the persistence vendor's specific concrete class name.
`pc-class`	zero or one	Contains data that specifies the persistence vendor's specific class.
`is-one-one-cmp`	zero or one	Contains the boolean specifics for container-managed persistence (CMP) 1.1. Used to identify CMP 1.1 with old descriptors.
`one-one-finders`	zero or one	Describes the finders for container-managed persistence (CMP) 1.1.

cmp-resource

Specifies the database to be used for storing container-managed persistence (CMP) beans in an EJB JAR file.

Subelements

The following table describes subelements for the `cmp-resource` element.

`cmp-resource` Subelements

Subelement	Required	Description
`jndi-name`	only one	Specifies the absolute `jndi-name`.
`default-resource-principall`	zero or one	Specifies the default runtime bindings of a resource reference.

concrete-impl

Specifies the location of the persistence vendor's specific concrete class name.

Subelements

none

finder

Describes the finders for container-managed persistence 1.1 with a method name and query.

Subelements

The following table describes subelements for the `finder` element.

`finder` Subelements

Subelement	Required	Description
`method-name`	only one	Specifies the method name for the query field.
`query-params`	only one	Optional data that specifies the query parameters for the container-managed persistence (CMP) 1.1 finder.
`query-filter`	only one	Specifies the query filter for the container-managed persistence (CMP) 1.1 finder.

`finder` Subelements *(Continued)*

Subelement	Required	Description
`query-variables`	only one	Optional data that specifies variables in query expression for the container-managed persistence 1.1 finder.

is-one-one-cmp

Specifies the boolean specifics for container-managed persistence 1.1. Used to identify CMP 1.1 with old descriptors.

Subelements

none

mapping-properties

Specifies the location of the persistence vendor's specific object-to-relational (O/R) database mapping file. Most persistence vendors use the concept of a project, which represents all the related beans and their dependent classes, and can be deployed as a single unit. There can be a vendor-specific XML file associated with the project.

Subelements

none

method-name

Specifies the method name for the query field. The method-name element contains a name of an EJB method or the asterisk (*) character. The asterisk is used when the element denotes all the methods of an EJB's component and home interfaces.

Examples

```
<method-name>create</method-name>
```

```
<method-name>*</method-name>
```

Subelements

none

one-one-finders

Describes the finders for container-managed persistence (CMP) 1.1.

Subelements

The following table describes subelements for the `one-one-finders` element.

`one-one-finders` Subelements

Subelement	Required	Description
`finder`	must have one or more	Describes the finders for container-managed persistence (CMP) 1.1 with a method name and query.

pc-class

Specifies the persistence vendor's specific class.

Subelements

none

pm-class-generator

Specifies which vendor-specific concrete class generator is to be used. This is the name of the class specific to the vendor.

Subelements

none

pm-config

Specifies the vendor-specific configuration file to be used.

Subelements

none

pm-descriptor

Describes the properties of the persistence manager associated with an entity bean.

Subelements

The following table describes subelements for the `pm-descriptor` element.

`pm-descriptor` Subelements

Subelement	Required	Description
`pm-identifier`	only one	Specifies the vendor who provided the persistence manager implementation. For example, this could be Sun ONE Application Server container-managed persistence or a third-party vendor.
`pm-version`	only one	Specifies which version of the persistence manager vendor product is to be used.
`pm-config`	zero or one	Specifies the vendor-specific configuration file to be used.
`pm-config`	zero or one	Specifies which vendor-specific concrete class generator is to be used. This is the name of the class specific to the vendor.
`pm-mapping-factory`	zero or one	Specifies which vendor-specific mapping factory is to be used. This is the name of the class specific to the vendor.

pm-descriptors

Describes the persistence manager descriptors. One of them must be in use at a given time. This basically applies to Sun ONE Application Server pluggable persistence manager APIs.

Subelements

The following table describes subelements for the `pm-descriptors` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

`pm-descriptors` Subelements

Subelement	Required	Description
`pm-descriptor`	one or more	Describes the properties of the persistence manager associated with an entity bean.
`pm-inuse`	only one	Specifies whether this particular persistence manager must be used or not.

pm-identifier

Specifies the vendor who provided the persistence manager implementation. For example, this could be Sun ONE Application Server container-managed persistence or a third-party vendor.

Subelements

none

pm-inuse

Specifies whether this particular persistence manager must be used or not.

Subelements

The following table describes subelements for the `pm-inuse` element.

`pm-insue` Subelements

Subelement	Required	Description
`pm-identifier`	only one	Contains data that specifies the vendor who provided the persistence manager implementation. For example, this could be Sun ONE Application Server container-managed persistence or a third-party vendor.
`pm-version`	only one	Contains data that specifies which version of the persistence manager vendor product is to be used.

pm-mapping-factory

Specifies which vendor-specific mapping factory is to be used. This is the name of the class specific to the vendor.

Subelements

none

pm-version

Specifies which version of the persistence manager vendor product is to be used.

Subelements

none

query-filter

Specifies the query filter for the container-managed persistence 1.1 finder. Optional.

Subelements

none

query-params

Specifies the query parameters for the container-managed persistence 1.1 finder.

Subelements

none

query-variables

Specifies variables in query expression for the container-managed persistence 1.1 finder. Optional.

Subelements

none

Pooling and Caching Elements

This section describes the elements associated with cache, timeout, and the EJB pool. These elements are used to control memory usage and performance tuning. For more information, refer to the *Sun ONE Application Server Performance, Tuning, and Sizing Guide*.

The following elements are discussed:

- bean-cache
- bean-pool
- cache-idle-timeout-in-seconds
- commit-option
- is-cache-overflow-allowed
- max-cache-size
- max-pool-size

- max-wait-time-in-millis
- pool-idle-timeout-in-seconds
- removal-timeout-in-seconds
- resize-quantity
- steady-pool-size
- victim-selection-policy

bean-cache

Specifies the entity bean cache properties. Used for entity beans and stateful session beans.

Subelements

The following table describes subelements for the `bean-cache` element.

`bean-cache` Subelements

Subelement	Required	Description
`max-cache-size`	zero or one	Specifies the maximum number of beans allowable in cache.
`is-cache-overflow-allowed`	zero or one	Deprecated.
`cache-idle-timeout-in-seconds`	zero or one	Specifies the maximum time that a stateful session bean or entity bean is allowed to be idle in cache before being passivated. Default value is 10 minutes (600 seconds).
`removal-timeout-in-seconds`	zero or one	Specifies the amount of time a bean remains before being removed. If `removal-timeout-in-seconds` is less than `idle-timeout`, the bean is removed without being passivated.
`resize-quantity`	zero or one	Specifies the number of beans to be created if the pool is empty (subject to the `max-pool-size` limit). Values are from 0 to MAX_INTEGER.
`victim-selection-policy`	zero or one	Specifies the algorithm that must be used by the container to pick victims. Applies only to stateful session beans.

Example

```
<bean-cache>
   <max-cache-size>100</max-cache-size>
   <cache-resize-quantity>10</cache-resize-quantity>
   <removal-timeout-in-seconds>3600</removal-timeout-in-seconds>
   <victim-selection-policy>LRU</victim-selection-policy>
      <cache-idle-timeout-in-seconds>
      600
      </cache-idle-timeout-in-seconds>
   <removal-timeout-in-seconds>5400</removal-timeout-in-seconds>
</bean-cache>
```

bean-pool

Specifies the pool properties of stateless session beans, entity beans, and message-driven bean.

Subelements

The following table describes subelements for the `bean-pool` element.

`bean-pool` Subelements

Subelement	Required	Description
`steady-pool-size`	zero or one	Specifies the initial and minimum number of beans maintained in the pool. Default is 32.
`resize-quantity`	zero or one	Specifies the number of beans to be created if the pool is empty (subject to the `max-pool-size` limit). Values are from 0 to MAX_INTEGER.
`max-pool-size`	zero or one	Specifies the maximum number of beans in the pool. Values are from 0 to MAX_INTEGER. Default is to `server.xml` or 60.
`max-wait-time-in-millis`	zero or one	Deprecated.
`pool-idle-timeout-in-seconds`	zero or one	Specifies the maximum time that a bean is allowed to be idle in the pool. After this time, the bean is removed. This is a hint to the server. Default time is 600 seconds (10 minutes).

Example

```
<bean-pool>
   <steady-pool-size>10</steady-pool-size>
   <resize-quantity>10</resize-quantity>
   <max-pool-size>100</max-pool-size>
   <pool-idle-timeout-in-seconds>600</pool-idle-timeout-in-seconds>
</bean-pool>
```

cache-idle-timeout-in-seconds

Optionally specifies the maximum time that a bean can remain idle in the cache. After this amount of time, the container can passivate this bean. A value of `0` specifies that beans may never become candidates for passivation. Default is 600.

Applies to stateful session beans and entity beans.

Subelements

none

commit-option

Optionally specifies the commit option that will be used on transaction completion. Valid values for the Sun ONE Application Server are B or C. Default value is B.

NOTE	Commit option A is not supported for the Sun ONE Application Server 7 release.

Applies to entity beans.

Subelements

none

Example

```
<commit-option>B</commit-option>
```

is-cache-overflow-allowed

This element is deprecated and should not be used.

max-cache-size

Optionally specifies the maximum number of beans allowable in cache. A value of zero indicates an unbounded cache. In reality, there is no hard limit. The max-cache-size limit is just a hint to the cache implementation. Default is 512.

Applies to stateful session beans and entity beans.

Subelements

none

Example

```
max-cache-size>100</max-cache-size>
```

max-pool-size

Optionally specifies the maximum number of bean instances in the pool. Values are from 0 (1 for message-driven bean) to MAX_INTEGER. A value of 0 means the pool is unbounded. Default is 64.

Applies to all beans.

Subelements

none

Example

```
<max-pool-size>100</max-pool-size>
```

max-wait-time-in-millis

This element is deprecated and should not be used.

pool-idle-timeout-in-seconds

Optionally specifies the maximum time, in seconds, that a bean instance is allowed to remain idle in the pool. When this timeout expires, the bean instance in a pool becomes a candidate for passivation or deletion. This is a hint to the server. A value of 0 specifies that idle beans can remain in the pool indefinitely. Default value is 600.

Applies to stateless session beans, entity beans, and message-driven beans.

NOTE For a stateless session bean or a message-driven bean, the bean can be removed (garbage collected) when the timeout expires.

Subelements

none

Example

```
<pool-idle-timeout-in-seconds>600</pool-idle-timeout-in-seconds>
```

removal-timeout-in-seconds

Optionally specifies the amount of time a bean instance can remain idle in the container before it is removed (timeout). A value of 0 specifies that the container does not remove inactive beans automatically. The default value is 5400.

If `removal-timeout-in-seconds` is less than or equal to `cache-idle-timeout-in-seconds`, beans are removed immediately without being passivated.

Applies to stateful session beans.

For related information, see `cache-idle-timeout-in-seconds`.

Subelements

none

Example

```
<removal-timeout-in-seconds>3600</removal-timeout-in-seconds>
```

resize-quantity

Optionally specifies the number of bean instances to be:

- Created, if a request arrives when the pool has less than `steady-pool-size` quantity of beans (applies to pools only for creation). If the pool has more than `steady-pool-size` minus `resize-quantity` of beans, then `resize-quantity` is still created.
- Removed, when the `pool-idle-timeout-in-seconds` timer expires and a cleaner thread removes any unused instances.
    - For caches, when `max-cache-size` is reached, `resize-quantity` beans will be selected for passivation using `victim-selection-policy`. In addition, the `cache-idle-timeout-in-seconds` or `cache-remove-timeout-in-seconds` timers will passivate beans from the cache.

- For pools, when the `max-pool-size` is reached, `resize-quantity` beans will be selected for removal. In addition the `pool-idle-timeout-in-seconds` timer will remove beans until `steady-pool-size` is reached.

Values are from 0 to MAX_INTEGER. The pool is not resized below the `steady-pool-size`. Default is 16.

Applies to stateless session beans, entity beans, and message-driven beans.

For EJB pools, the default value can be the value of the `ejb-container` element `pool-resize-quantity` in the `server.xml` file. Default is 16.

For EJB caches, the default value can be the value of the `ejb-container` element `cache-resize-quantity` in the `server.xml` file. Default is 32.

For message-driven beans, the default can be the value of the `mdb-container` `pool-resize-quantity` element in the `server.xml` file. Default is 2.

Subelements

none

Example

```
<resize-quantity>10</resize-quantity>
```

steady-pool-size

Optionally specifies the initial and minimum number of bean instances that should be maintained in the pool. Default is 32.

NOTE If `steady-pool-size` is set to a value greater than 0, the beans are created when the server starts. If a bean relies on caching information during the `setInitialContext` method that is not available at server startup (such as a user's security role), then the bean should throw `EJBException` during the `setInitialContext`. The container handles this exception and does not instantiate the beans. If the bean swollows this exception, then `steady-pool-size` should be set to 0 in the `sun-ejb-jar.xml` file.

Applies to stateless session beans and message-driven beans.

Subelements

none

Example

```
<steady-pool-size>10</steady-pool-size>
```

victim-selection-policy

Optionally specifies how stateful session beans are selected for passivation. Possible values are First In, First Out (FIFO), Least Recently Used (LRU), Not Recently Used (NRU). The default value is NRU, which is actually pseudo-LRU.

NOTE The user cannot plug in his own victim selection algorithm.

The victims are generally passivated into a backup store (typically a file system or database). This store is cleaned during startup, and also by a periodic background process that removes idle entries as specified by `removal-timeout-in-seconds`. The backup store is monitored by a background thread (or sweeper thread) to remove unwanted entries.

Applies to stateful session beans.

Subelements

none

Example

```
<victim-selection-policy>LRU</victim-selection-policy>
```

Class Elements

This section describes the elements associated with classes. The following elements are included:

- gen-classes
- local-home-impl
- local-impl
- remote-home-impl
- remote-impl

gen-classes

Specifies all the generated class names for a bean.

NOTE This is automatically generated by the server at deployment/redeployment time. It should not be specified by the developer or changed after deployment.

Subelements

The following table describes subelements for the `gen-class` element.

`gen-classes` Subelements

Subelement	Required	Description
`remote-impl`	zero or one	Specifies the fully-qualified class name of the generated `EJBObject` impl class.
`local-impl`	zero or one	Specifies the fully-qualified class name of the generated `EJBLocalObject` impl class.
`remote-home-impl`	zero or one	Specifies the fully-qualified class name of the generated EJBHome impl class.
`local-home-impl`	zero or one	Specifies the fully-qualified class name of the generated `EJBLocalHome` impl class.

local-home-impl

Specifies the fully-qualified class name of the generated `EJBLocalHome` `impl` class.

NOTE This is automatically generated by the server at deployment/redeployment time. It should not be specified by the developer or changed after deployment.

Subelements

none

local-impl

Specifies the fully-qualified class name of the generated `EJBLocalObject` `impl` class.

NOTE This is automatically generated by the server at deployment/redeployment time. It should not be specified by the developer or changed after deployment.

Subelements

none

remote-home-impl

Specifies the fully-qualified class name of the generated `EJBHome` `impl` class.

NOTE This is automatically generated by the server at deployment/redeployment time. It should not be specified by the developer or changed after deployment.

Subelements

none

remote-impl

Specifies the fully-qualified class name of the generated `EJBObject` `impl` class.

NOTE This is automatically generated by the server at deployment/redeployment time. It should not be specified by the developer or changed after deployment.

Subelements

none

Sample EJB XML Files

This section includes the following sample files:

- Sample ejb-jar.xml File
- Sample sun-ejb-jar.xml File

For information on the elements associated with enterprise beans, refer to "Elements in the sun-ejb-jar.xml File," on page 176 and the *Sun ONE Application Server Developer's Guide*.

Sample ejb-jar.xml File

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

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN'

'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
   <description>no description</description>
   <display-name>CustomerJAR</display-name>
      <enterprise-beans>
         <entity>
            <description>no description</description>
            <display-name>CustomerEJB</display-name>
            <ejb-name>CustomerEJB</ejb-name>
            <home>samples.SimpleBankBMP.ejb.CustomerHome</home>
            <remote>samples.SimpleBankBMP.ejb.Customer</remote>
            <ejb-class>samples.SimpleBankBMP.ejb.CustomerEJB</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>java.lang.String</prim-key-class>
            <reentrant>False</reentrant>
            <security-identity>
               <description></description>
               <use-caller-identity></use-caller-identity>
            </security-identity>
            <resource-ref>
               <res-ref-name>jdbc/SimpleBank</res-ref-name>
               <res-type>javax.sql.DataSource</res-type>
               <res-auth>Container</res-auth>
               <res-sharing-scope>Shareable</res-sharing-scope>
            </resource-ref>
         </entity>
      </enterprise-beans
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>CustomerEJB</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>NotSupported</trans-attribute>
      </container-transaction>
   </assembly-descriptor>
</ejb-jar>
```

Sample sun-ejb-jar.xml File

For information on these elements, refer to "Elements in the sun-ejb-jar.xml File," on page 176.

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

<!DOCTYPE sun-ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Sun ONE Application
Server 7.0 EJB 2.0//EN'

'http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2_0-0.dtd'>

<sun-ejb-jar>
   <display-name>First Module</display-name>
   <enterprise-beans>
      <ejb>
         <ejb-name>CustomerEJB</ejb-name>
         <jndi-name>customer</jndi-name>
         <resource-ref>
            <res-ref-name>jdbc/SimpleBank</res-ref-name>
‘           <jndi-name>jdbc/PointBase</jndi-name>
         </resource-ref>
         <is-read-only-bean>false</is-read-only-bean>
         <commit-option>B</commit-option>
         <bean-pool>
            <steady-pool-size>10</steady-pool-size>
            <resize-quantity>10</resize-quantity>
            <max-pool-size>100</max-pool-size>
            <pool-idle-timeout-in-seconds>600</pool-idle-timeout-in-seconds>
         </bean-pool>
         <bean-cache>
            <max-cache-size>100</max-cache-size>
            <resize-quantity>10</resize-quantity>
               <removal-timeout-in-seconds>3600</removal-timeout-in-seconds>
            <victim-selection-policy>LRU</victim-selection-policy>
            </bean-cache>
        </ejb>
    </enterprise-beans>
</sun-ejb-jar>
```

Appendix A

CMP Mapping with the Sun ONE Studio 4 Interface

This section provides guidelines on mapping between a set of Java programming language classes and a relational database using the Sun ONE Studio 4 interface.

This section addresses the following topics:

- Mapping CMP Beans
- EJB Persistence Properties

You should already be familiar with the "Using Container-Managed Persistence for Entity Beans," on page 79," and chapter 10 of the Enterprise JavaBeans Specification, v2.0 before using these procedures.

Mapping CMP Beans

To map container-managed persistence beans, you must first capture the schema, then map the beans to the schema.

This section contains the following sections:

- Capturing a Schema
- Mapping Existing Enterprise Beans to a Schema

Capturing a Schema

Before mapping any enterprise beans to a database schema, you need to capture the schema to create a working copy in your file system. This allows you to do your work without affecting the database itself.

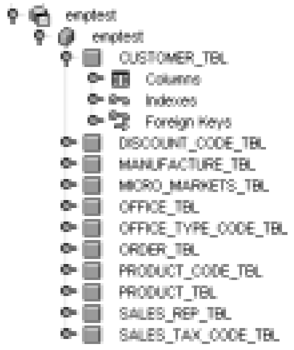
NOTE It is best to store the captured schema in a package. If you do not have a package to contain the schema, create one by right-clicking on the file system and selecting New Package.

1. You have three ways to display the Mapping Tool:
   - Right-click on the file system and select New > Databases > Database Schema.
   - Choose New from the File menu and then, in the Template Chooser, double-click Databases and select Database Schema.
   - Select Capture Database Schema from the Tools menu.
2. In the Target Location pane, type a file name for the working copy of your schema, then select a package for the captured schema.
3. In the Database Connection pane, if you have a connection established, you can select it from the Existing Connection menu. Otherwise, under New Connection, enter the following information:
   - The name of the database you are connecting to. (If your database is not listed in the dropdown menu, you might need to quit the Mapping Tool and install the driver in the IDE before continuing.)
   - Your system's JDBC driver.
   - The JDBC URL for the database, including the driver identifier, server, port, and database name. For example, `jdbc:pointbase://localhost:9092/sample.`

     The format of a JDBC URL varies depending on which kind of database management system (DBMS) you use—Oracle, Microsoft SQL Server, or PointBase—and the version of that DBMS. Ask your system administrator for the correct URL format for your DBMS.
   - A user name for your database.
   - The password for that user.
4. In the Tables and Views pane, choose the tables and views you want to capture, then click Finish.

NOTE If you choose one table and exclude another that is referenced to the included table by a foreign key, both tables will be captured even though you specified only one.

The database and its schema will be represented as shown in following figure.



Mapping Existing Enterprise Beans to a Schema

This section discusses how to use container-managed persistence to customize mappings or to create a mapping for an existing object model.

Before you can map an enterprise bean to a database schema, you must make sure that the database schema is captured and mounted in your Explorer file system. See “Capturing a Schema,” on page 217 for instructions on how to do this.

You can set up or edit a mapping piecemeal by editing the individual properties in the Properties window. All the mapping and persistence information can be accessed through the Properties window. The mapping fields property editor provides a way to view and edit groups of classes and fields at one time, providing a useful overview of your mapping model.

1. Under Filesystem, open the EJB Module.

   The enterprise beans in that module are listed.

2. Select the enterprise bean from its containing EJB module.

   The properties table for the enterprise bean is displayed.

3. If you have completed the preliminary tasks, click Next to bring up the Select Tables pane of the Mapping Tool.

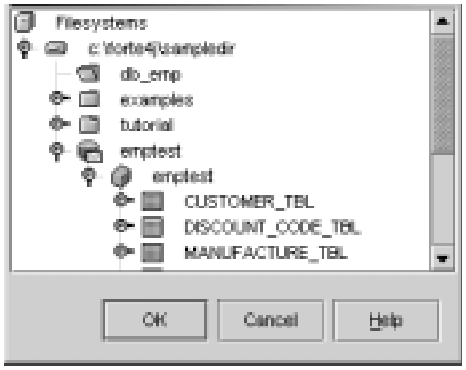
   Otherwise, click Cancel, complete the tasks, and restart the Mapping Tool.

4. Select a primary table from the Primary Table combo box, or click Browse to open the Select Primary Table dialog.

5. If you open the Select Primary Table dialog, find a schema and expand it to find its tables.

6. Select a table and click OK.

   The table you select as the primary table should be the one that most closely matches your class.



7. Once the primary table is set up, you can map one or more secondary tables by clicking Add.

   This opens the Secondary Table Settings dialog box.

   A secondary table enables you to map fields in your enterprise bean to columns that are not part of your primary table. For example, you might add a DEPARTMENT table as a secondary table in order to include a department name in your Employee class. A secondary table differs from a relationship, in which one class is related to another by way of a relationship field. In a secondary table mapping, fields in the same class are mapped to two different tables. A secondary table enables you to map your field directly to columns that are not part of your primary table. You can use this pane to select secondary tables, and to show how they are linked to the primary table.

A secondary table must be related to the primary table by one or more columns whose associated rows have the same values in both tables. Normally, this is defined as a foreign key between the tables. When you select a secondary table from the drop-down menu, the Mapping Tool checks for a foreign key between the two tables. If a foreign key exists, it is displayed as the reference key by default.

a. Select a secondary table from the combo box.

Once you select a secondary table, the container-managed persistence implementation checks to see if there is a foreign key between the primary and secondary tables. If so, the foreign key is displayed as the default reference key. If there is no foreign key, the editor displays Choose Column, and you must set up a reference key.

b. To set up a reference key, click Choose Column and select a column from the dropdown menu.

Once you pick a primary column, the choices in the secondary column are limited to columns of compatible types. If no column is compatible, the field displays No Compatible Columns. If you select a primary column that is incompatible with your secondary column, the value of the secondary column reverts to Choose Column.

NOTE If no pair of columns seems to relate in a logical manner, preventing a logical reference key, you may want to reconsider your choice of a secondary table.

You can select the Add Pair key to set up a complex key using more than one pair of columns.

8. Click OK to save your selections.

9. Click Next in the Mapping Tool to bring up the Field Mappings panel of the Mapping Tool.

   The Field Mappings panel displays all the persistent fields of the enterprise bean and their mapping status. You can map a field to a column by selecting the column in the drop-down menu for that field, or try to map all unmapped fields by selecting Automap. Automap will make the most logical selections, ignoring any relationship fields and any fields that have already been mapped. It will not change any existing mappings.

   If you want to map a field to a column from another table that is not available, click Previous to return to the previous Mapping Tool page and add a secondary table that contains the column you want.

   Unmap works on whatever field or fields are selected. You can unmap a group of fields at once by holding down the Shift key or Control key while selecting the fields you want. If you want to unmap one item, choose Unmapped in the drop-down menu for that field.

   a. To map a field to multiple columns, click the ellipsis button (...) for the appropriate field in the Field Mappings pane to display the Map Field to Multiple Columns dialog box.

      In this dialog box, you add columns to the list of mapped columns. Columns are from the tables you have mapped to this class.You can change the order of the columns by using Move Up / Move Down.

      If you do not see the column you want to map, you might need to add a secondary table to your mapping, or change the primary table you have selected. If no columns are listed, you have not yet mapped a primary table, or you have mapped a table that has no columns.

      If you map a field to more than one column, all columns will be updated with the value of the first column listed. Therefore, if the value of one of the columns is changed outside of a container-managed persistence application, the value will only be read if the change was made to that first column. Writing a value to the database overwrites any conflicting changes made to any other columns.

      You must also make sure that if you map more than one field to any of these columns, the mappings cannot partially overlap. Consider the following three scenarios:

      - Field A mapped to Columns A and B, Field B mapped to Column B. Since the mappings only partially overlap, this example will get a validation error at compilation.

- Field A mapped to Column A, and Field B mapped to Column B. Since there is no overlap, this mapping is allowed.
- Field A mapped to Columns A and B, Field B mapped to Columns A and B. Since the mappings completely overlap, this mapping is allowed.

b. Click OK to save the mapping.

Mapping Relationship Fields

When you have foreign keys between database tables, you usually want to preserve those relationships in Java class references. Mapping CMR fields lets you specify the relationships that correspond to the class reference fields.

1. To Map a Relationship Field, click the ellipsis button (...) in the Field Mappings panel next to the drop-down menu of a relationship field to bring up the Relationship Mapping editor.

To use the Relationship Mapping editor outside of the Mapping Tool, click the relationship field in Explorer and edit its Mapping property.

a. In this pane, verify that the Related Class is set. If the related class is not set, then set it. If the class you want to select is not persistence-capable, you might need to cancel out of the editor, convert the class to persistence-capable, then return.

b. Verify that the Related Field (if any) is also correct, and that the Primary Table is set for the related class.

NOTE If you have a logical related field, you should choose a Primary Table. That will create a managed relationship.

---

c. Select between linking the tables directly, or through a join table.

2. If your relationships are one-to-one or one-to-many, choose to link the tables directly. Clicking Next opens the Map to Key pane of the Relationship Mapping editor.

This pane shows:

❍ An existing mapping if there is one and there were no changes on the initial setup page.

- The default mapping if there is no existing mapping or the mapping is no longer valid.

  The editor attempts to determine the most logical key column pairs between the two related classes, based on existing foreign keys. If there are no foreign keys, you need to create the key column pairs by selecting local and foreign columns. The columns in each pair are expected to have the same value.

  To create a complex key, use the Add Pair button to add additional Key Column Pairs.

  If the Finish button is disabled, you need to choose a key column pair.

3. If your relationship is many-to-many, link tables through a join table. Click Next to open the Map to Key: Local to Join pane.

   This pane shows:

   - The first class and field in the relationship
   - The join table to be used to create the relationship between the fields
   - Key column pairs between the field join table and the table to which the related class is mapped

     In this pane, you choose a join table, then map the relationship field to a key. This is only the relationship between the table This Class is mapped to and the join table. If you don't have a join table, go back to the previous panel and select Link the Mapped Tables Directly.

     Choose a join table that sits between the two tables that your classes are mapped to. The Editor will attempt to determine the most logical key column pairs between the join table and the table that This Class is mapped to.

     If the tables have a foreign key between them, the editor will use the foreign key as the default key column pair. If there is no foreign key, then you must create a key by choosing a pair of columns that will allow navigation from the join table to the table to which This Class is mapped. The columns in each pair are expected to have the same value.

     To create a compound key, use Add pair to add additional Key Column Pairs.

     If the Next button is disabled, you need to pick a join table or make sure that at least one key column pair exists that has columns on both sides.

4. Click Next to open the Map to Key: Join to Foreign pane.

   In this pane, you relate a second table to the join table you chose in the previous pane.

   The editor will attempt to determine the most logical key column pairs between the join table and the table that the Related Class is mapped to.

   If the tables have a foreign key between them, the editor will use the foreign key as the default key column pair. If there is no foreign key, then you must create a key by choosing a pair of columns that will allow navigation from the join table to the table to which the Related Class is mapped. The columns in each pair are expected to have the same value.

   To create a compound key, use Add Pair to add additional key column pairs.

   If the Finish button is disabled, you need to choose a valid key column pair.

5. Click Finish to return to the Field Mappings pane of the Mapping Tool.

6. Click Finish to close the Field Mappings pane and map the Java classes to the database schema.

EJB Persistence Properties

Enterprise beans that use container-managed persistence have several unique properties that can be specified outside the Mapping Tool.

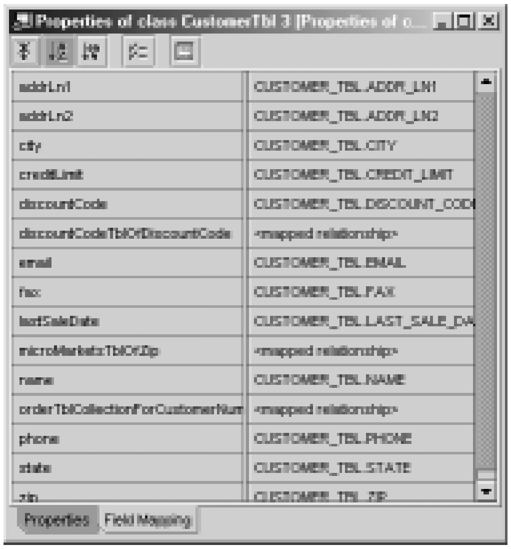
The following table describes these unique properties.

Properties for CMP Enterprise Beans

Property	Description
Mapped primary table	The primary table you select for a persistence-capable class should be the table in the schema that most closely matches the class. You must specify a primary table in order to map a persistence-capable class. See “Mapping Existing Enterprise Beans to a Schema,” on page 219 for information on how to do this.
Mapped schema	The schema containing the tables to which you are mapping the persistence-capable class. The primary table, secondary tables, and related classes must be from this schema. This setting cannot be made until you capture the schema as described in “Capturing a Schema,” on page 217.
Mapped secondary table(s)	Secondary tables let you map columns that are not part of your primary table to your class fields. For example, you might add a `DEPARTMENT` table as a secondary table in order to include a department name in your `Employee` class. You can add multiple secondary tables, but no secondary table is required. This property is only enabled when Mapped Primary Table is set. See page 98 and page 221 for more information on adding a secondary table.
Consistency levels	Specifies container behavior in guaranteeing transactional consistency of the data in the bean. If the consistency checking flag element is not present, `none` is assumed. For further information on consistency levels, see “consistency,” on page 115.
Fetch groups	The `fetched-with` property specifies the fetch group configuration for fields and relationships. A field may participate in a hierarchical or independent fetch group. If the `fetched-with` element is not present, the following value is assumed: `<fetched-with><none/></fetched-with>`. Refer to “fetched-with,” on page 117 for further information.

You can unmap a class by choosing `<unmapped>` from the drop-down menu for the Mapped Primary Table property. When you unmap a currently mapped class, a warning appears if there are field mappings or secondary tables. Click OK if you are sure that you want to unmap the class. Otherwise, click Cancel to cancel the mapping status change and leave the class mapped.

Click the Field Mapping tab at the bottom of the Properties window to see the field mapping properties for a persistence-capable class.



addrLn1	CUSTOMER_TBL.ADDR_LN1
addrLn2	CUSTOMER_TBL.ADDR_LN2
city	CUSTOMER_TBL.CITY
creditLimit	CUSTOMER_TBL.CREDIT_LIMIT
discountCode	CUSTOMER_TBL.DISCOUNT_COD
discountCodeTblOfDiscountCode	<mapped relationship>
email	CUSTOMER_TBL.EMAIL
fax	CUSTOMER_TBL.FAX
lastSaleDate	CUSTOMER_TBL.LAST_SALE_DA
microMarketsTblOfZip	<mapped relationship>
name	CUSTOMER_TBL.NAME
orderTblCollectionForCustomerNum	<mapped relationship>
phone	CUSTOMER_TBL.PHONE
state	CUSTOMER_TBL.STATE
zip	CUSTOMER_TBL.ZIP

Appendix B

Elements Listings

This section provides alphabetic listings of the elements for the DTD files associated with Enterprise JavaBeans (EJBs) in the Sun ONE Application Server 7 environment.

This section addresses the following topics:

- sun-ejb-jar_2_0-0.dtd File Elements
- sun-cmp-mapping_1_0.dtd File Elements

sun-ejb-jar_2_0-0.dtd File Elements

Explanations on these elements are contained in "Elements in the sun-ejb-jar.xml File," on page 176.

`as-context`

`auth-method`

`bean-cache`

`bean-pool`

`cache-idle-timeout-in-seconds`

`caller-propagation`

`cmp`

`cmp-resource`

`commit-option`

`concrete-impl`

`confidentiality`

`default-resource-principal`

```
ejb
ejb-name
ejb-ref
ejb-ref-name
enterprise-beans
establish-trust-in-client
establish-trust-in-target
finder
gen-classes
group-name
integrity
ior-security-config
is-cache-overflow-allowed
is-one-one-cmp
is-read-only-bean
jms-durable-subscription-name
jms-max-messages-load
jndi-name
local-home-impl
local-impl
mapping-properties
max-cache-size
max-pool-size
max-wait-time-in-millis
mdb-connection-factory
method-name
name
one-one-finders
pass-by-reference
password
pc-class
```

pm-class-generator

pm-config

pm-descriptor

pm-descriptors

pm-identifier

pm-inuse

pm-mapping-factory

pm-version

pool-idle-timeout-in-seconds

principal

principal-name

query-filter

query-params

query-variables

realm

refresh-period-in-seconds

remote-home-impl

remote-impl

removal-timeout-in-seconds

required

res-ref-name

resize-quantity

resource-env-ref

resource-env-ref-name

resource-ref

role-name

sas-context

security-role-mappingserver-name

steady-pool-size

sun-ejb-jar

transport-config

`unique-id`

`victim-selection-policy`

sun-cmp-mapping_1_0.dtd File Elements

Explanations on these elements are contained in "Mapping Fields and Relationships," on page 94 and "Elements in the sun-cmp-mappings.xml File," on page 112.

`check-all-at-commit`

`check-modified-at-commit`

`cmp-field-mapping`

`cmr-field-mapping`

`cmr-field-name`

`column-name`

`column-pair`

`consistency`

`ejb-name`

`entity-mapping`

`fetched-with`

`field-name`

`level`

`lock-when-loaded`

`lock-when-modified`

`named-group`

`none`

`read-only`

`schema`

`secondary-table`

`sun-cmp-mapping`

`sun-cmp-mappings`

`table-name`

Index