Oracle9*i*AS TopLink

CMP for Users of BEA WebLogic Server Guide

Release 2 (9.0.3)

August 2002

Part No. B10065-01

ORACLE

Oracle 9*i*AS TopLink CMP for Users of BEA WebLogic Server Guide, Release 2 (9.0.3)

Part No. B10065-01

Contents

5 Defining and Executing Finders

6 Run-time Considerations

7 Customization

8 Clustering

9 The EJB 2.0 Single Bean Example Application

A EJB Architectures Summary

Send Us Your Comments

Oracle 9*i*AS TopLink CMP for Users of BEA WebLogic Server Guide, Release 2 (9.0.3)

Part No. B10065-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: iasdocs_us@oracle.com
- FAX: 650-506-7407 Attn: Oracle9*i* Application Server Documentation Manager
- Postal service:
  Oracle Corporation
  Oracle9*i* Application Server Documentation
  500 Oracle Parkway, M/S 2op3
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This section introduces the information you need to get the most out of the documentation that accompanies your software. This preface contains these topics:

Intended Audience

This document is intended for application developers who perform the following tasks:

- Application design and development
- Application testing and benchmarking
- Application integration

This document assumes that you are familiar with the concepts of object-oriented programming, the Enterprise JavaBeans (EJB) specification, and with your own particular Java development environment.

The document also assumes that you are familiar with your particular operating system (Windows, UNIX, or other). The general operation of any operating system is described in the user documentation for that system, and is not repeated in this manual.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Structure

This document contains:

Chapter 1, "Introduction"

This chapter provides an overview of the TopLink and CMP concepts that enable you to fully-leverage TopLink CMP.

Chapter 2, "Mapping Entity Beans"

This chapter describes how to map container-managed entity beans using the object mapping features of TopLink for Java. Instructions and hints for using direct and relationship mappings in an EJB context are provided, and differences between beans and regular Java objects are outlined.

Chapter 3, "Configuring TopLink Container-Managed Persistence"

This chapter describes the configuration and testing of TopLink Container-Managed Persistence.

Chapter 4, "EJB Entity Bean Deployment"

This chapter describes how to deploy beans within the application server.

Chapter 5, "Defining and Executing Finders"

This chapter describes the TopLink support for creating and customizing finders.

Chapter 6, "Run-time Considerations"

This chapter discusses some of the run-time issues associated with developing an application that uses TopLink Container-Managed Persistence.

Chapter 7, "Customization"

This chapter describes advanced customization of mappings, logins, and other aspects of persistence. These customizations enable you to take advantage of advanced TopLink features, JDBC driver features, or gain “low-level” access to some of TopLink for Java APIs that are normally masked.

Chapter 8, "Clustering"

This chapter describes the integration of multiple server instances into what can be viewed by clients as a single server entity. This is referred to as clustering.

Chapter 9, "The EJB 2.0 Single Bean Example Application"

This chapter introduces the basic concepts that are required to build and deploy an entity bean with TopLink. It provides an example of how TopLink CMP is used in a simple application that combines Java server pages (JSPs) and EJBs.

Appendix A, "EJB Architectures Summary"

This appendix provides an overview of some of the basic design patterns available when using TopLink and TopLink CMP. It briefly suggests some of the more useful EJB designs and their suitability to specific applications.

Appendix B, "The toplink-ejb-jar DTD"

This appendix contains a listing of the `toplink-ejb-jar` document type description (DTD).

Related Documents

For more information, see these Oracle resources:

Oracle9*i*AS TopLink Getting Started

Provides installation procedures to install and configure TopLink. It also introduces the concepts with which you should be familiar to get the most out of TopLink.

Oracle9*i*AS TopLink Tutorials

Provides tutorials illustrating the use of TopLink. It is written for developers who are familiar with the object-oriented programming and Java development environments.

Oracle9*i*AS TopLink Foundation Library Guide

Introduces TopLink and the concepts and techniques required to build an effective TopLink application. It also gives a brief overview of relational databases and describes who TopLink accesses relational databases from the object-oriented Java domain.

Oracle9*i*AS TopLink Mapping Workbench Reference Guide

Includes the concepts required for using the TopLink Mapping Workbench, a stand-alone application that creates and manages your descriptors and mappings for a project. This document includes information on each Mapping Workbench function and option and is written for developers who are familiar with the object-oriented programming and Java development environments.

Oracle9*i*AS TopLink Container Managed Persistence for Application Servers

Provides information on TopLink container-managed persistence (CMP) support for application servers. Oracle provides an individual document for each application server specifically supported by TopLink CMP.

Oracle9*i*AS TopLink Troubleshooting

Contains general information about TopLink's error handling strategy, the types of errors that can occur, and Frequently Asked Questions (FAQs). It also discusses troubleshooting procedures and provides a list of the exceptions that can occur, the most probable cause of the error condition, and the recommended action.

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://otn.oracle.com/admin/account/membership.html`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://otn.oracle.com/docs/index.htm`

Conventions

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.

Convention	Meaning	Example
lowercase monospace (fixed-width) font	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id` and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
lowercase italic monospace (fixed-width) font	Lowercase italic monospace font represents placeholders or variables.	You can specify the *parallel_clause*. Run `U`*old_release*`.SQL` where *old_release* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*digits* `[ ,` *precision* `])`
{ }	Braces enclose two or more items, one of which is required.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`

Convention	Meaning	Example
. . .	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	`CREATE TABLE ... AS subquery;` `SELECT col1, col2, ... , coln FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

Conventions for Microsoft Windows Operating Systems

The following table describes conventions for Microsoft Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose **Start** >	How to start a program.	To start the Oracle Database Configuration Assistant, choose **Start** > **Programs** > **...** .
Case sensitivity and file and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \\, then Windows assumes it uses the Universal Naming Convention.	`c:\winnt"\"system32` is the same as `C:\WINNT\SYSTEM32`
	IMPORTANT NOTE: File names and directory names *are* case sensitive under UNIX. Where the name of a file or directory is mentioned and the operating system is a non-Windows platform, you must enter the names exactly as they appear unless instructed otherwise.	

Convention	Meaning	Example
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`

	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (') do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	`C:\>exp scott/tiger TABLES=emp QUERY=\"WHERE job='SALESMAN' and sal<1600\"`
		`C:\>imp SYSTEM/`*`password`*
		`FROMUSER=scott TABLES=(emp, dept)`

`<`*`INSTALL_DIR`*`>`	Represents the Oracle home installation directory name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	`SET CLASSPATH=<`*`INSTALL_DIR`*`>\jre\bin`

Convention	Meaning	Example
ORACLE_HOME and *ORACLE_BASE*	In releases prior to Oracle8*i* release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *ORACLE_HOME* directory that by default used one of the following names: ▪ `C:\orant` for Windows NT ▪ `C:\orawin95` for Windows 95 ▪ `C:\orawin98` for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *ORACLE_HOME* directory. There is a top level directory called *ORACLE_BASE* that by default is `C:\oracle`. If you install Oracle9*i* release 1 (9.0.1) on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is `C:\oracle\ora90`. The Oracle home directory is located directly under *ORACLE_BASE*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle9i Database Getting Starting for Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *ORACLE_BASE*`\`*ORACLE_HOME*`\rdbms\admin` directory.

1

Introduction

This document includes instructions on installing, configuring, and testing your software, and includes information that will help you get the most out of TopLink Container-Managed Persistence, as well as a demonstration application.

If you are a new user, go through the single bean example, as well as the other examples included in the TopLink installation. These examples provide you with some hands-on experience with TopLink, and give you a better understanding of TopLink's power and usefulness.

TopLink Container-Managed Persistence

TopLink Container-Managed Persistence is an extension of the TopLink for Java persistence framework. In addition to providing all of TopLink for Java's object-relational persistence facilities, TopLink Container-Managed Persistence also provides container-managed persistence (CMP) for Enterprise JavaBeans (EJBs) deployed in the BEA WebLogic server.

TopLink's CMP supports complex mappings from entity beans to relational database tables, and enables you to model relationships between beans, and between beans and regular Java objects. TopLink provides a rich set of querying options and allows query definition at the bean-level rather than the database level.

TopLink Container-Managed Persistence provides container-managed persistence and other object-relational mapping features for BEA WebLogic Server 6.1 (service Pack 3) and 7.0. Earlier versions of BEA WebLogic Server are not supported by this release.

TopLink Container-Managed Persistence supports the EJB 1.1 and EJB 2.0 specifications as defined by Sun Microsystems.

TopLink Container-Managed Persistence is an extension of the TopLink for Java product and shares all of its core functionality.

TopLink for Java

TopLink for Java provides an easy way to map a Java object model to a relational database. TopLink is a persistence framework that bridges the gap between objects and relational databases, and allows you to work at the object level.

TopLink supports the creation of a wide variety of Java applications. For building two-tier, three-tier, or n-tier applications; TopLink can be used within EJB and non-EJB environments. It can also be used within Java application servers or on its own.

If you are using TopLink for persistence requirements other than container-managed persistence (such as traditional two-, three-, or *n*-tier applications, non-EJB applications, or session bean-based applications), refer to the *Oracle9iAS TopLink Foundation Library Guide*. That document includes information on advanced TopLink features that are not included in this manual.

TopLink Mapping Workbench

TopLink Mapping Workbench is a separate tool that provides a graphical method of configuring the descriptors and mappings of a project. It provides many checks to ensure that the descriptor settings are valid, and it also provides advanced functionality for accessing the database and creating a database schema.

The TopLink Mapping Workbench does not generate Java code during development, which would be unmanageable if the descriptors changed. Instead, it stores descriptor information in an XML deployment file, which can be read into a Java application using a TopLink method. When the application needs to be repackaged into a runtime, TopLink can then generate a `.java` file from the XML file, eliminating the need for TopLink Mapping Workbench files at runtime.

The TopLink Mapping Workbench displays all of the project information for a given project, including classes and tables. Refer to the *Oracle9iAS TopLink Mapping Workbench Reference Guide* for more information on editing projects and descriptors using TopLink Mapping Workbench.

Understanding container-managed persistence

This section introduces the concepts required to use TopLink's container-managed persistence (CMP) facilities. It highlights the particular features available in TopLink Container-Managed Persistence that are not available in TopLink's core Java Foundation Library and explains any differences in the use of other core features.

Enterprise JavaBeans (EJBs)

This manual assumes that you have some familiarity with Enterprise JavaBeans (EJBs) and related concepts. This section provides an overview of some of the key terms that are encountered when discussing EJBs.

For more information about Enterprise JavaBeans, visit the Sun Microsystems EJB site at `http://java.sun.com/products/ejb`.

EJB 2.0 Support

TopLink Container-Managed Persistence provides support for EJB 2.0 container-managed persistence (CMP) entity beans. Our implementation is based on the EJB 2.0 support provided in WebLogic Server 6.1 (service Pack 3) and 7.0. This implementation is based on the Final Release of the 2.0 specification.

Some specific features of EJB 2.0 that are supported are:

- support for local interfaces and local relationships
- generation of concrete bean subclasses
- EJB-QL
- automatic management of bi-directional relationships
- Mapping Workbench support for EJB 2.0
- initializing project from ejb-jar.xml
- support for finders
- support for home methods
- support for ejbSelect

Terminology and definitions

Enterprise JavaBeans To quote the Sun EJB specification, an enterprise bean implements a business task, or a business entity. Enterprise JavaBeans are server-side domain objects that fit into a standard component-based architecture for building enterprise applications using the Java language. They are Java objects that, when installed in an EJB server such as the BEA WebLogic Server, become distributed, transactional, and secure components. There are three kinds of EJBs: session beans, entity beans, and message-driven beans.

EJB Server and Container An EJB bean is said to reside within an EJB Container that in turn resides within an EJB Server. The exact distinction between container and server is not completely defined. In general, the server provides the bean with access to various services (transactions, security, and so on.) while the container provides the execution context for the bean by managing its life cycle.

Deployment descriptors The additional information required to install an EJB within its server is provided in the *deployment descriptors* for that bean. The deployment descriptors consists of a set of XML files that provide all of the required security, transaction, relationship, and persistence information for the bean.

Session beans Session beans represent a business operation, task, or process. Although the use of a session bean may involve database access, the beans are not in themselves persistent – they do not directly represent a database entry. Session beans may or may not retain conversational state; they may be stateful and retain client information between calls, or they may be stateless and only retain information within a single method call.

TopLink may be used with session beans to make the regular Java objects that they access persistent, or can be used to access TopLink persistent entity beans. Session beans may also act as wrappers to other legacy applications.

Entity beans Entity beans represent a persistent data object – an object with durable state that exists from one access to the next. To accomplish this, the entity bean must be made persistent in a relational database, object database, or some other storage facility.

Two schemes exist for making entity beans persistent: bean-managed persistence (BMP) and container-managed persistence (CMP). BMP requires that the bean developer hand-code the methods that perform the persistence work. CMP uses information supplied by the developer or deployer to handle all aspects of persistence.

Message-driven beans Message-driven beans process asynchronous Java Message Service (JMS) messages. A bean method is transactionally-invoked by a JMS message sent to the objects registered against the given topic. From a client perspective, a message-driven bean is simply a JMS consumer with no conversational state and no home or remote interfaces.

Java objects and Entity Beans

A Java object contains the following components:

Attributes. Store primitive data such as integers, and also store simple Java types such as String and Date.

Relationships References to other TopLink-enabled classes. A TopLink-enabled class has a descriptor and can be stored in the database. Because TopLink-enabled classes can be stored in a database, they are called persistent classes.

Methods Paths of execution that can be invoked in a Java environment. Methods are not stored in the database because they are static.

An entity bean has the following parts:

The bean instance An instance of an entity bean class supplied by the developer of the bean. It is a regular Java object whose class implements the `javax.ejb.EntityBean` interface. The bean instance has persistent state. The client application should never access the bean instance directly.

The EJBObject An instance of a generated class that implements the remote interface defined by the bean developer. This instance wraps the bean and all client interaction is made through this object. The EJBObject does not have persistent state.

The EJBHome An instance of a class that implements the home interface supplied by the bean developer. This instance is accessible from JNDI and provides all create and finder methods for the EJB. The EJBHome does not have persistent state.

The EJBLocalObject An instance of a generated class that implements the local interface defined by the bean developer. The key difference between an EJBLocalObject and an EJBObject is that the EJBLocalObject can only be accessed from within the same server on which the beans are deployed. The EJBLocalObject does not have persistent state.

The EJBLocalHome An instance of a class that implements the local home interface supplied by the bean developer. This instance is accessible from JNDI and provides all `create` and `finder` methods for the EJB. The key difference between an EJBLocalHome and an EJBHome is that the EJBLocalHome can only be accessed from within the same server on which the beans are deployed even when using JNDI. The EJBLocalHome does not have persistent state.

The EJB Primary Key An instance of the primary key class provided by the bean developer. The primary key is a serializable object whose fields match the primary key fields in the bean instance. Although the EJB Primary Key shares some data with the bean instance, it does not have persistent state. Note that as of EJB 1.1, it is not required that a bean have a separate primary key class when the key consists of a single field.

For more information about BEA WebLogic Server tools, APIs, or concepts, refer to the BEA WebLogic Server documentation (available online at `http://e-docs.beasys.com`).

For more information about the Enterprise JavaBeans standard, visit the Sun Microsystems EJB site at `http://java.sun.com/products/ejb`.

2

Mapping Entity Beans

This chapter describes how to map container-managed entity beans using the object mapping features of TopLink for Java. Instructions and hints for using direct and relationship mappings in an EJB context are provided, and differences between beans and regular Java objects are outlined.

For information on direct and relationship mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*. You should read and thoroughly understand those chapters before attempting to map entity beans.

Using TopLink Mapping Workbench

When using TopLink Mapping Workbench with entity beans, the bean classes themselves should be loaded into TopLink Mapping Workbench. The remote, local, home, and local home interfaces and the primary key class do not need to be loaded, nor should mappings be defined using these classes.

Make sure you include any classes referred to by the entity beans on the classpath that is used by the TopLink Mapping Workbench, otherwise errors may occur when the beans are loaded. The remote, local, home, and localhome interfaces should also be avilable on the classpath, as they may be used during EJB validation.

Mappings

TopLink mappings define how an object's attributes are to be represented in the database. Attributes that are to be persistent, or that reference other beans or mapped objects, must be mapped to the database using either direct or relationship mappings.

To enable container-managed persistent storage of entity beans, the attributes on the bean implementation class must be mapped. The implementation class is the one

specified in the `ejb-class` element for the particular bean in the `ejb-jar.xml` deployment descriptor file. The home and remote interface classes should not be mapped. Primary key classes, if they exist, also should not be mapped.

Creating mappings

You can create mappings by using TopLink Mapping Workbench or by using the Java code-based API. TopLink Mapping Workbench is a visual tool that offers windows and dialogs to set properties and to configure the mappings and TopLink descriptors for any given project. This is the preferred method of creating mappings, and should be used whenever possible.

TopLink Mapping Workbench imposes some limitations that require you to use the code API instead of the tool, but these limitations are few and are mentioned in the TopLink Mapping Workbench documentation.

For more information on the TopLink Mapping Workbench features and usage, and on the limitations mentioned above, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Direct mappings

Direct mappings define how a persistent object refers to objects that do not have TopLink descriptors, such as the JDK classes, primitive types and other non-persistent classes.

Attributes containing state that is a primitive object, or a regular object that is not itself mapped to the database should be mapped using a direct mapping. For example, a String attribute would need a direct to field mapping for the attribute to be stored in a VARCHAR field.

For a complete description of direct mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Entity bean attributes can be mapped using direct mappings without any special considerations.

Note: The entity context attribute (type `javax.ejb.EntityContext`) should not be mapped.

Relationship mappings

Persistent objects use *relationship mappings* to store references to instances of other persistent classes. The appropriate mapping type is based primarily upon the cardinality of the relationship (for example, one-to-one compared to one-to-many). For a complete description of relationship mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Entity beans may be related to regular Java objects, other entity beans, or both. The following sections outline the mappings and conditions where special attention must be paid to correctly map beans and execute operations that traverse or modify these relationships.

Mappings between entity beans

The EJB 2.0 specification introduces and defines the concept of relating beans to one another. It also imposes a number of restrictions on CMP relationships that TopLink does not enforce. Developers who wish to write their beans in such a way that they may be more easily migrated to full EJB 2.0 compliance may wish to follow some of the programming restrictions required by EJB 2.0, even if these restrictions are not enforced by TopLink.

TopLink support for the EJB 2.0 specification includes the following concepts:

- Bean relationships are managed automatically by the persistence layer, and do not require any internal use of finder methods.
- One-to-one, one-to-many and many-to-many relationships can be defined between beans.
- Dependent objects (regular Java objects) may be used to model fine-grained objects that are associated with a particular entity.

Some of the restrictions imposed by the EJB 2.0 specification that are not enforced by TopLink include:

- CMP beans must be abstract and have only “virtual” fields.
- Collections of entities used in relationship mappings must not be implemented by the bean developer, and must never be exposed directly to the client.
- Beans that are referenced by other beans must be related through local interfaces.

Additional restrictions to the mapping and run-time behavior of EJB 2.0 CMP beans are described in the EJB 2.0 specification (http://java.sun.com/products/ejb).

A bean that has a relationship to another bean acts as a "client" of that bean; that is, it does not access the actual bean directly but acts through the local interface of the bean. For example, if an `OrderBean` is related to a `CustomerBean`, it has an instance variable of type `Customer` (the local interface of the `CustomerBean`) and only accesses those methods defined on the Customer interface.

> **Note:** Although beans must refer to each other through their `local` interface, all TopLink descriptors and projects refer to the bean class. For example, if you are mapping beans using the TopLink Mapping Workbench and defining relationships between them, you need to load only the bean classes and not the `local`, `remote`, or `home` interfaces. When defining a relationship mapping in both the TopLink Mapping Workbench and code API, the "reference class" is always the bean class.

Importing relationship metadata in the Mapping Workbench In accordance with the EJB 2.0 specification, the Mapping Workbench can obtain relationship metadata from the `ejb-jar.xml` file. For more information on how to update TopLink relationships in the Mapping Workbench from the `ejb-jar.xml` deployment descriptor, see "Working with project properties" in the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Mappings between entity beans and Java objects

The EJB 2.0 specification notes that entity beans should represent "independent business objects" and that *dependent objects* are "better implemented as a Java class (or several classes) and included as part of the entity bean on which it depends."

The following relationship mappings may exist between an entity bean and regular Java objects:

- One-to-one, privately-owned mappings (bean is source, Java object is target)
- One-to-many, privately-owned mappings (bean is source, Java object(s) is target)
- Aggregate mappings (bean is source, Java object is target)
- Direct collection mappings (bean is source, Java object is target and is a "base" datatype, such as String, or Date)

Relationships from entity beans to regular Java objects should be dependent and relationships between entity beans should be independent.

If dependent objects are exposed to the client, these objects must be serializable.

One-to-one mappings

One-to-one mappings represent simple pointer references between two objects. For a complete description of one-to-one mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

One-to-one mappings are valid between entity beans, or between an entity bean and a regular Java object where the entity bean is the source and the regular Java object is the target of the relationship.

To maintain EJB compliance, the object attribute that points to the target of the relationship must be of the correct type if the target is a bean. This must be the local interface type and not the bean class.

There are a number of advanced variations on one-to-one mappings, that allow for more complex relationships to be defined — in particular *variable one-to-one mappings* allow for polymorphic target objects to be specified. These variations are not available for entity beans, but are valid for dependent Java objects. For more information on these kinds of mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

One-to-many mappings

One-to-many mappings are used to represent the relationship between a single source object and a collection of target objects. For more information on one-to-many mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

One-to-many mappings are valid between entity beans or between an entity bean and a collection of privately-owned regular Java objects.

As described in the *Oracle9iAS TopLink Mapping Workbench Reference Guide*, a one-to-one mapping should also be created from the target object back to the source. The object attribute that contains a pointer to the bean must be of the correct type (the local interface type) and not the bean class.

TopLink automatically maintains back-pointers as bi-directional relationships between beans are created or updated. See "Maintaining bi-directional relationships" on page 6-2.

Many-to-many mappings

Many-to-many mappings represent the relationships between a collection of source objects and a collection of target objects. They require the creation of an intermediate table for managing the associations between the source and target records. For more information on many-to-many mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

When using container-managed persistence, many-to-many mappings are valid only between entity beans and cannot be privately owned. The exception is when a many-to-many mapping is used to implement a logical one-to-many mapping with a relation table.

TopLink automatically maintains back-pointers as bi-directional relationships are created or updated. See "Maintaining bi-directional relationships" on page 6-2.

Aggregate object mappings

Two objects are related by aggregation if there is a strict one-to-one relationship between the objects and all the attributes of the second object can be retrieved from the same table(s) as the owning object. This means that if the target (child) object exists, then the source (parent) object must also exist. The child (owned object) cannot exist without its parent.

For a complete description of aggregate object mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide.*

Aggregate mappings can be used with entity beans when the source of the mapping is an entity bean and the target is a regular Java object. It is not valid to make an entity bean the target of an aggregate object mapping. As a consequence, it follows that aggregate mappings between entity beans are likewise invalid.

Note: Aggregate objects are privately owned and should not be shared or referenced by other objects.

Aggregate collection mappings

Aggregate collection mappings are used to represent aggregate relationships between a single source object and collection of target objects. Unlike normal one-to many mappings, there is no one-to-one back reference required. Unlike the normal aggregate mappings, a target table is required for the target objects.

For a complete description of Aggregate collection mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide.*

Aggregate collection mappings can be used with entity beans if the source of the relationship is an entity or Java object, and the targets of the mapping are regular Java objects. It is not possible to define an aggregate collection mapping with entity beans as the targets.

Aggregate collections are most appropriate when the target collections are expected to be moderate in size and a one-to-one mapping from target to source would be difficult. In addition, great care should be taken to ensure the identity of the Aggregate object, when referencing objects from an Aggregate within an Aggregate Collection.

Caution: Although aggregate collection mappings appear similar to one-to-many mappings, aggregate collections should not be used in place of one-to-many mappings. One-to-many mappings are more robust and scalable, and offer better performance. In addition, aggregate collections are privately owned by the source of the relationship and should not be shared or referenced by other objects.

Sequencing with Entity Beans

Sequencing is a mechanism which can be used to populate the primary key attribute of new objects/entity beans before inserting them into the database. Refer to the *Oracle9iAS TopLink Mapping Workbench Reference Guide* for details on the different kinds of TopLink sequencing: table and native.

The configuration of sequencing is similar for both Java objects and entity beans. However, with entity beans a `create()` method exists on the bean home interface, and `ejbCreate()` and `ejbPostCreate()` methods are implemented on the bean implementation class.

Because the primary key is automatically generated, no primary key is passed into the `create()` method on the home interface when the bean is created. If you are using table-based sequencing or native sequencing for databases that support pre-allocation of sequence numbers, the bean's primary key is available in the `ejbPostCreate()` method.

Inheritance

Although inheritance is a standard tool in object-oriented modeling, no implementation guidelines are outlined in the EJB specification. The EJB 1.0 specification does not address the issue, and the 1.1 and 2.0 specifications discuss it only in general terms. As a result, any use of inheritance should be approached cautiously.

Some restrictions apply to entity beans when using inheritance:

- The home interfaces cannot inherit. The `findByPrimaryKey` method must be overloaded in order to have the correct return type, but this is not allowed. As a result, inheritance is not applicable to the home interfaces.
- The primary key of the subclass must be the same as that of the parent class.

The advanced example application illustrates inheritance. For more information, see the ReadMe.html file in the root directory of the advanced example application. This application is located in

- `<`*`INSTALL_DIR`*`>\examples\wls70\examples\ejb\cmp20\advanced\` (WebLogic 7.0 and CMP 2.0)
- `<`*`INSTALL_DIR`*`>\examples\wls70\examples\ejb\cmp1.1\advanced\` (WebLogic 7.0 and CMP 1.1)
- `<`*`INSTALL_DIR`*`>\examples\wls61\examples\ejb\cmp20\advanced\` (WebLogic 6.1 and CMP 2.0)
- `<`*`INSTALL_DIR`*`>\examples\wls61\examples\ejb\cmp20\advanced\` (WebLogic 6.1 and CMP 1.1)

Indirection

TopLink provides several mechanisms for just-in-time reading of relationships (also referred to as “lazy-loading” and “indirection”). There are three techniques that are available:

- use of indirection objects
- transparent indirection
- proxy indirection

While these indirection mechanisms are described in the *Oracle9iAS TopLink Mapping Workbench Reference Guide*, there are a number of issues that entity bean developers should be aware of when using indirection. In general these issues arise due to the migration of objects between client and server.

Issues include:

- Un-instantiated ValueHolders (indirection objects) do not survive serialization. If a ValueHolder is sent from the server to the client, it will no longer function unless it has been previously triggered.
- ValueHolders can be used in bean-bean relationships, and bean-object relationships, but should be avoided in relationships whose source is likely to be serialized to the client.
- Collections that use transparent indirection should not be serialized to the client application before they are instantiated. These collections will not function if they are serialized.
- Proxy indirection (available in JDK 1.3) cannot be used for relationships whose target is an entity bean. The proxies used for this kind of indirection will interfere with the RMI stubs and skeletons generated for the entity. Proxies should be instantiated before being serialized to the client.
- ValueHolders should generally be used for bean-bean relationships, and for bean-object relationships. Transparent indirection can be used for collections that are not exposed to the client application.

Under the EJB 2.0 specification, the indirection policies for CMP fields must be one of the following:

- Transparent indirection for 1-Many or Many-Many relationships
- Value holder indirection for 1-1 relationships

Because of the code-generated subclasses, all indirection is hidden from the user.

For more information about these and other important issues, consult "Run-time Considerations" on page 6-1.

3

Configuring TopLink Container-Managed Persistence

This chapter describes the configuration and testing of TopLink Container-Managed Persistence. Please refer to *Oracle9iAS TopLink Getting Started* for installation information.

Software requirements

TopLink Container-Managed Persistence requires:

- BEA WebLogic Server 6.1 (Service Pack 3) or 7.0
- A JDBC driver that is configured to connect with your local database system (see your database administrator)
- A Java development environment that is compatible with the JDBC API, such as:
  - Oracle JDeveloper
  - Sun JDK 1.3 or higher
  - Any other Java environment that is compatible with the Sun JDK 1.3 or higher
- A command-line Java virtual machine (VM) executable (such as `java.exe` or `jre.exe`)

Configuring TopLink CMP

This procedure configures TopLink Container-Managed Persistence Foundation Library. This procedure assumes you have already installed TopLink Container-Managed Persistence.

Notes:

- If you are running under Windows NT, make sure you have administrator privileges. Also, make sure you modify the System Variables, not the User Variables.
- Java package names are case-sensitive. If you are installing under a 32-bit Windows environment, ensure the case sensitivity is enabled.

The TopLink class library is certified 100% pure Java and can be run on any JDK 1.2 or higher Java VM/platform.

To configure TopLink:

1. Locate the WebLogic persistence directory, which is located above the installation drive and root directory of your BEA WebLogic executable, as follows:

WebLogic Version	Persistence directory (above <WebLogLogicINSTALL_DIR>)
6.1 (service Pack 3)	`\wlserver6.1\lib\persistence`
7.0	`\weblogic700\server\lib\persistence`

   Under Windows, if BEA WebLogic Server is already installed, the TopLink installer automatically copies a file called `persistence.install` to the WebLogic persistence directory.

   If BEA WebLogic Server is not yet installed, or if you are installing TopLink in a non-Windows environment, then you must either:

   - Open the persistence directory using a text editor and add a new line to the `persistence.install` file referencing `TopLink_CMP_Descriptor.xml`.

   or

- Replace your existing `persistence.install` file with the new `persistence.install` file found in the *<INSTALL_DIR>*`\wls_cmp` folder.

Note: Refer to *<INSTALL_DIR>*`\Index.html` for the most recent installation notes. If you installed to the default directory, *<INSTALL_DIR>* is `C:\{ORACLE_HOME}\toplink`.

2. If you are writing your own start script, you must ensure that the CLASSPATH includes all of the following

   <INSTALL_DIR>`\core\lib\toplink.jar; `*<INSTALL_DIR>*`\core\lib\xerces.jar; `*<INSTALL_DIR>*`\wls_cmp\lib\tl_wlsx.jar;`
   where *<INSTALL_DIR>* is the directory into which you installed TopLink (`C:\{ORACLE_HOME}\toplink` if you installed to the default directory).

   Note: When editing the CLASSPATH, ensure that `weblogic.jar` and `weblogic_sp.jar` are placed before the TopLink JAR files in the CLASSPATH statement.

Testing TopLink Container-Managed Persistence with entity beans

To test TopLink Container-Managed Persistence with entity beans, run the Single Bean example documented in Chapter 9, "The EJB 2.0 Single Bean Example Application". For detailed instructions on how to set up and run this example, see Chapter 9, "The EJB 2.0 Single Bean Example Application".

When the TopLink Mapping Workbench, and the Single Bean example all run successfully, your TopLink installation is complete.

Running the BEA WebLogic Server with TopLink

Start the server as described in the WebLogic documentation. The most reliable way to start a server is to create a startup script. Refer to the BEA WebLogic documentation for more information on class loader and classpath issues. Once the server is running, start the TopLink CMP application.

Note: If you encounter problems running BEA WebLogic Server, contact BEA's WebLogic support.

TopLink Container-Managed Persistence includes a sample domain used to deploy and run the example application. This start script is available in *<INSTALL_DIR>*\examples\ under

- wls6.1\Server\config\TopLink_Domain for WebLogic 6.1 (service Pack 3)
- wls7.0\Server\config\TopLink_Domain for WebLogic 7.0

If you write your own start script, ensure that the CLASSPATH includes all of the following

```
<INSTALL_DIR>\core\lib\toplink.jar; <INSTALL_DIR>\core\lib\xerces.jar;
<INSTALL_DIR>\wls_cmp\lib\tl_wlsx.jar;
```

where *<INSTALL_DIR>* is the directory into which you installed TopLink (C:\{ORACLE_HOME}\toplink if you installed to the default directory).

> **Note:** Ensure that weblogic.jar and weblogic_sp.jar are placed before the TopLink JAR files in the CLASSPATH.

If a security manager is used, you must specify a security policy file.

```
-Djava.security.manager
-Djava.security.policy==c:\weblogic\weblogic.policy
```

A sample security policy file is supplied with the BEA WebLogic installation procedure.

Note that you will have to edit the weblogic.policy file (normally located in the BEA WebLogic install directory) to grant permission for TopLink to use reflection. To do this, add the following line to the file in the general "grant" section.:

```
permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
```

A subset of a "grant" section taken from a weblogic.policy file

```
grant
// Permission "enableSubstitution" needed to run the WebLogic console
permission java.io.SerializablePermission "enableSubstitution";
// Permission "modifyThreadGroup" required to run the WebLogic Server
permission java.lang.RuntimePermission "modifyThreadGroup";
permission java.lang.RuntimePermission
//... bulk of permissions not shown ...
//required for toplink
    permission java.lang.reflect.ReflectPermission
    suppressAccessChecks";
};
```

Configuration troubleshooting

If after successfully installing TopLink, you encounter problems running TopLink, one or more of the following suggestions may help resolve the problem:

- Ensure that the `persistence.install` file in the WebLogic installation contains the required `TopLink_CMP_Descriptor.xml` entry
- Some database servers (such as DB2) require an extra Windows NT service to be running for JDBC connections. If your server has been rebooted, this service may not have been restarted.
- If the BEA WebLogic Server is installed to a directory other than the default directory, ensure that the `weblogic.policy` file is edited to reflect this.

See *Oracle9iAS TopLink Troubleshooting* for more information.

4

EJB Entity Bean Deployment

TopLink Container-Managed Persistence provides container-managed persistence (CMP) for 1.1 and 2.0 Enterprise JavaBeans (EJBs). The deployment process generates CMP code that allows TopLink to handle persistence aspects of EJBs. To install entity beans within the BEA WebLogic Server and make them available for client applications, entity beans must be *deployed* within the server.

Overview of deployment

The goal of deployment is to make entity beans available to client applications.The process of deploying entity beans requires the use of several BEA WebLogic tools, the creation or editing of deployment descriptors and properties files, and ultimately running the BEA WebLogic Server to deploy the entity beans. For much of this process, BEA WebLogic tools and programs must be used. Consult your BEA WebLogic Server documentation for the most up-to-date information on these topics.

Understanding Deployment

The term "deployment" can sometimes cause confusion since there are actually a number of stages that occur between creating the bean classes and installing them in a running server.

Generally speaking the deployment process is three distinct steps:

1. **Configuration** - A number of properties are specified for the bean, including what persistence mechanism is being used and additional information required by the persistence mechanism.
2. **Code generation -** The information provided in the configuration stage is used by both BEA WebLogic and TopLink tools to generate the classes required for

the bean. This includes helper classes related to transactions, persistence, and security, the `EJBLocalHome`, `EJBLocalObject`, `EJBHome`, and `EJBObject` implementations, and the stubs and skeletons required for RMI.

3. **Installation -** The server is started and instructed to make the bean available to clients.

Requirements before deployment

The following tasks must be completed prior to the deployment of TopLink persisted entity beans:

- Write and compile the various parts of each entity bean to be deployed, including the bean class, any required local and remote interfaces, home and local home interfaces, and the primary key class (if required).
- Map the entity beans to the appropriate database tables, and save the mapping information in a TopLink project class or project file (deployable XML file).

Steps in the deployment process

After the beans have been created and the mappings have been defined, the three deployment steps must be followed:

1. **Configure the beans.** Ensure that the XML deployment descriptors have all of the information that BEA WebLogic and TopLink require.
2. **Generate the required run-time classes.** Classes are generated using the `weblogic.ejbc` utility.

> **Note:** Generating the required run-time classes is optional when you are using WebLogic Server 6.1 (service Pack 3) or 7.0; if you deploy any uncompiled JAR files using WebLogic Server 6.1 (service Pack 3) or 7.0, the server runs ejbc for you automatically. Consult your WebLogic documentation for details.

3. **Install the beans in the server.**

This chapter describes setting up deployment descriptors, generating the run-time classes, and deploying your entity beans in the BEA WebLogic Server.

Configuring entity bean deployment descriptors

A TopLink project represents a single deployment unit. In other words, all related beans (beans that reference each other) and dependent objects within a given TopLink project must be deployed within a single EJB JAR.

There are three XML files to configure for every EJB JAR file that is to be deployed. These files can be created and edited using a text editor, or some other tool. These files must be properly specified to use container-managed persistence:

ejb-jar.xml Contains standard EJB deployment properties

weblogic-ejb-jar.xml Contains BEA WebLogic -specific properties

toplink-ejb-jar.xml Contains TopLink-specific properties

Related beans share the same `ejb-jar.xml`, `weblogic-ejb-jar.xml`, and `toplink-ejb-jar.xml` files.

Configuring the ejb-jar.xml file

There is one `ejb-jar.xml` file for every JAR, although multiple beans may be specified in a single `ejb-jar.xml` file. The following information is stored in the `ejb-jar.xml` file:

persistence-type Each entity specifies what type of persistence it is to use. To specify that an entity use container-managed persistence, the bean must have its `<persistence-type>` tag be set to Container.

Container managed fields Each entity must list the bean fields which are to be persisted. The `<cmp-field>` tag is used for each field.

Finders Finders are optional, and may be specified using the `<query>` tag and its subtags. An EJB-QL string is used to define the query. The finders found in the `ejb-jar.xml` file are read in by the Mapping Workbench. You can use the Mapping Workbench to further customize finders for your application.

Relationships Bean-to-bean relationships are described under the `<relationships>` tag. Each relationship requires an `<ejb-relation>` tag and associated subtags. The Mapping Workbench uses these relationship descriptions as a starting point for defining the reference mappings between beans. Note that the Mapping Workbench can be used to define other relationships between beans or

between beans and regular Java objects which are not described in the `ejb-jar.xml` file.

Updating the ejb-jar.xml file

The `ejb-jar.xml` may either be manually updated as you develop your application or modified by the Mapping Workbench after mapping the TopLink project. The `ejb-jar.xml` file and the Mapping Workbench project should be synchronized as follows:

- When manual changes are made directly in the `ejb-jar.xml` file, refresh the Mapping Workbench project by re-importing the `ejb-jar.xml` file into the project.
- When changes to the project are made in the Mapping Workbench, the `ejb-jar.xml` file is automatically updated when the project is saved.

For more information on managing the ejb-jar.xml file in the Mapping Workbench, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Configuring the weblogic-ejb-jar.xml file

The `weblogic-ejb-jar.xml` file allows you to configure WebLogic-specific information for your beans.

Persistence descriptor

Within the weblogic-ejb-jar.xml file, each bean must have a `<persistence-descriptor>` entry with subentries indicating that TopLink is available and should be used.

- If you are deploying to WebLogic 6.1 (service Pack 3), there must be a `<persistence-type>` entry, which indicates that TopLink is available and a `<persistence-use>` entry that specifies that TopLink is to be used.
- If you are deploying to WebLogic 7.0, the entries under `<persistence-use>` includes both of the required pieces of information.

The persistence descriptor includes settings for specifying the EJB version supported by the bean. Set `<type-identifier>` to

- `TopLink_CMP_2_0` to indicate support for EJB 2.0
- `TopLink_CMP_1_1` to indicate support for EJB 1.1

The persistence descriptor also includes settings for specifying the WebLogic version supported by the bean. Set <type-version> to

- 4.5 to indicate support for WebLogic 7.0
- 4.0 to indicate support for WebLogic 6.1 (service Pack 3)

> **Note:** The deprecated <type-version> setting of version 3.5 will also function correctly with WebLogic 6.1 (service Pack 3) if used with EJB 1.1.

Persistence descriptors and WebLogic 6.1 (service Pack 3) A typical persistence descriptor for WebLogic 6.1 (service Pack 3) is shown below:

```
<persistence>
    <persistence-type>
        <type-identifier>TopLink_CMP_2_0</type-identifier>
        <type-version>4.0</type-version>
        <type-storage>META-INF\toplink-ejb-jar.xml</type-storage>
    </persistence-type>
        <persistence-use>
            <type-identifier>TopLink_CMP_2_0</type-identifier>
            <type-version>4.0</type-version>
        </persistence-use>
</persistence>
```

There may be several entries for the persistence type, but exactly one persistence use entry must be specified for a given entity bean.

Persistence descriptors and WebLogic 7.0 A typical persistence descriptor for WebLogic 7.0 is shown below:

```
<persistence>
    <persistence-use>
        <type-identifier>TopLink_CMP_2_0</type-identifier>
        <type-version>4.5</type-version>
        <type-storage>META-INF\toplink-ejb-jar.xml</type-storage>
    </persistence-use>
</persistence>
```

It is very important that the <type-version> entry is consistent with the installed version of TopLink (TopLink_CMP_2_0 for 2.0 entity beans and TopLink_CMP_1_1 for 1.1 entity beans). The <type-version> must be correct in order to use BEA WebLogic tools, and to deploy the bean in the BEA WebLogic Server.

Within the persistence type, the <type-storage> should be set to META-INF\toplink-ejb-jar.xml. The toplink-ejb-jar.xml contains

TopLink-specific information and is stored in the `META-INF` directory in the deployable JAR file.

Enabling Call by Reference

To allow TopLink to manage relationships between beans, *call-by-reference* must be enabled for all beans managed by TopLink. To enable call-by-reference for beans you must set the entry `<enable-call-by-reference>` to `True` in the `weblogic-ejb-jar.xml` file.

```
<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <ejb-name>AccountBean</ejb-name>
        ...
        <enable-call-by-reference>True</enable-call-by-reference>
        ...
    </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Unsupported tags in the weblogic-ejb-jar.xml file

There are a number of tags included in the `weblogic-ejb-jar.xml` that are either not supported or not required by TopLink, as follows:

concurrency-strategy In BEA WebLogic Server 6.1 (service Pack 3) and 7.0, the weblogic-ejb-jar.xml includes a <concurrency-strategy> tag that does not apply to entity beans deployed with TopLink for WebLogic. TopLink supports concurrent access to entities through the use of several locking options, including no locking, optimistic database-level locking, and pessimistic database-level locking. For more information please refer to the TopLink documentation.

db-is-shared TopLink does not require this tag because TopLink does not make any assumptions about the exclusivity of database access. Issues arising from multi-user access can be addressed through various locking and refreshing policies.

delay-updates-until-end-of-tx TopLink always delays updates until the end of a transaction, and so does not require this tag. This optimization allows for minimal write calculations.

finders-load-bean TopLink always loads the bean upon execution of the finder. This optimization eliminates excessive select calls during the ejbLoad sequence.

pool TopLink for WebLogic does not use a pooling strategy for entity beans. This avoids object-identity problems that can occur due to pooling.

lifecycle This element applies to beans that follow a pooling strategy. As TopLink does not use a pooling strategy, this tag is not required.

is-modified-method-name TopLink does not require a bean developer-defined method to detect changes in object state. This is handled automatically.

isolation-level Isolation level settings for the cache or database transactions can be specified in the TopLink project.

cache TopLink caching properties are defined using the TopLink Mapping Workbench.

Configuring the toplink-ejb-jar.xml file

The `toplink-ejb-jar.xml` file is similar to the `session.xml` file used by non-CMP beans. The `toplink-ejb-jar.xml` file specifies all TopLink-related information. The DTD for this XML file is located in the `tl_wlsx.jar` file, and is documented in Appendix B, "The toplink-ejb-jar DTD".

```
<?xml version="1.0" ?>
<!DOCTYPE toplink-ejb-jar (View Source for full doctype...)>
<toplink-ejb-jar>
    <session>
        <name>ejb_AccountDemo</name>
        <project-xml>Account.xml</project-xml>
        <login>
            <datasource>jdbc/ejbJTSDataSource</datasource>
            <non-jts-datasource>jdbc/ejbNonJTSDataSource</non-jts-datasource>
        </login>
    </session>
</toplink-ejb-jar>
```

Defining required project options: the Session Section

The session section is used to define those settings that apply to an entire project, and must be included in the `toplink-ejb-jar.xml` file. The xml elements that are defined in the session section are as follows:

name A string that provides a unique name for the project. No two projects deployed within the same server instance are permitted to have the same name.

project-class The fully qualified name of the TopLink project class. The source code for this class may be created using the TopLink Mapping Workbench and included in the deployed JAR.

project-xml The fully qualified name of the deployable TopLink project file. This file may be included in the deployed JAR or located on the CLASSPATH

> **Note:** Use either `project-xml` or `project-class` but do not use both.

login A section in which the following entries are provided:

- **`connection-pool`** A string identifying the JDBC pool that is to be used by TopLink. The name of the pool should correspond to a JDBC connection pool specified in the WebLogic administration console.
- **`datasource`** A string identifying the name of the data source to use for this project. `datasource` must be used in conjunction with `non-jts-datasource`. The combination is intended as an alternative to using a `connection-pool`.

  `datasource` is intended to map to a Jts data source, and `non-jts-datasource` is intended to map to a non-Jts data source.

  For more information about data sources, see your user documentation for BEA WebLogic Server.
- **`non-jts-datasource`** A string identifying the name of the read only data source to use for this project. non-jts-datasource must be used in conjunction with datasource. The combination is intended as an alternative to using a connection-pool. See "datasource" earlier for more information.
- **`should-bind-all-parameters (optional)`** A string value that indicates whether all queries should use parameter binding. Valid values are *True* or *False*. Default is *False*.
- **`uses-byte-array-binding (optional`**) A string value that indicates whether byte arrays should be bound. Valid values are *True* or *False*. Default is *False*.
- **`uses-string-binding (optional)`** A string value that indicates whether strings should be bound. Valid values are *True* or *False*. Default is *False*.

cache-synchronization (optional) When provided, indicates that changes made to one TopLink cache in a cluster should be automatically propagated to all other server caches. The following elements may also be provided:

- **`is-asynchronous`** (optional) Set to *True* if synchronization should not wait until all sessions have been synchronized before returning. Valid values are *True* or *False*. Default is *True*.
- **`should-remove-connection-on-error`** (optional) Set to *True* if a synchronization connection should be removed from the session if a communication error occurs. Valid values are *True* or *False*. Default is *True*.

use-remote-relationships (optional) TopLink goes beyond the EJB 2.0 specification and allows you to define relationships between beans in terms of their remote interfaces. This may be especially useful when porting EJB 1.1 applications to EJB 2.0. When this option is specified, all relationships in the JAR must be defined using remote interfaces. Valid values are *True* or *False*. Default is *False*.

> **Note:** If you use remote relationships, you must run the weblogic.ejbc tool with the `-nocompliance` flag set.

customization-class (optional) This fully qualified name of a `DeploymentCustomization` class is optional. For more information, see Chapter 7, "Customization".

Generating the run-time classes

After creating the deployment descriptors, generate the run-time classes from the deployment descriptors using of the **ejbc** tool.

> **Note:** Generating the required run-time classes is optional when you are using WebLogic Server 6.1 (service Pack 3) and 7.0; if you deploy any uncompiled JAR files using WebLogic Server, the server runs ejbc for you automatically. Consult your WebLogic documentation for details.

Running the Weblogic EJB Compiler

When TopLink is used to provide container-managed persistence for BEA WebLogic entity beans, the standard BEA WebLogic EJB Compiler (**ejbc**) is used. For more information about this tool and its use, refer to the BEA WebLogic Server documentation.

There are two stages involved in running **ejbc**:

1. Create a "standard" EJB JAR file containing all bean classes and all required XML files (`ejb-jar.xml`, `weblogic-ejb-jar.xml`, `toplink-ejb-jar.xml`). The XML files should be placed in the META-INF directory within the JAR.
2. Run **ejbc** with the JAR file created in step one as command line arguments. **ejbc** creates an EJB JAR containing the original classes as well as all required generated classes and files.

Several things happen when **ejbc** is run:

- A partial EJB conformance check is performed on the beans and their associated interfaces
- A number of internal BEA WebLogic classes that manage security, transactions, and so on, are generated and compiled
- Concrete bean subclasses are generated by TopLink and compiled
- RMI stubs and skeletons are generated for client access of the beans

Running **ejbc** can take some time because of the number of the processes involved. If errors occur while running **ejbc**, attempt to determine which stage is causing the problem. Problems encountered running this tool may be related to one of the following areas:

- Errors or problems in the bean classes caused by non-conformance with the EJB specification
- Problems due to not having all required classes on the CLASSPATH, which should include all domain classes, all required TopLink classes, and all required BEA WebLogic classes
- A problem encountered when running the Java compiler (`javac`), potentially caused by using an incorrect version of the JDK
- A failure encountered when generating the RMI stubs and skeletons (a failure of `rmic`)

Note: Use a command script (for example, a batch or ant script) to run **ejbc**. This enables you to pre-configure all of the required variables for the command line and helps to prevent typing errors. Sample build scripts are provided with the TopLink for BEA WebLogic example applications.

Installing the beans in the server

Installing entity beans that use TopLink for BEA WebLogic container-managed persistence follows the same steps as installing other EJBs in the BEA WebLogic Server. For information on installing entity beans, consult the BEA WebLogic documentation.

Connection pools and data sources

BEA WebLogic can provide TopLink with either a connection pool or a data source.

Creating JDBC connection pools

A BEA WebLogic JDBC connection pool must be defined for the entity beans that are to be deployed. Examples of how connection pools are defined and used can be found in your BEA WebLogic Server documentation.

Creating JTS and non-JTS data sources

To work with data sources, TopLink requires both a JTS and a non-JTS data source. When working with a connection pool, Toplink configures the connection automatically, and only needs a single connection pool to operate properly

Please consult the BEA WebLogic Server user documentation for more information about data sources.

Using the defined connection pool

The pool name is supplied to TopLink using the `toplink-ejb-jar.xml` file. The name is specified using the <connection-pool> tag (see "login" on page 4-8).

Using the defined data source

Instead of connecting through a connection pool directly, it is possible to instruct TopLink to use a defined data source. If data sources are to be used, both JTS and non-JTS data sources must be used together.

The `toplink-ejb-jar.xml` file has data-source options in the form of the tags `<datasource>` and `<non-jts-data-source>`. These tags correspond to JTS and non-JTS data sources respectively.

The values for these data source tags correspond directly to the names of the data sources as defined in WebLogic Server. Following is an example of a partial `toplink-ejb-jar.xml` file listing using data sources:

```
...
<datasource>myJtsDataSource</datasource>
<non-jts-data-source>myNonJtsDataSource</non-jts-data-source>
...
```

Problems with deployment

Various configuration errors can cause problems with entity bean deployment. Be sure that you have followed the steps outlined in the relevant BEA WebLogic Server documentation.

Message Logging

The logging facilities are configured at the server level and affect all TopLink-enabled CMP entity beans deployed in that server. The granularity and destination of TopLink's logging can be configured by using two system properties. These properties can be set at the command line of the server start script.

toplink.log.level Determines the granularity of log messages that TopLink will generate. Possible values include `INFO` (default), `NONE` and `DEBUG`. When set to `DEBUG`, TopLink trace statements will be generated. For example:

```
-Dtoplink.log.level=DEBUG
```

toplink.log.destination Determines where TopLink log messages get logged. Possible values include SYSOUT (default), SERVER or the name of a file.

When set to SYSOUT log messages are sent to System.out.

When set to SERVER, logging is integrated with the WebLogic logging streams. If this mode is in effect then the WebLogic logging levels must be considered since logging may be filtered by the server log level settings.

When neither of the above is selected, the value is assumed to be the name of a file (relative or fully qualified). Consider these examples:

```
-Dtoplink.log.destination=SERVER
```

```
-Dtoplink.log.destination=c:\toplink.log
```

Hot deployment of EJBs

Hot deployment is a feature in BEA's WebLogic Server product that allows for the deployment of EJBs on a running server. It is useful for situations where rebooting the BEA WebLogic Server is not feasible.

The BEA WebLogic Server hot deployment feature allows:

- Newly-developed EJBs to be deployed to a running production system
- Deployed EJBs to be removed from a running server (undeployment)
- The behavior of deployed EJBs to be modified by updating the bean class definition (redeployment)

For detailed information on hot deployment, see the BEA WebLogic documentation.

Consider the following points when deploying a newly-created EJB JAR or redeploying an existing JAR:

- All related beans (all beans that share a common TopLink project) must be deployed within the same EJB JAR file. TopLink for BEA WebLogic views deployment on a project level. If one bean is to be deployed or updated, then all of the project's beans should be deployed or updated to maintain consistency across the project.
- When a bean is redeployed, its TopLink project is reset. This process flushes all object caches and rolls back any active object transactions that are associated with the project.

The client receives deployment exceptions when attempting to access bean instances that have been undeployed or re-deployed. The client side is responsible for catching and handling those exceptions.

Running an EJB Client

After the beans have been deployed, an EJB client can be run to access them. The client can either be a SessionBean or a Java program running outside the server.

The EJB client requires the following bean classes in its CLASSPATH: remote interface, home interface, and primary key classes for all the beans accessed. In addition, if the client is a session bean running on the same server as the entity beans and you want to access the local interfaces of the entity beans, you must also include their local and home interfaces on the CLASSPATH.

To lookup a bean's home interface a JNDI InitialContext must be setup. Setting up the initial context requires that the server's URL be supplied.

5

Defining and Executing Finders

TopLink provides a feature-rich query framework in which complex database queries can be constructed and executed to retrieve entity beans. TopLink Container-Managed Persistence enables you to define the finder methods on the home interface, but does not require you to implement them in the entity bean. TopLink Container-Managed Persistence provides this required functionality, and offers a number of strategies for creating and customizing finders. The EJB container and TopLink automatically generate the implementation.

Defining finders in TopLink

The general steps required to successfully define a finder method for an entity bean using TopLink Container-Managed Persistence's query framework are as follows:

1. Declare finders in the `ejb-jar.xml` file.
2. Define the finder method on the entity bean's home and/or local home interface(s) (as required by the EJB specification)
3. Use the Mapping Workbench to change any options on finders.
4. If required, create an implementation for the query. Some query options require that the query be defined in code on a helper class, but this is not required for most queries.

ejb-jar.xml Finder Options

The `ejb-jar.xml` file specifies all of the EJB 2.0 specification related information for a bean, including the definitions for any finders that are to be used for that bean. The `ejb-jar.xml` file may be created and edited using a text editor, or it may be created using the Mapping Workbench. All finders are defined within the `ejb-jar.xml` file with a structure similar to the following example.

Example 5–1 A simple finder within an ejb-jar.xml file

```
...
<query>
    <query-method>
        <method-name>findLargeAccounts</method-name>
            <method-params>
               <method-param>double</method-param>
            </method-params>
    </query-method>
<ejb-ql><![CDATA[SELECT OBJECT(account) FROM AccountBean account WHERE
account.balance > ?1]]></ejb-ql>
</query>
...
```

Query Section - XML Elements

The `ejb-jar.xml` file can contain zero or more `<query>` elements in the `<entity>` tag. Each one of these `<query>` tags corresponds to a finder method that is defined on the bean's home or local home interface. If you define the same finder (same name, return type, and parameters) on both home interfaces, then only a single `<query>` element is defined in the ejb-jar.xml file and they must share the same TopLink query.

The elements that are defined in the `<query>` section of the `ejb-jar.xml` file are:

- **`description`** (optional)- Used to provide text describing the finder.
- **`query-method`** - Used to specify the method for a finder or ejbSelect query.
- **`method-name`** - Specifies the name of a finder or select method in the entity bean's implementation class.
- **`method-params`** - Contains a list of the fully-qualified Java type names of the method parameters.
- **`method-param`** - Contains the fully-qualified Java type name of a method parameter.
- **`result-type-mapping`** (optional)- Used in the query element to specify whether an abstract schema type returned by a query for an ejbSelect method is to be mapped to an EJBLocalObject or EJBObject type. Valid values are Local or Remote
- **`ejb-ql`** - Used for all finders that can be expressed using EJB QL. It contains the EJB QL query string that defines a finder or ejbSelect query. This parameter is left empty for non-EJBQL finders.

Choosing the best finder type for your query

TopLink supports five general types of finders:

- EJBQL queries
- Expressions built using the TopLink expression framework
- Dynamic queries
- redirect queries
- SQL queries

Most finders can be defined using the EJBQL mechanism. However, the other mechanisms have their own advantages:

- Expression finders give the user access to TopLink's expression framework.
- If dynamic querying is required (the query logic must be defined at run time) then one of the Dynamic types may be used.
- SQL and redirect queries are provided for those rare cases where no other mechanism is suitable.

For more information about defining finders, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Using EJBQL

EJBQL is the standard query language defined in the EJB 2.0 specification and is available for use in TopLink with both 1.1 and 2.0 beans. EJBQL finders enable a specific EJBQL string to be specified as the implementation of the query.

Advantages EJBQL offers several advantages in that it:

- is the EJB 2.0 standard for queries
- can be used for most queries
- can be used in dependent object queries

Disadvantages Some complex queries may be difficult to define using EJBQL.

Creating an EJBQL finder

To create an EJBQL finder

1. Declare the finder in the `ejb-jar.xml` and enter the EJBQL string in the `ejb-ql` tag.
2. Declare the finder on the Home interface, the LocalHome interface, or both as required.
3. Start the Mapping Workbench.
4. Specify the `ejb-jar.xml` location and select **File** > **Updated** Project from ejb-jar.xml to read in the finders.
5. Go to the **Queries** > **Named Queries** tab for the bean.
6. Select and configure the finder.

Following is an example of a simple EJBQL query that takes one parameter. In this example, the question mark ("?") is used to bind the argument name within the EJBQL string.

```
SELECT OBJECT(employee) FROM Employee employee WHERE (employee.name =?1)
```

Notes:

- The argument (**bolded** in the example) must be a numeric value.
- `Employee` (**bolded** in the example) refers to the `<abstract-schema-name>` defined for that particular bean.

For more information on EJBQL, see the *Oracle9iAS TopLink Foundation Library Guide*.

Using the TopLink Expression framework

Finders can take advantage of TopLink's rich expressions framework to define the logic of the query.

Advantages Using TopLink expressions to access the database has some advantages over using EJBQL:

- Standardizes queries to Java code, which can be version controlled.
- Can simplify complex operations.
- Fuller set of querying features than is available through EJBQL.

Example A sub-query expression using a comparison and count operation

This code queries all employees that have more than 5 managed employees.

```
ExpressionBuilder emp = new ExpressionBuilder();
ExpressionBuilder managedEmp = new ExpressionBuilder();
ReportQuery subQuery = new ReportQuery(Employee.class, managedEmp);
subQuery.addCount();
subQuery.setSelectionCriteria (managedEmp.get("manager").equal(emp));
Expression exp = emp.subQuery(subQuery).greaterThan(5);
```

> **Note:** This type of query is only possible using the TopLink expression framework.

Disadvantages The disadvantages to using TopLink Expressions in finders are:

- The Mapping Workbench does not support the TopLink Expression framework, so the Expression finders must be created in code using a project class or descriptor amendment.

Creating an Expression Finder

1. Declare the finder in the ejb-jar.xml and leave the ejb-ql tag empty. This step is optional but should be performed in order to maintain compliance with the EJB 2.0 spec.
2. Declare the finder on the Home interface, the LocalHome interface, or both as required.
3. Create an amendment method as described in "Creating amendment methods for Expression finders" on page 5-7.
4. Start the Mapping Workbench.
5. Select **Advanced Properties** > **After Load** from the menu for the bean.
6. Enable the amendment method for the descriptor by specifying the class and name of the static method.

Building an expression

An example of an expression is as follows:

```
builder.get("address").get("city").equal(theCity);
```

This represents the query logic, or the "selection criteria" for the query. The logical translation for this query is:

> Find all Employees whose address attribute's city attribute is equal to the value passed in as an argument when the finder method is invoked.

To introduce the basics of constructing a query expression, examine each element of this expression:

- **`builder`**: Represents an instance of TopLink's ExpressionBuilder (`oracle.toplink.expressions.ExpressionBuilder`), and represents the entry point for defining an expression.
- **`get`**: The **`Get`** predicate allows comparison of attributes within the expression (in this case, the `city` attribute of the entity bean's `address` attribute). If the attribute is
  - a target of a 1:1 relationship, the relationship is traversed and subsequent predicates can access attributes of the related object. For example:
  - get("address").get("city")
  - a target of a one-to-many (1:M) or many-to-many (M:M) relationship, the predicate `anyof` can be used to apply selection criteria to any of the related objects, resulting in multiple objects returned `equal(theCity)`, where the `equal` operator describes the operation to be applied on the attribute.

  In this case, `city` is compared based on equality to the parameter `theCity`, which represents the first argument passed into the finder method.
- **`equal(theCity)`**: The `equal` qualifier represents the string or value against which records are being compared. The equal operator describes the operation to be applied on the attribute. In this case, city is compared based on equality to the parameter `theCity`, which represents the first argument passed into the finder method.
- **`("address") and ("city")`**: These components represent nested attributes within the data. Multiple `get` predicates indicate a logical comparison that mirrors the structure of the data. Consider the two `get`s in the example:

  ```
  builder.get("address").get("city")
  ```

The first `get` in this statement specifies the `address` attribute, while the second `get` examines `city`, which is an attribute of the attribute, `address`.

Creating amendment methods for Expression finders

A TopLink Expression query must first be implemented and then registered with the runtime within a "TopLink descriptor amendment" method. Define the named query in the static amendment method, and add the query to the TopLink descriptor's QueryManager. The named query must be defined based on the following:

- The name of the query must match the name of the finder on the home interface.
- If the return type for the finder method on the home interface is `java.util.Collection`, then the query object defined must be a `oracle.toplink.queryframework.ReadAllQuery`.
- If the return type is an entity bean's remote or local interface (that is, only a single entity bean is returned), then the query must be of type `oracle.toplink.queryframework.ReadObjectQuery`.
- The reference class must be the bean class against which the finder is querying.
- The arguments defined in the query must exactly match the parameter name and type of the corresponding finder declared on your home interface. This includes the argument names, the number of arguments, as well as the order in which they occur.

For more information on configuring an amendment method, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Using Dynamic finders

The EJB 2.0 specification allows for finders defined in the `ejb-jar.xml` file as queries, with their search criteria specified as EJBQL query strings. TopLink expands on the specification, enabling you to create queries using other query formats such as SQL, expressions, dynamic query objects, and Redirects (see "Choosing the best finder type for your query" on page 5-3).

In addition to this support, TopLink provides a number of predefined finders that can be used for executing dynamic queries (queries for which the logic is determined by the user at run-time). The names for these finders are reserved by the TopLink runtime and cannot be reused for other finders.

The predefined finders are:

- `EJBObject findOneByEJBQL(String ejbql, Vector args)`
- `Collection findManyByEJBQL(String ejbql, Vector args)`
- `EJBObject findOneBySQL(String sql, Vector args)`
- `Collection findManyBySQL(String sql, Vector args)`
- `EJBObject findOneByQuery(DatabaseQuery query, Vector args)`
- `Collection findManyByQuery(DatabaseQuery query, Vector args)`

Each of these finders can also be used without the Vector of arguments. For example, `EJBObject findOneByEJBQL(String ejbql)` is a valid dynamic finder. The return type of "EJBObject" is replaced by the component interface of your bean.

Creating a Dynamic finder

To create a Dynamic finder

1. Declare the finder in the `ejb-jar.xml` file and leave the ejb-ql tag empty.
2. Declare the finder on the Home interface, the LocalHome interface, or both as required.
3. Start the Mapping Workbench.
4. Specify the `ejb-jar.xml` location and select **File** > **Updated** Project from ejb-jar.xml to read in the finders.
5. Go to the **Queries** > **Named Queries** tab for the bean.
6. Select and configure the finder.

Notes:

- If the advanced query options described in "Advanced finder options" on page 5-14 are not required, only steps 1 and 2 needs to be completed.
- The `findOneByQuery` and `findManyByQuery` dynamic finders should not have any query options configured for them. The reason is the query is created at runtime by the client and passed as a parameter to the finder. Any query options that you wish to set should be done on that query.

Using findAll

Like the dynamic finders, the name `findAll` is reserved by the TopLink runtime and cannot be reused for other finders. For more information on defining and configuring the finder, see "Creating a Dynamic finder" on page 5-8.

Using findByPrimaryKey

The `findByPrimaryKey` finder is always created in the Mapping Workbench on the initial loading of a bean class. Like other finders, the `findByPrimaryKey` finder can be configured with the various query options that TopLink provides (see "Advanced finder options" on page 5-14) but can also be deleted from the Mapping Workbench project. In this case, however, a warning is issued informing the user that the default container `findByPrimaryKey` options will be active. The EJB 2.0 specification requires that the `findByPrimaryKey` call is present on the home interface, but should not have a query entry in the `ejb-jar.xml` file.

Using redirect finders

Redirect finders enable you to specify a finder for which the implementation is defined in code as a static method on an arbitrary helper class. When the finder is invoked, the call is re-directed to the specified static method.

The finder can have any arbitrary parameters or none at all. If the finder includes parameters, they are packaged into a vector and passed to the redirect method.

Advantages Redirect finders provide client parameter-passing flexibility. Compared to other finder types in which the parameters are relatively simple objects used to match against an entity bean's attributes, redirect finders may include arguments that are not linked to these values, because the finder implementation is completely defined by the bean developer. The redirect method typically contains the logic required to extract the relevant data from the parameters and use it to construct a TopLink query.

Disadvantages Redirect queries are complex and often more difficult to configure. they also require an extra helper method to define the query.

To create a redirect finder

1. Declare the finder in the `ejb-jar.xml` leaving the `ejb-ql` tag empty.
2. Declare the finder on the Home interface, the localHome interface, or both as required.

3. Create an amendment method (see "Creating amendment methods for Expression finders"). on page 5-7
4. Start the Mapping Workbench.
5. Select **Advanced Properties** > **After Load** from the menu for the bean.
6. Enable the amendment method for the descriptor by specifying the class and name of the static method.

The amendment method should then add a query to the descriptor's QueryManager as follows:

```
ReadAllQuery query = new ReadAllQuery();query.setRedirector(new
MethodBaseQueryRedirector (examples.ejb.cmp20.advanced.
FinderDefinitionHelper.class, "findAllEmployeesByStreetName"));
descriptor.getQueryManager().addQuery ("findAllEmployeesByStreetName", query);
```

`examples.ejb.cmp20.advanced.FinderDefinitionHelper` includes a static method `findAllEmployeesByStreetName` **`(Session session, Vector args)`** which executes the query. It is up to the implementor of the query method to ensure that the proper types are returned. For methods returning more than one bean, the return type must be `java.util.Vector`. TopLink converts this result to `java.util.Enumeration` (or Collection) if required.

> **Note:** The redirect method also takes a TopLink Session as a parameter. For more information on TopLink Session, see "Database Sessions" in the *Oracle9iAS TopLink Foundation Library Guide*.

The redirect method must return either a single entity bean (Object) or a Vector. The possible method signatures are:

- `public static Object redirectedQuery2(oracle.toplink.sessions.Session s, Vector args)`
- `public static Vector redirectedQuery4(oracle.toplink.sessions.Session s, Vector args)`

Example 5–2 A simple Redirect query implementation:

```
public static Vector findAllEmployeesByStreetName(Session s, Vector args) {
    ReadAllQuery raq = new ReadAllQuery();
    raq = raq.setReferenceClass(EmployeeBean.class);
    raq.addArgument("streetName");
```

```
    ExpressionBuilder builder = newExpressionBuilder();
    .raq.setSelectionCriteria(builder.get("address")
     .get("street").equal(args.elmentAt(0)));
    return (Vector)s.executeQuery(raq);
}
```

At run time when the client invokes the finder from the entity bean's home, the arguments are automatically packaged into the args Vector (in order of appearance from the finder's method signature) for use within the static method. The code implementing the Redirect finder can then use any necessary APIs to extract information out of the arguments (once retrieved from the args Vector) for use within a TopLink expression.

Using SQL

SQL type finders allow a specific SQL string to be specified as the implementation of the query.

Advantages The advantages of using SQL include:

- It can be used if the query logic cannot be expressed using EJBQL or the TopLink expression framework.
- It allows for the use of a stored procedure instead of TopLink generated SQL
- There may be cases where custom SQL will improve performance.

Disadvantages This approach is generally not recommended if the query can be created using any of the other options, because

- Writing complex custom SQL statements requires a significant maintenance effort if the database tables change.
- The hard coded SQL limits portability to other databases.
- No validation is done on the SQL String, so if the SQL has errors, these will not be detected until run time.
- If the SQL does something other than SELECT, unpredictable errors may result.

Creating an SQL finder

To create an SQL finder

1. Declare the finder in the `ejb-jar.xml` and leave the `ejb-ql` tag empty.
2. Start the Mapping Workbench.
3. Specify the `ejb-jar.xml` location and select **File** > **Updated Project** from ejb-jar.xml to read in the finders.
4. Go the **Queries** > **Named Queries** tab for the bean.
5. Select the finder, check the SQL radio button and enter the SQL string.
6. Configure the finder.

Following is an example of a simple SQL finder that takes one parameter. In this example, the hash-character '#' is used to bind the argument `projectName` within the SQL string.

```
SELECT * FROM EJB_PROJECT WHERE (PROJ_NAME = #projectName)
```

Using ejbSelect

ejbSelects are similar to finders in function, but they can only be invoked from within a bean. Like finders, ejbSelects are defined in the `ejbjar.xml` using a `<query>` entry and can have various options configured using the Mapping Workbench. Also like finders, ejbSelects require a `SELECT` clause in addition to `FROM` and `WHERE` clauses.

However, ejbSelects differ from regular finders in the following ways:

- ejbSelects require an additional <result-type-mapping> tag, which specifies whether the return type is intended to be Local or Remote (the default is Local).
- ejbSelects are not defined on the bean home, but rather on the bean itself. These are not exposed to the client, but are used internally by bean business methods and by home methods.
- ejbSelect queries can return other return types other than the entity bean type on which they are invoked, and in fact may return any type corresponding to a container-managed relationship or container-managed field.

Understanding select methods

Select methods are query methods intended for internal use within an entity bean instance. Unlike finder methods, select methods are not specified in the entity bean's home interface but on the abstract bean itself.

The format for an ejbSelect method definition looks like this:

```
public abstract type ejbSelect<METHOD>(...);
```

The select method represents a query method that is not directly exposed to the client in the home or component interface. It is defined as being abstract, and each bean can include zero or more such methods.

Even though the select method is not based on the identity of the entity bean instance on which it is invoked, it can use the primary key of an entity bean as an argument to an `ejbSelect<METHOD>` to define a query that is logically scoped to a particular entity bean instance.

Select methods have the following characteristics:

- The method name must have ejbSelect as its prefix.
- It must be declared as public.
- It must be declared as abstract.
- The throws clause must specify the javax.ejb.FinderException, although it may also specify application-specific exceptions.

The return type for ejbSelects that return entities is determined by the `<result-type-mapping>` tag in the `ejb-jar.xml`. If the flag is set to `Remote`, then EJBObjects are returned; if set to `Local`, then `EJBLocalObjects` are returned.

Creating an ejbSelect

1. Declare the ejbSelect in the ejb-jar.xml, enter the EJBQL string in the `<ejb-ql>` tag, and specify the return type in the `<result-type-mapping>` tag (if required).
2. Declare the ejbSelect on the abstract bean class.
3. Start the Mapping Workbench.
4. Specify the `ejb-jar.xml` location and select **File** > **Updated Project** from ejb-jar.xml to read in the finders.
5. Go the **Queries** > **Named Queries** tab for the bean.
6. Select and configure the ejbSelect query.

Advanced finder options

There are a number of options that can be used by the experienced TopLink developer. These options should only be used when the developer has a complete understanding of the consequences of making changes to them.

Caching options

Various configurations can be applied to the underlying query to achieve the correct caching behavior for the application. There are several ways to control the caching options for queries.

For most queries, caching options can be set in the Mapping Workbench (see "Caching objects" in the *Mapping Workbench Reference Guide*).

The caching options can be set on a per-finder basis. The valid values are:

- `ConformResultsInUnitOfWork` (default): For finders returning a single result and finders returning a collection, the 'UnitOfWork' cache for the current JTS UserTransaction is queried. The finder's results will conform to uncommitted new objects, deleted objects and changed objects.
- `DoNotCheckCache`: For finders returning a single object and finders returning a collection, the cache is not checked.
- `CheckCacheByExactPrimaryKey`: If a finder returning a single object involves an expression that contains the primary key and only the primary key, the cache is checked.
- `CheckCacheByPrimaryKey`: If a finder returning a single object involves an expression that contains the primary key, a cache hit can still be obtained through processing the expression against the object in the cache.
- `CheckCacheThenDatabase`: A finder returning a single object queries the cache completely before resorting to accessing the database.
- `CheckCacheOnly`: For finders returning a single object and finders returning a collection, only the cache is checked; the database is not accessed.

For more information about TopLink queries as well as the TopLink UnitOfWork and how it integrates with JTS, see "Database Sessions" in the *Oracle9iAS TopLink Foundation Library Guide*.

Note: For finders whose queries are manually created (`findOneByQuery`, `findManyByQuery`), caching options must be applied manually using TopLink for Java APIs.

Disabling caching of returned finder results

By default, TopLink adds to the cache all returned objects whose primary keys are not currently in the cache. This can be disabled if the client knows that the set of returned objects is very large and wants to avoid the expense of storing these objects. This option is configurable through the Mapping Workbench or on the TopLink query API for queries using `dontMaintainCache()`.

Caching of returned finder results can also be disabled in the Mapping Workbench. For more information on disabling caching for returned finder results, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Refreshing finder results

A finder may return information from the database for an object whose primary key is already in the cache. When set to true, the refresh cache option in the Mapping Workbench indicates that the object's non-primary key attributes are refreshed with the returned information. This occurs on `findByPrimaryKey` finders as well as all EXPRESSION and SQL finders for that bean when set at the bean attributes level.

When refreshing is enabled, the `refreshIdentityMapResult()` method is invoked on the query. This is configured to automatically cascade private parts. If behavior other than private object cascading is desired, use a dynamic finder.

Caution: When issuing refreshing finders while in user transactions, refreshing the object may cause changes already made to that object during that transaction to be lost.

In the case where an `OptimisticLock` field is in use, the refresh cache option can be used in conjunction with the `onlyRefreshCacheIfNewerVersion()` option. In that case, the non-primary key attributes are refreshed only if the version of the object in the database is newer than the version in the cache.

For finders that have no refresh cache setting, the `onlyRefreshCacheIfNewerVersion()` method has no effect.

Managing large result sets

Finders can return large result sets which can be resource intensive to collect and process. To give the client more control over the returned results, TopLink finders can be configured to use cursors. This leverages TopLink's `CursoredStream` and a database's cursoring ability to break up the result set into smaller, more manageable pieces.

Building the query

Any finder that returns a `java.util.Enumeration` under EJB 1.1 or a `java.util.Collection` under EJB 2.0 can be configured to use a cursor. When the query is created for the finder, `useCursoredStream()` enables cursoring.

Example A query that uses a CursoredStream

```
ReadAllQuery raq = new ReadAllQuery();
ExpressionBuilder bldr = new ExpressionBuilder();
raq.setReferenceClass(ProjectBean.class);
raq.useCursoredStream();
raq.addArgument("projectName");
raq.setSelectionCriteria(bldr.get("name").
like(bldr.getParameter("projectName")));
descriptor.getQueryManager().addQuery ("findByNameCursored");
```

Executing the finder from the client in EJB 1.1

An extended protocol is available on the client in `oracle.toplink.ejb.cmp.wls11.CursoredEnumerator` (based on `java.util.Enumeration`):

hasMoreElements() As with `java.util.Enumeration`, this method returns a boolean indicating if any elements remain.

nextElement() As with `java.util.Enumeration`, this method returns the next available element.

nextElements(int count) Retrieve a `Vector` of at most count elements from the available results, depending on how many elements are left to read.

close() close the cursor on the server. It is mandatory that the client send this message when it is done with the results.

The behavior differs from a normal finder as follows:

- Only the elements requested by the client are sent to the client.
- Nothing is cached on the client in the `CursoredEnumerator`.
- If you are using the transactional attribute REQUIRED for your entity bean, all reads must be wrapped in a `UserTransaction begin()` and `commit()`. If not, reads beyond the first page of the cursor will have no transaction in which to work.

The following example illustrates client-code executing a cursored finder:

```
import oracle.toplink.ejb.cmpwaswls11. CursoredEnumerator;
//... other imports as necessary
getTransaction().begin();
CursoredEnumerator cursoredEnumerator = (CursoredEnumerator)getProjectHome()
 .findByNameCursored("proj%");

Vector projects = new Vector();
for (int index = 0; index < 50; i++) {
Project project = (Project)cursoredEnumerator.nextElement();
projects.addElement(project);
}
// Rest all at once ...
Vector projects2 = cursoredEnumerator.nextElements(50);
cursoredEnumerator.close();
getTransaction().commit();
```

Executing the finder from the client in EJB 2.0

An extended protocol is available for the client in `oracle.toplink.ejb.cmp.wls.CursoredCollection` (based on `java.util.Collection`):

isEmpty() As with `java.util.Collection`, `isEmpty()` returns a boolean indicating if the Collection is empty or not.

size() As with `java.util.Collection`, `size()` returns an integer which is the number of elements in the Collection.

iterator() As with `java.util.Collection`, `iterator()` returns a `java.util.Iterator` for enumerating the elements in the Collection.

An extended protocol is also available for `oracle.toplink.ejb.cmp.wls.CursoredIterator` (based on `java.util.Iterator`):

close() closes the cursor on the server. It is mandatory that the client send this message when it is done with the results.

hasNext() returns a boolean indicating if there is a next element.

next() returns the next available element.

next(int count) retrieves a `Vector` of at most `count` elements from the available results, depending on how many elements are left to read.

This behavior differs from a normal finder as follows:

- Only the elements requested by the client are sent to the client.
- Nothing is cached on the client.
- If you are using the transactional attribute REQUIRED for your entity bean, all reads must be wrapped in a `UserTransaction begin()` and `commit()`. If not, reads beyond the first page of the cursor will have no transaction in which to work.

The following example illustrates client-code executing a cursored finder

```
//import both CursoredCollection and CursoredIterator
import oracle.toplink.ejb.cmp.wls.*;
//... other imports as necessary
getTransaction().begin();
CursoredIterator cursoredIterator = (CursoredIterator)
getProjectHome().findByNameCursored("proj%")
 .iterator();
Vector projects = new Vector();
for (int index = 0; index < 50; i++) {
Project project = (Project)cursoredIterator.next();
projects.addElement(project); !
}
// Rest all at once ...
Vector projects2 = cursoredIterator.next(50);
cursoredIterator.close();
getTransaction().commit();
```

6

Run-time Considerations

This chapter discusses some of the relevant run-time issues surrounding writing an application that uses TopLink Container-Managed Persistence in the BEA WebLogic Server container. Other facets of the run-time execution that relate to EJB's and the BEA WebLogic Server are beyond the scope of this document and should be reviewed in the EJB specification and/or the BEA WebLogic Server documentation.

Transaction support

Entity beans that use container-managed persistence may participate in transactions that are either *client-demarcated* or *container-demarcated*.

Clients of entity beans may directly set up transaction boundaries using the `javax.transaction.UserTransaction` interface. Invocations on entity beans are automatically wrapped in transactions that are initiated by the container based upon the *transaction attributes* supplied in the EJB deployment descriptor.

For more information on how to use transactions with EJBs, consult the EJB specification and the BEA WebLogic Server documentation. The following sections describe briefly how TopLink participates in EJB transactions.

TopLink within the BEA WebLogic Server

Within the BEA WebLogic Server, TopLink provides a persistence layer for entity beans. While the BEA WebLogic Server controls all aspects of transaction management, the TopLink layer is synchronized with the BEA WebLogic transaction service so that updates to the database are carried out at the appropriate times. BEA WebLogic, through its JTS JDBC Driver, determines the majority of transactional behavior

When updates occur

In general, TopLink does not issue updates to the underlying data store until the transaction that the enterprise beans are active in begins its two-stage commit process. This allows for:

- SQL optimizations to ensure that only changed data is written out to the data store
- Proper ordering of updates to allow for database constraints

Valid transactional states

All modifications to persistent beans and objects should be carried out in the context of a transaction. The transaction may either be client-controlled or container-controlled.

The TopLink container does not support modifying beans through their remote interface when no transaction is active. In this case, TopLink does not write out any changes to the data. Modifying entity beans without a transaction leads to an inconsistent state, potentially corrupting the values in the TopLink cache. Transactional attributes MUST be properly specified in the bean deployment descriptors, to ensure that data is not corrupted.

Although it is not valid to modify entity beans through their remote interface without a transaction, in the current release it is permitted to invoke methods on EJB homes that change the state in the underlying database. Invocation of removes and creates that are invoked against homes in the absence of a transaction are permitted.

Maintaining bi-directional relationships

When one-to-one or many-to-many mappings are bi-directional, the back-pointers must be correctly maintained as the relationships change. When the relationship is between two entity beans (in EJB 2.0), TopLink automatically maintains the relationship. However, when the relationship is between an entity bean and a Java object, or when the application is built to the EJB 1.1 specification, the relationship must be maintained manually. To set the back-pointer under the EJB 2.0 specification, either

- The entity bean can maintain it when the relationship is established or modified, or
- The client can explicitly set the back-pointer.

If back-pointers are set within the entity bean, the client is freed of this responsibility. This has the advantage of encapsulating the mapping maintenance implementation in the bean.

Note: Under the EJB 1.1 specification, all back pointers must be updated manually.

One-to-Many relationship

In a one-to-many mapping, an `EmployeeBean` might have a number of dependent `phoneNumbers`. When a `phoneNumber` is added to an employee record, the `phoneNumber`'s back-pointer to its owner (the employee) must also be set.

Example 6–1 Setting the back-pointer in the entity bean

Maintaining a one-to-many relationship in the entity bean involves getting the local object reference from the context of the EmployeeBean, then updating the back-pointer. The following code illustrates this technique:

```
// obtain owner and phoneNumber
owner = empHome.findByPrimaryKey(ownerId);
phoneNumber = new PhoneNumber("cell", "613", "5551212");
// add phoneNumber to the phoneNumbers of the owner
owner.addPhoneNumber(phoneNumber);
```

The Employee's addPhoneNumber() method maintains the relationship as follows:

```
public void addPhoneNumber(PhoneNumber newPhoneNumber) {
    //get, then set the back pointer to the owner
    Employee owner = (Employee)this.getEntityContext()
     .getEJBLocalObject();
    newPhoneNumber.setOwner(owner);
    //add new phone
    getPhoneNumbers().add(newPhoneNumber);
}
```

Managing dependent objects (EJB 1.1)

The EJB 1.1 specification recommends that entity beans be modeled such that all dependent objects are regular Java objects and not entity beans. If a dependent or privately owned object is to be exposed to the client application it must be serializable (it must implement the `java.io.Serializable interface`) so that it may be sent over to the client and back to the server.

Serializing Java objects between client and server

Recall that entity beans are remote objects. This results in a "pass-by-reference" situation when entity beans are referenced remotely. When an entity bean is returned to the client, a remote reference to the bean is returned.

Regular Java objects are not remote objects like entity beans are. Instead of a "pass-by-reference" situation, when regular Java objects are referenced remotely they are "passed-by-value" and serialized (copied) from the remote machine that they were originally on.

Merging changes to regular Java objects

One of the side-effects of serializing regular Java objects from server to client and vice-versa is a loss of object identity, due to the copying semantics inherent in serialization. When a dependent object is serialized from the server to the client and then back, two objects with the same primary key but different object identity exist in the server cache. These objects must be merged to avoid exceptions.

If relationships exist between entity beans and Java objects and these objects are serialized back and forth between the client and server:

- Use the TopLink SessionAccessor utility class to perform the merging for you, or
- Do the merging yourself by putting merge methods on your regular Java objects and within your "set" methods

Using SessionAccessor to merge dependent objects Following the first option, you would use the class `oracle.toplink.ejb.WebLogic.SessionAccessor` to perform merges for you within your "set" methods on your bean class that take regular Java objects as their arguments.

There are two static methods defined on SessionAccessor that allow you to do the register/merge operation. One is called `registerOrMergeObject()` and the other is called `registerOrMergeAttribute()`.

The registerOrMergeObject() method takes two arguments: the object to merge and the EntityContext for the bean. For example:

```
public void setAddress(Address address) {
    this.address = (Address) SessionAccessor.registerOrMergeObject
    (address,this.ctx);
}
```

The `registerOrMergeAttribute()` method requires three arguments: the Java object to be merged, the name of the attribute, and the EntityContext for the bean:

```
public void setAddress(Address address) {
    this.address = (Address) SessionAccessor.registerOrMergeAttribute
     (address, "address", this.ctx);
}
```

The `registerOrMergeAttribute()` call can be used "as is" for collection mappings: you pass in the whole collection as the attribute object. For example:

```
public void setPhones(Vector phones) {
    this.phones = (Vector)SessionAccessor.registerOrMergeAttribute(phones,
    "phones", this.ctx);
    //... additional logic to set back-pointers on the phones
}
```

The `registerOrMergeObject()` method is not as simple to use for setters of collection mappings. It can be used, but the collection must be iterated through, invoking the `registerOrMergeObject()` for each element in the collection. A new collection, set in the entity bean, must be created to hold the return values of the call.

Merging code may be required in methods that add elements to a collection. For example:

```
//The old version of this phone number is removed from the collection. It is
assumed that equals() returns true for phones with the same primary key value.
If this is not true, the phones must be iterated through to see if a phone with
the same primary key already exists in the collection.
public void addPhoneNumber(PhoneNumber phone) {
phone.setOwner((Employee)this.ctx.getEJBObject());
//add to collection
//merge new phone
PhoneNumber serverSidePhone =
(PhoneNumber)SessionAccessor.registerOrMergeObject(phone,this.ctx);
//set back pointer
getPhoneNumbers().addElement(serverSidePhone);
}
```

> **Note:** This exmple only requires merging code if there is a risk that a **Phone** with the same primary key could be added twice. If it can be assured that the elements in a collection are not added in more than once, the merging code is not required.

Merging dependent objects without SessionAccessor As noted above, the issue that arises with serializing is that multiple copies of the same object (with different object identity) can exist within the server cache. These objects must be "merged", either using the `SessionAccessor` methods described above, or manually.

There may be several ways to merge the objects. For example, you could use a `set()` method as follows:

```
public void setAddress(Address address) {
    if(this.address == null){
        this.address = address;
    } else{
        this.address.merge(address);
    }
}
```

Merging must also be done when objects are added to a collection on the entity bean. If it is certain that objects are never "re-added" to a collection, then merging is not necessary.

Merging a whole collection requires a little more work. For each object in the new collection, it must be determined if a copy of the object already exists in the collection. If it does, it must be merged. If not, the new object must simply be added to the collection.

Managing collections of EJBObjects (EJB 1.1)

Collections typically use the `equals()` method to compare objects. However, in the case of a Java object that contains a collection of entities, the `EJBObjects` do not respond as expected to the `equals()` method. In this case, the `isIdentical()` method should be used instead. Consequently, you cannot expect the standard collection methods such as `remove()` or `contains()` to work properly when applied to a collection of `EJBObjects`.

Note: This issue does not arise in the case of an entity containing a collection of entities, because a special EJB 2.0 container collection is used which handles equality appropriately.

Several options are available when dealing with collections of `EJBObjects`. One option is to create a helper class to assist with collection-type operations. An example of such a helper is provided in the distribution named `EJBCollectionHelper`:

```
public void removeOwner(Employee previousOwner){
EJBCollectionHelper.remove(previousOwner, getOwners());
}
```

The implementation of remove() and indexOf() in EJBCollectionHelper is shown in the next example:

```
public static boolean remove(javax.ejb.EJBObject ejbObject, Vector vector) {
    int index = -1;
    index = indexOf(ejbObject, vector);
    // indexOf returns -1 if the element is not found.
    if(index == -1){
        return false;
    }
    try{
            vector.removeElementAt(index);
    } catch(ArrayIndexOutOfBoundsException badIndex){
        return false;
    }
    return true;
}
public static int indexOf(javax.ejb.EJBObject ejbObject, Vector vector) {
    Enumeration elements = vector.elements();
    boolean found = false;
    int index = 0;
    javax.ejb.EJBObject current = null;
    while(elements.hasMoreElements()){
        try{
            current = (javax.ejb.EJBObject)
            elements.nextElement();
            if(ejbObject.isIdentical(current)){
               found = true;
               break;
            }
        }catch(ClassCastException wrongTypeOfElement){
            . . .
        }catch (java.rmi.RemoteException otherError){
            . . .
        }
        index++; //increment index counter
    }
```

```
    if(found){
        return index;
    } else{
        return -1;
    }
}
```

If JDK 1.2 is used, a special Collection class could be created that uses `isIdentical()` instead of **`equals()`** for all its comparison operations. For `isIdentical()` to function correctly, the `equals()` method must be properly defined for the primary key class.

7

Customization

With container-managed persistence (CMP), many aspects of persistence are handled transparently by the EJB “container”. Other properties may be configured, as required, in the bean deployment descriptors (see "Configuring entity bean deployment descriptors" on page 4-3). The intent is to minimize the amount of persistence code that the EJB developer has to write.

However, there are cases where a bean developer or deployer wants to take advantage of advanced features that require additional customization and configuration of bean deployment.

TopLink Container-Managed Persistence provides a number of entry points for advanced customization of mappings, logins, and other aspects of persistence. These can be used to take advantage of advanced TopLink features, JDBC driver features, or to gain “low-level” access to TopLink for Java APIs that are normally masked in the container-managed persistence layer.

Note: For basic information about TopLink descriptors and mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Customizing TopLink descriptors and mappings

TopLink projects and descriptors are normally created using the TopLink Mapping Workbench. The output of the TopLink Mapping Workbench tool is an XML file that contains all of the mapping information required to store beans and persistent objects in the database.

Some customizations available to the TopLink descriptors that make up the project cannot be configured using the Mapping Workbench. In these situations, customize the mapping information by specifying an *amendment method* to be run at deployment

time.. Each TopLink descriptor can have an amendment method. For more information, see "Customizing TopLink descriptors with amendment methods" on page 7-3.

Alternatively, the TopLink Mapping Workbench can be bypassed entirely, and create all the mappings directly in Java code. With this approach, any customizations can be made directly in the source code.

Creating projects and TopLink descriptors in Java

Creating mappings and TopLink descriptors directly in Java code provides access to features that are not available in TopLink Mapping Workbench.

To define a project using Java code:

1. Implement a project class that extends the `oracle.toplink.sessions.Project` class.
2. Compile the project class.
3. Edit the `toplink-ejb-jar.xml` deployment descriptor so that the `<project-class>` element is used. For more about creating project classes, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

> **Note:** The TopLink Mapping Workbench can be used to create a Java Project class from an existing project which can be used as a starting point for a custom project class. See the *Oracle9iAS TopLink Mapping Workbench Reference Guide* for more information.
>
> Also note that the TopLink Mapping Workbench has an **Export Project to Java Source...** option which can be used as starting point for coding the project class manually.

After the TopLink project is written and compiled, it can be used in deployment. You can specify the project class to be used instead of a project file by filling in the `project-class` element in the `toplink-ejb-jar.xml` deployment descriptors for your entity beans.

The next example shows the session portion of a deployment descriptor that specifies a session class.

```
<session>
    <name>EmployeeDemo</name>
    <project-class>oracle.toplink.demos.ejb.cmp.wls.employee.EmployeeProject
    </project-class>
    <login>
        <connection-pool>ejbPool</connection-pool>
    </login>
</session>
```

Customizing TopLink descriptors with amendment methods

The TopLink descriptor of any persistent class can be modified when the descriptor is first instantiated. For container-managed persistence, this happens when the entity beans are deployed into the EJB server.

Amendment methods are static methods that are run at deployment time and allow for arbitrary descriptor customization code to be run.

For more information on amendment methods, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Working with TopLink ServerSession and Login

TopLink interacts with databases using two key components:

- The `ServerSession` is a TopLink component that interacts with the underlying database on behalf of the application.
- The `DatabaseLogin` contains connection information and settings that are specific to the underlying database.

Understanding ServerSession

In TopLink container-managed persistence support, the `ServerSession` is normally hidden from the EJB developer because interaction with the database is performed transparently by the EJB container (via TopLink). The `ServerSession` is still present "behind-the-scenes", but plays a lesser role in its direct interaction with the EJB application.

The `ServerSession` handles all aspects of persistence, such as caching, reading and writing.

Understanding DatabaseLogin

Databases typically require a valid username and password to login successfully. In a TopLink application, this login information is stored in the `DatabaseLogin` class. All sessions must have a valid `DatabaseLogin` instance before logging in to the database.

For more information on `DatabaseLogin`, see "Database Sessions" in the *Oracle9iAS TopLink Foundation Library Guide*.

Customizing ServerSession and DatabaseLogin

A session amendment class can be used to configure the ServerSession and DatabaseLogin in ways not available through the deployment descriptor file.

The `ServerSession` and `DatabaseLogin` may need to be customized for any of the following reasons:

- You need to specify special settings for the JDBC driver, such as to use *parameter binding* or to use a different data conversion routine to work with an incompatible driver
- You wish to directly access regular TopLink for Java features, such as database connections or caching

Other settings that can be applied to the `ServerSession` and `DatabaseLogin` are:

- Native SQL support — required if your JDBC bridge does not support the JDBC standard SQL syntax
- Binding and parameterized SQL — these options determine whether values are inlined directly into the generated SQL or are parameterized
- Batch writing — allows groups of insert/update/delete statements to be sent to the database in a single batch
- Optimizing data conversion

Additional configuration changes

To make additional configuration changes, one must obtain access to the `ServerSession` or `DatabaseLogin` that exists on the server. This access enables direct method invocation on the session or login.

There are currently two supported methods of obtaining the session and login objects and configuring them:

- Use the Deployment Customization interface
- Use a BEA WebLogic Startup class

Using the DeploymentCustomization interface

A *DeploymentCustomization* class is one that implements the `oracle.toplink.ejb.cmp.DeploymentCustomization` interface. A class implementing this interface must be specified in the `toplink-ejb-jar.xml` bean deployment descriptor for the project. If a class is specified, an instance of the class is created during deployment, and the code provided by the class is run.

The `DeploymentCustomization` interface defines the following methods:

```
public String beforeLoginCustomization(Session session) throws Exception;
public String afterLoginCustomization(Session session) throws Exception;
```

These methods are invoked immediately before and after TopLink logs into the database for the first time (during bean deployment). This gives developers the chance to invoke methods on the `ServerSession` as required.

The `DatabaseLogin` can be obtained by invoking the `getLogin()` method on the Session that is passed in as a parameter to each method. The following example illustrates how the `beforeLoginCustomization()` method configures TopLink to use parameter binding:

```
public String beforeLoginCustomization(Session session)
throws Exception{
    session.getLogin().useBinding();
    return "beforeLogin customization successful";
}
```

The class implementing the `DeploymentCustomization` interface should have a zero argument constructor. The Strings returned from each of the methods are output to the BEA WebLogic Server console.

To supply a class to be used for this purpose, the fully-qualified name of the class must be supplied in the `customization-class` element of the `toplink-ejb-jar.xml` deployment descriptor.

The following example shows the project portion of the toplink-ejb-jar.xml deployment descriptor that specifies a customization class

```
<session>
    <name>EmployeeDemo</name>
    <project-class>
    oracle.toplink.demos.ejb.cmp.wls
    .employee.EmployeeProject.class
    </project-class>
    <login>
        <connection-pool>ejbPool</connection-pool>
    </login>
    <customization-class>
    oracle.toplink.demos.ejb.cmp.wls
    .employee.EmployeeCustomizer
    </customization-class>
</session>
```

Using a BEA WebLogic Startup class

When a project is deployed, the TopLink Session is placed in a static Hashtable keyed on the project identifier (the value of name in the toplink-ejb-jar.xml deployment descriptor).

This can be retrieved at startup-time by using a BEA WebLogic Startup class (an implementor of the interface weblogic.common.T3StartupDef) to:

- Call SessionManager.getManager().getSession(String) to look up the Session under the name supplied by the name parameter
- Invoke methods on the Session

For example:

```
import weblogic.common.*;
import oracle.toplink.threetier.ServerSession
import oracle.toplink.tools.sessionmanagement.SessionManager

public class BindingStartup implements T3StartupDef {
    // Define project identifier here
    static final String PROJECT_IDENTIFIER = "EmployeeDemo";
    public T3ServicesDef services;
    public BindingStartup() {}
    public String startup(String theName, Hashtable properties) {
        System.out.println(theName + " startup" );
        try{
            ServerSession session = SessionManager.getManager().getSession
```

```
            (PROJECT_IDENTIFIER);
            If (session == null) return name + " startup could not find project
            " + PROJECT_IDENTIFIER;
            session.getLogin().useBinding();
        } catch(Exception e) {
            return name + " startup failed: " + e.getMessage();
        }
        return name + " startup successful";
    }
    public void setServices(T3ServicesDef theServices) {
        services = theServices;
    }
}
```

For more information on BEA WebLogic Startup classes, see your BEA WebLogic Server documentation.

8

Clustering

A key feature provided by BEA WebLogic Server is the ability to integrate multiple server instances into what can be viewed by clients as a single server entity referred to as a *cluster*. Once formed, the cluster will support deployment of EJB's and other J2EE components and provide load-balancing and a measure of failover on those components. For more information on BEA WebLogic Server clustering please consult the BEA WebLogic Server documentation.

This chapter discusses how TopLink for BEA WebLogic may be used within a clustered BEA WebLogic Server environment. There are certain issues that affect how a TopLink application should be configured to ensure that it executes correctly and consistently on a cluster. This chapter discusses those issues, explains the TopLink features that help resolve those issues and offers some best practices for clustered applications.

Terminology

A BEA WebLogic Server *instance* is an operating system process in which a single Java Virtual Machine has been invoked passing the BEA WebLogic Server class as the main program argument to the JVM. Server instances may be distributed across multiple host machines or running on the same machine.

When an entity bean is invoked through its remote interface then it must of necessity get loaded into a server instance. Once that bean has been loaded then it is said to be *pinned* to that server, meaning that all subsequent invocations of business logic on that same remote interface stub instance will be directed to the previously instantiated bean on the server into which it was loaded. This does not preclude the bean instance from being instantiated on other servers by acquiring another remote interface to the bean through the use of a finder or reference from another bean.

When two or more beans get pinned to the same server then they are said to be *co-located*. Bean co-location implies that optimizations of locality, such as call-by-reference inter-bean invocations, may be employed on the server.

Other terms and concepts that relate to BEA WebLogic Server clustering are further explained in the BEA WebLogic Server documentation.

TopLink in a Cluster

When using TopLink for BEA WebLogic in a cluster the TopLink run-time jars must be available to all servers in the cluster, or at least all servers in which TopLink CMP beans are deployed. The beans may be deployed on any number or subset of servers in the cluster, with the conditions discussed below in the *Static Partitioning* section.

Related beans (beans that are associated using a TopLink relationship mapping) require special consideration to provide acceptable performance and correctness. The issues surrounding relating beans are discussed in the *Relationships* section.

Each server in the cluster manages its own cache independently, with additional capabilities as described in "Cache Synchronization" on page 8-5. If cache synchronization is not used then the caches must be manually refreshed.

Relationships

To define relationships between beans, all of the related beans and objects must be co-located. Source and target objects should also be retrieved on the same server.

Co-location of related beans may be achieved in BEA WebLogic Server 6.1 and 7.0 by making use of one or more of the following observations:

- Beans that are deployed in a single server are only ever invoked on that server.
- Bean home interfaces are clustered, but bean instances are not. Once a bean has been instantiated, then it is pinned to the server in which it was instantiated
- The server attempts to execute operations that are performed inside the same JTS user transaction on the same server.
- When executing operations that are performed from a session bean (that has already been instantiated in a server) the server attempts to carry out the operations locally.

The first point leads to partitioning the beans across the server as a statically-defined means of co-location. The second point introduces the relevance of pinning to co-location. The last two points convey the ideas that using user transactions or session beans can cause the desired co-location to occur.

Static partitioning

Beans can be deployed to particular servers only, allowing for static partitioning of beans. Statically partitioning the beans across the cluster provides the required co-location conditions as long as all related beans are deployed in the same server. Other unrelated beans may be deployed in the same or a different server, and depending upon the amount of predicted access traffic could be deployed in more than one server. No application code need be modified. Failover is limited, and load-balancing is statically determined. Cache inconsistency is not an issue in this configuration since beans will only ever get loaded on the server in which they were deployed. These types of systems may suffer from bottlenecking and overhead costs.

Pinning

Once a bean is created or found it is pinned to a particular server. The BEA WebLogic server will attempt to keep all beans accessed in a given transaction on the same server in an effort to localize the transaction. A transaction cannot be localized if it involves beans that were previously pinned to different servers.

In order to ensure that all beans are local in the transaction each bean used should be re-looked up in the context of that transaction. Beans that were created or found in previous transactions should be discarded.

There are two common methods of using pinning to dynamically co-locate beans: user transactions and using session beans.

Using User Transactions

Ensuring that all bean invocations are in an enclosing transaction is one way of influencing where beans get instantiated in the cluster. If the beans are deployed in multiple servers then the user transaction may be initiated on any one of the server instances. It doesn't matter which server is chosen since an attempt is made to pin all accessed beans to that server for the duration of the transaction. This way load-balancing can occur while still allowing the co-location demands to be satisfied.

For example, the following code is a portion of a client program that uses a user transaction to co-locate related beans.

```
UserTransaction transaction = lookupUserTransaction()
// Enclose all construction of relationships in the same transaction
transaction.begin();
// Look up the home interface and the bean even if they have already been looked
up previously
Employee emp = lookupEmployeeHome().findByPrimaryKey(new EmployeePK(EMP_ID));
Address address = new Address(EMP_ID, "99 Bank", "Ottawa", "Ontario", "Canada",
"K2P 4A1");
emp.setAddress(address);
Project project = lookupProjectHome().findByPrimaryKey(new ProjectPK(PROJ_ID));
emp.addProject(project);
transaction.commit();
```

Using session beans

Entity beans accessed through a session bean are instantiated on the same server as the session bean. By moving the application logic from the client to a session bean, the optimization of locality can be exploited to allow the bean code to run on the same VM. The client need only invoke a single method in the session bean and the bean performs all of the required logic on the same server. If deployed in every server in the cluster, then scalability as well as failover (which must still be handled by the client) can be achieved.

It depends upon the application whether to use session beans, user transactions, or static partitioning of the beans as a means of achieving co-location, since some of these techniques may not be appropriate for certain models. Regardless of the method, co-location is required in order to define relationships between beans.

Caching issues

Another issue that must be considered when running in a clustered configuration is that of cache consistency. Under normal conditions a TopLink session in a BEA WebLogic Server is an independent and autonomous object. Changes made to a bean in one server are not reflected in the caches of other servers. This situation could lead to objects in different states existing in multiple servers and result in phantom reads or updates to stale data (causing previous changes to be lost), amongst other incorrect behavior.

Explicit query refreshes

One solution that may make sense if only some objects are required to be fresh is to cause them to be refreshed from the database whenever appropriate. This would involve configuring certain finders to cause refreshing and then invoking these queries when the situation warrants. There are two facets to making a query refresh the object -- setting the refresh policy and the cache usage.

Refresh Policy

Whenever a query is issued there is a possibility that the result from the database is more recent than the cached version. But by default if objects are already in the cache then they are used instead of the database results, even if the database query was issued. This is a TopLink optimization that reduces the number of objects that have to be built from database results. This feature may be overridden by setting a refresh policy to ensure the objects from the database replace the ones in the cache, if such objects exist there. This way the cache is always updated with the latest copies of the objects from the database whenever a query is completed. Refreshing can be done on each TopLink descriptor or just on certain queries depending upon the nature of the data that can change.

Note: Refreshing does not prevent phantom reads from occurring. See the "Refreshing finder results" on page 5-15.

Cache Usage

When a `findByPrimaryKey` finder is invoked then the object in the cache is returned if it exists there. The refresh policy is not applied because no database query is issued. In this case, disabling cache hits is required to prevent the finder from using the cached object in case it was deleted or modified by another server. This can be achieved by setting the caching option element to `DoNotCheckCache` in the bean deployment descriptor. For more information, see "Caching options" on page 5-14.

Cache Synchronization

Cache synchronization automatically causes updates made to one TopLink cache to be propagated to all other server caches. This obviates the need to do manual refreshing, and can provide a consistent view of cached data across the cluster. This feature is enabled by supplying a value for the `cache-synchronization` element in the TopLink deployment descriptor.

Cache synchronization is currently supported at the project level. This means that all updates to beans and dependent objects in a given project marked for propagation will be propagated to the caches on all other servers. Propagation at a finer granularity, such as individual beans or objects, may be available in future releases.

Propagation of changes can be configured to function in one of two modes--synchronous or asynchronous. Users should choose the mode that best meets their requirements. For many applications, synchronous mode is more appropriate as it provides for tighter data consistency models.

Remote Merge

Once a transaction has been committed, whether it be through an explicit client commit call or bean method invocation that triggered a transaction begin and subsequent commit, the changes to objects in the transaction must be merged into the TopLink cache. When Cache synchronization is in effect, a remote merge process is also initiated.

Remote merging involves merging the changes into all other remote TopLink caches after the local merge has completed. If problems occur during update propagation it is typically the remote merge process that will be the root cause of such problems. Each server must be able to merge the changes into its local cache and finish up with a consistent version of the object. As mentioned above, the TopLink cache does not begin the merge or update process until the database transaction has already been committed. This is quite beneficial in that it avoids letting uncommitted data into the shared cache, but should be recognized where transactional synchronization is considered. In cases where a merge may have failed there is no way to roll back the changes made to the database (although it is questionable whether this would be a good idea in any case). As a consequence, failures during remote merging can leave the cache in an inconsistent state. This makes it important to handle any errors that occur by performing cache normalization actions, such as resetting the cache, or even the server.

Synchronous Mode

When updates are synchronously propagated, the committing client is blocked until the remote merge process is complete. This provides the client with the assurance that its changes have either successfully reached all remote servers, or that an error occurred and was already handled by its server-side handler. Thus, when the client process gets control, it can invoke another business method and rely upon the receiving server having already incorporated the changes of the clients previous transaction.

Asynchronous Mode

Depending upon the requirements of the application, asynchronous operation may be a more efficient approach to updating the distributed caches. When a transaction commits the updates are sent off to the remote servers while the committing client gets control returned to it. Though there are no guarantees as to delivery, errors resulting from merging the updates can be caught and handled by server-side handlers installed by the application, just as in the synchronous case. However, since the client has already been unblocked there is no opportunity to take any action that would affect the calling client, and the client may have already gone on to invoke other business methods on the server cache where the merge failed.

The asynchronous mode of operation is particularly appropriate when freshness time constraints are softer or less of an issue. This would include such applications where it is acceptable to read stale data on some occasions immediately after an update, as long as the cached data gets updated within a "reasonable" period of time.

Configuring Cache Synchronization

Cache Synchronization is configured using the `toplink-ejb-jar.xml` deployment descriptor (See Chapter 4, "EJB Entity Bean Deployment"). It is invoked using the optional `cache-synchronization` element and configured using the a number of optional sub-elements:

cache-synchronization (optional) When provided, indicates that changes made to one TopLink cache in a cluster should be automatically propagated to all other server caches. The following elements may also be provided:

- **is-asynchronous** (optional): Set to *True* if synchronization should not wait until all sessions have been synchronized before returning. Valid values are *True* or *False*. Default is *True*.
- **should-remove-connection-on-error** (optional): Set to *True* if a synchronization connection should be removed from the session if a communication error occurs. Valid values are *True* or *False*. Default is *True*.

Following is an example TopLink descriptor that specifies cache synchronization:

```
<toplink-ejb-jar>
    <session>
        <name>ejb20_AccountDemo</name>
        <project-class>
            oracle.toplink.demos.ejb20.cmp
            .account.AccountProject
```

```
        </project-class>
        <login>
            <connection-pool>ejbPool
            </connection-pool>
        </login>
        <cache-synchronization>
            <is-asynchronous>True
            </is-asynchronous>
            <should-remove-connection-on-error>True
            </should-remove-connection-on-error>
        </cache-synchronization>
    </session>
</toplink-ejb-jar>
```

Cache Locking

To protect objects from being written by more than one client at a time, or to protect against using stale data for updates, optimistic locking causes a check to occur at transaction commit time. This check ensures that no client has gotten in and modified the data since it was last read by the client making the update. An OptimisticLockException exception will be generated and will cause the commit to fail should a stale write be detected. This strategy should be used regardless of whether updates are refreshed or automatically propagated.

Note: Using optimistic locking by itself does not protect against phantom reads or having different copies of the same object existing in multiple nodes. See "Optimistic Locking" in the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Even when synchronous mode is activated and when optimistic locking is in place this cannot guarantee that all clients will always read the freshest data, because that is not possible without pessimistically locking the data being read. Update propagation is designed to provide a convenient and efficient trade-off that minimizes any optimistic locking conflicts that occur and provides specialized functionality to many clients that have consistency requirements.

Using cache locking

When update propagation is in effect the remote merging process causes the number of updates to each cache to be increased substantially, because each cache is updated once for every transaction in the system. The default cache locking policy is set to allow concurrent reading and writing to optimize cache access, but this may

be changed to ensure safer cache updates during propagation. To change the cache isolation level to lock the cache during updates, a customization class must be supplied and the cache isolation level can be set on the login. For available isolation options, refer to "Cache Isolation" in the *Oracle9iAS TopLink Foundation Library Guide.*

```
afterLoginCustomization(Session session) throws Exception
{session.getLogin().setCacheTransactionIsolation(DatabaseLogin.SYNCHRONIZED_
READ_ON_WRITE);
```

9

The EJB 2.0 Single Bean Example Application

This chapter introduces the basic concepts that are required to build and deploy an entity bean with TopLink. It provides an example of how TopLink CMP is used in a simple application that combines Java server pages (JSPs) and EJBs. A simple entity bean is used to illustrate TopLink's basic direct-to-field mapping capabilities, along with some simple EJBQL queries.

Included in this example are instructions for

- creating bean classes and interfaces
- creating deployment descriptors
- mapping entities to the database using the Mapping Workbench
- generating a deployable JAR file

The example package is `examples.ejb.cmp20.singlebean`.

See `<INSTALL_DIR>/doc/demos.html` for links to all the examples and details on configuring the examples for WebLogic Application Server.

A note about this example This chapter guides you through running the example, as well as describing in detail how the example was built. Although the Single Bean example is a relatively simple application, the process and procedures it introduces can be used to build much more complex applications as well.

Running the Single Bean example

To run the Single Bean example, you must

- Start the TopLink domain WebLogic Server. From Windows Start menu, click **Programs** > **Oracle9iAS TopLink** > **Example Utilities** > **Start TopLink WebLogic Demo Server**.

Configuring the example database

TopLink includes HSQL, as well as a preconfigured database containing all of the tables required to run the example. The SQL commands required to recreate the database are found in the `createTables.sql` file. Use the provided `ResetDatabase.cmd` script to reset the database tables if required.

Understanding the Single Bean example

The Single Bean example shows how a single bean can be made persistent using TopLink Container-Managed Persistence CMP support. This example illustrates simple direct-to-field mappings and introduces the basic steps required to deploy a bean.

This is a simple example that demonstrates how to use entity beans when deployed using TopLink CMP. The example consists of an entity bean called `Account`.

The Single Bean example demonstrates:

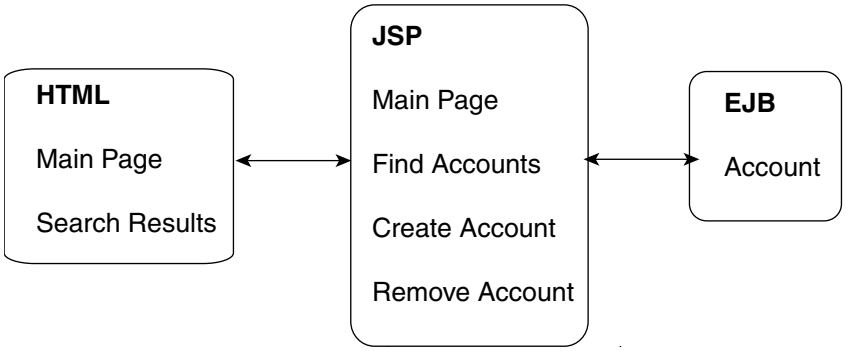
- The use of an entity bean in an application
- That persistence-related code is not required in the entity bean
- That implementation of `ejbFind` methods is not required in the entity bean – query logic for each finder method defined in the home interface is defined using the Mapping Workbench or in an amendment method
- The use of the TopLink "project" class, which contains bean-to-database mapping meta information
- How TopLink transparently manages persistence for beans when they are being created, updated, removed, and queried

The client application is an HTML page generated by JSP components. The Single Bean example follows the basic J2EE application architecture, using pure HTML generated by JSPs. The JSPs access the Single Bean entity bean.

The example can be viewed by pointing a web browser at http://localhost:7001/Account/mainPage.jsp. Starting at this main page, you can query for single beans based on the associated balance and owner, and add and delete single beans.

The use of JSPs and EJBs in this example provides a simple example of a J2EE application, and demonstrates one of several different ways to leverage this technology.

Figure 9–1 Structure of the Single Bean Example



Single Bean example: packages, classes, and file

The following table lists the packages, classes, and files associated with the Single Bean example in this chapter. All files associated with the example are located in `<install>\examples\wlsxx\examples\ejb\cmp20\singlebean` unless otherwise noted.

Table 9–1 Packages, classes, and files in the Single Bean example

Component Type	Component Name / Location
Package	`examples.ejb.cmp20.singlebean` **Note:** The root directory for the package is `<install>\examples\`
Server classes/interfaces	`Account, AccountHome, AccountBean`

JSP source	`./jsp/mainPage.jsp,`
	`./jsp/findAccounts.jsp,`
	`./jsp/createAccount.jsp,`
	`./jsp/removeAccount.jsp`

Table 9–1 Packages, classes, and files in the Single Bean example (Cont.)

Component Type	Component Name / Location
HSQL database build script	`ResetDatabase.cmd` (Windows only)
Table definitions and data population SQL	`createTables.sql`
Environment settings	`<Install>/setenv.cmd` (Windows only)
Deployment files	`/ejb-jar.xml,weblogic-ejb-jar.xml, toplink-ejb-jar.xml`
TopLink Mapping Workbench project	`./mw/Account.mwp`
Sample build script	`build.cmd`
Deployable project	`Account.xml`
Mapping Workbench directory	`./mw`

The Object model

The Single Bean example provides a simplified view of the standard "bank account" example, and shows how a single class can be modeled as an entity bean and made persistent using TopLink.

The interface `examples.ejb.cmp20.singlebean.Account` provides the public "local" interface for the bean. It extends the `javax.ejb.EJBLocalObject` interface, and contains all of the business methods that are accessible to local clients of the entity. This includes getters and setters for the instance data, as well as `deposit()` and `withdraw()` methods.

The class `examples.ejb.cmp20.singlebean.AccountBean` provides the actual bean implementation for the bank single bean. It has methods corresponding to the methods on the remote interface, as well as the methods required by the `javax.ejb.EntityBean` interface, which it extends. The `account`'s fields include `AccountId (String)`, `balance (double)`, and `owner (String)`.

The interface `examples.ejb.cmp20.singlebean.AccountHome` provides the "local home" interface of the bean. It extends the `javax.ejb.EJBLocalHome` interface, and defines the required create, remove, and finder methods.

In this example, no additional primary key class is used. In EJB 1.1 and above, additional primary key classes are not required if the primary key consists of one field.

Database schema

The Single Bean data is stored in a single table.

Table 9–2 The EJB_ACCOUNT BEAN table

Column Name	Column Type	Details
ACCOUNT_ID	Numeric	primary key
BALANCE	DOUBLE	balance in single bean
OWNER	VARCHAR	owner's name

The example also makes use of table-based sequencing through the EJB_ACCOUNT_SEQ table. This table provides primary keys (account numbers) for the EJB_ACCOUNT table.

Table 9–3 The EJB_ACCOUNT_SEQ table

Name	Type	Details
SEQ_NAME	VARCHAR	Used by TopLink to identify the table that requires sequence data
SEQ_VALUE	DECIMAL	Sequence data

Entity Development

A deployable component is typically developed as follows:

- Create the interfaces.
- Create and implement the bean classes.
- Create the deployment descriptors.
- Map the entities to the database.
- Deploy the JAR file.
- Deploy the JAR file.

Create the interfaces

Each entity can contain any or all of the following interfaces:

- local home (under EJB 2.0)
- home
- local
- remote

These interfaces dictate how the bean is used by other components of the application. The Account interfaces have been created and are located in the main "single bean" example directory.

Create and implement the bean classes

Define abstract `get` and `set` methods to represent the persistent attributes and use the `get` and `set` methods when implementing the business logic in the beans. The AccountBean class has been created and is located in the main Single Bean Example directory.

Create the deployment descriptors

Three deployment descriptors are required for each JAR. They are

- `ejb-jar.xml`
- `weblogic-ejb-jar.xml`
- `toplink-ejb-jar.xml`

The `ejb-jar.xml` descriptor should be created as specified in the EJB 2.0 specification. The `weblogic-ejb-jar.xml` file should be defined as described in WebLogic Server documentation. The `toplink-ejb-jar.xml` file should be defined as described in this documentation. These descriptors have been created and are located in the example directory.

The deployment descriptors must be in the META-INF directory in the JAR.

ejb-jar.xml

This is the main deployment descriptor prescribed and specified by the EJB specification. It defines most of the basic properties for the session and entity beans, including

- the bean home and component interfaces
- the bean abstract class
- inter-bean relationships information
- security declarations
- transactional attributes
- references to other beans
- resources used by the bean

The key elements that are particularly relevant to TopLink CMP are:

<ejb-name> A unique name (across all beans in the JAR) which must match the `ejb-name` attribute in the `weblogic-ejb-jar.xml` file.

<cmp-version> Must be set to 2.x to indicate that EJB 2.0 is being used.

<resource-ref> Lists the data source used by the entity in which it is declared.

<query> A group of elements that, when combined, describe a particular bean finder by including the method name, parameter types and query criteria.

> **Note:** Mapping beans in the Mapping Workbench creates or modifies the information in some of the fields in the project. Using the Mapping Workbench to write to the `ejb-jar.xml` ensures that the `ejb-jar.xml` and the TopLink project information remain properly synchronized.

The use of all XML elements in the file are described in the EJB 2.0 specification.

weblogic-ejb-jar.xml

This descriptor file is specific to WebLogic Server and offers an opportunity to override some of the configuration settings in the server. Some of the elements that may require modification are:

<persistence-type> A section that sets the persistence type. Note that this tag is not required by WebLogic 7.0, as its function is included in the `<persistence-use>` tag.

<persistence-use> A section that specifies TopLink as the persistence storage mechanism.

For information on configuring the `weblogic-ejb-jar.xml` file, see "Configuring entity bean deployment descriptors" on page 4-3.

toplink-ejb-jar.xml

The TopLink deployment descriptor is included in the JAR in the same META-INF directory as the other two deployment descriptors. This descriptor provides the information that TopLink needs to deploy the entities in the JAR. Because the entities deployed in a JAR are all encompassed by a TopLink project, the deployment JAR file is associated with exactly one project. This project is in turn associated with exactly one TopLink session (as implied by the single session element in the descriptor).

The elements that have been modified for the Single Bean example in the `toplink-ejb-jar.xml` file are:

<name> A session name (unique among all deployed JARs) that is used as a key for the deployed TopLink project (or the JAR that contains the project).

<project-class> The fully-qualified name of the TopLink project class. This class should be included in the deployable JAR file. The project class can either be generated by the Mapping Workbench or written manually.

Note: You can use a `<project-xml>` rather than a <project-class> if you choose. The `<project-xml>` element specifies a project deployment XML file that can be stored either in the deployable JAR file or left on the file system. For more information, see "Configuring entity bean deployment descriptors" on page 4-3.

<connection-pool> A sub-element of login, the fully-specified connection pool specifies the connection pool URL of the data source.

For more information on the options available for the `<connection-pool>` element, refer to the DTD in `<install>\wls_cmp\toplink-wls-ejb-jar_903.dtd`.

Map the entities to the database

This section describes the steps required to create the Single Bean project using the Mapping Workbench. It also introduces and addresses some issues related to mapping EJB 2.0 beans, such as relationships and reserved finders, which are not encountered in the Single Bean example.

For more information about creating projects using the Mapping Workbench, consult the *Oracle91AS TopLink Mapping Workbench Reference Guide*.

This section assumes you have already read and completed the introductory tutorials in *Oracle9iAS TopLink Tutorials*, which offers an introduction to the fundamental concepts of the Mapping Workbench.

Creating a TopLink project

A TopLink project defines how the entity beans are to be persisted to the database. The Mapping Workbench can be used to easily build a TopLink project. The project is specified in the `toplink-ejb-jar.xml` in the `<project-class>` or `<project-xml>` element and used at runtime to persist the beans.

To create a TopLink project:

1. Create a new project. Click **File** > **New Project**.
2. In the General tab, set the **Persistence Type** to 2.0 CMP.
3. Specify an `ejb-jar.xml` file to use for the project. You can either choose an existing file or create a new one. The `ejb-jar.xml` file is typically created by another tool, and the mapping information it contains is read and used by the Mapping Workbench (see step 7 below).

   If no `ejb-jar.xml` file is available, the Mapping Workbench can be used to create one and to populate the file with the information the Mapping Workbench requires. Once the file is created, however, other elements usually need to be added to the `ejb-jar.xml` file before it can be used with ejbc and in a deployable JAR.

When the Mapping Workbench project is saved, changes to the `ejb-jar.xml file` can also be written out. See "Working with the ejb-jar.xml file" in th*e Oracle9iAS TopLink Mapping Workbench Reference Guide* for details on working with the ejb-jar.xml file.

4. In the General tab, specify a project classpath. The project classpath should contain the classes to be added to the project and interfaces associated with those classes. Classes to be added to the project include bean classes and referenced classes. Bean interfaces do not have to be added to the project, but must appear in the project classpath.
5. To add the beans to the project, click **Selected** > **Add/Refresh Classes**. While bean classes must to be added to the project at this point (for example, the Single Bean example requires the AccountBean class) referenced classes are not required. At this point the `cmp` and `cmr` fields do not appear on the AccountBean descriptor.
6. To specify the beans classes as bean descriptors, click **Selected** > **Descriptor Type** > **Class Descriptor**. This causes the Mapping Workbench to read the abstract getters/setters and `ejb-jar.xml` file for `cmp` and `cmr` fields.

   After setting the AccountBean to be an EJB descriptor, the `cmp` and `cmr` fields appear and are available for mapping.
7. Update the project from the `ejb-jar.xml` file by selecting the project and clicking **Selected** > **Update Project from ejb-jar.xml**. This is an optional step that is not required if you are generating the `ejb-jar.xml` file in the Mapping Workbench. However, the the Single Bean example uses an existing `ejb-jar.xml` file, so using it to update the project brings in two user-defined finders which appear in the AccountBean descriptor's Queries tab.

   The Mapping Workbench reads from and writes to the following elements in the `ejb-jar.xml`:

   - primkey-field
   - ejb-name
   - local-home
   - local
   - ejb-class
   - prim-key-class
   - abstract-schema-name

- cmp-field
- query and its sub-elements
- relationships

8. Create database tables. The Single Bean example uses an EJB_ACCOUNT table to persist the bean to an EJB_ACCOUNT_SEQ table for sequencing. Ensure that the ACCOUNT_ID is the primary key in the EJB_ACCOUNT table. The tables can either be imported from the database or created in the Mapping Workbench. For more information on working with tables, see "Working with database tables" *Oracle9iAS TopLink Mapping Workbench Reference Guide*.
9. To associate AccountBean with a table, select the AccountBean and set the EJB_ACCOUNT table as the associated table in the **Descriptor Info** tab.
10. Map the `cmp` and `cmr` fields. The AccountBean has three `cmp` fields to be mapped using direct-to-field mappings: accountId, balance, and owner. Map them to their corresponding database fields in the EJB_ACCOUNT table.
11. Set up sequencing. The AccountBean uses TopLink sequencing for its primary key generation. In the **Descriptor Info** tab, select the **Use Sequencing** check-box and specify ACCOUNT_SEQ for the Name, and EJB_ACCOUNT and ACCOUNT_ID for the Table and Field.
12. Export a project to be used at runtime. The project can be written out as a Java class which has to be compiled and included with the deployment JAR or an XML file. In the `toplink-ejb-jar.xml` file either the `<project-class>` or `<project-xml>` element is used depending on which export method was used.

Generate the deployable JAR file

The Single Bean example is packaged into and EAR file, which itself contains the following:

- A WAR file containing contains the JSP code and associated configuration XML files.
- A deployable JAR file containing the interface and abstract bean classes, the classes (RMI stubs and implementation classes) generated by weblogic.ejbc, and the deployment descriptor XML files.

Using the Build Script

A build script is included in the `...\singlebean` example directory which compiles the bean classes, runs weblogic.ejbc, and creates the EAR file. Running the build script places a copy of the EAR file in both the Single Bean directory and the server's deployment directory.

Windows users can use the build script as provided. Users of other platforms can use the build script as a template to create their own build script.

The build script can be modified and used to build deployable EAR or JAR files for other applications.

The build.cmd file uses the environment information in the `<install>/setenv.cmd file` (Under Windows) or the `setenv.sh` file (non-Windows). To avoid possible errors, ensure that your environment also matches these settings before running the script.

Deploy the JAR file

This can be done a number of different ways. See the WebLogic Server documentation for more detailed information on how to deploy an EJB JAR or an EAR into the server.

A

EJB Architectures Summary

Enterprise JavaBeans present a way to build components as well as a means to make these components exist in a transactional, secure, and distributed environment. However, a single bean represents only one component - and consequently only one part of a complete application. EJB provides developers with flexibility in determining how these components should be made to work together. There are a number of ways in which Enterprise JavaBeans can be made to work together to form a complete enterprise application. TopLink can be integrated into each variety of EJB application architecture to provide both the technology that enables these architectures and the features that add value to them.

This chapter gives an overview of some of the basic design patterns available when using TopLink and TopLink CMP. It is not meant to be prescriptive and neither is it complete. It briefly suggests some of the more useful EJB designs and their suitability to specific applications. Architects and developers may find these sections useful at the early stages of application design. As more experience is acquired the appropriateness of particular patterns will become more obvious, and architectural decisions will be more intuitively reached.

Introduction to EJB architectures

The basic ways in which EJBs can be assembled, or the basic "EJB architectures", can be described in terms of which kinds of beans or J2EE components are used, how client applications access them, and how the underlying "domain objects" are represented. The EJB architectures can be fundamentally divided into three categories: an Entity bean architecture, a Session bean architecture, and a Session and Entity "tiered" approach. Each basic architecture also has variations and refinements and can be decorated with a variety of J2EE components.

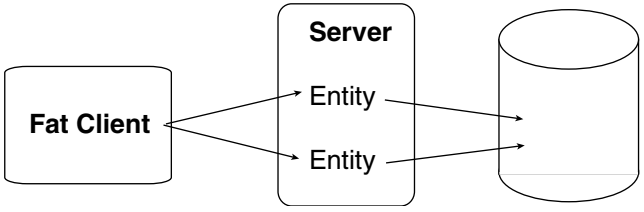
The EJB 2.0 specification does not dictate how enterprise entities are used, but it is clear from the evolution of the specification that certain architectures were assumed to be dominant. Some of the recommended architectures are explained in the J2EE Blueprints (see http://java.sun.com/blueprints). These documents should be reviewed for more information about J2EE and EJB architectures.

Remote Entities

If entities alone are used then they must have remote interfaces that expose all of the client servicing methods. In the absence of Session beans, the client may only access entity state through its remote interface, and may not traverse relationships, except as encapsulated by remote method calls. Only remote references and data may be returned by these calls. Finders may only return remote references as well.

If relationships do exist then they must be implemented using local interfaces, hence there must be different levels of access from the remote and local interfaces. References to related entities must be translated from local to remote references if they are to be passed back from remote client calls.

Figure A–1 Remote entities architecture



Clients gain from this approach in that the distribution of the entities is transparent. Clients reference the entities as if they were local, and do not need to worry about location. Entities can exist at different locations without the client even being aware of it.

The converse of transparent distribution is that if clients are not aware of the distributed nature of the entities then they may not be aware of the cost of invoking them. If the entities are “fine-grained” objects then each fine-grained method invocation on them will end up being a remote call. The accumulation of these remote method invocations could sum up to a potentially serious network or communication latency cost.

If Container transactions are used for each entity operation a separate transaction will end up being initiated for each method invocation. This could introduce excessive and unnecessary transaction management overhead if client-demarcated UserTransactions are not used.

Since Entity beans are intended to be "components" there are more restrictions placed on them than on regular Java objects (e.g. thread-spawning is disallowed). This may impose limits on how they can be used to model certain domain concepts. The limitations should be well understood and compared against the model to ensure that they are not discovered too late in the design phase.

In general, however, this architecture is less desirable than other architecture types. The relationship limitations imposed by the EJB 2.0 specification are often an impediment to using this approach.

Advantages

- Increased distribution
- Potential for greater location transparency

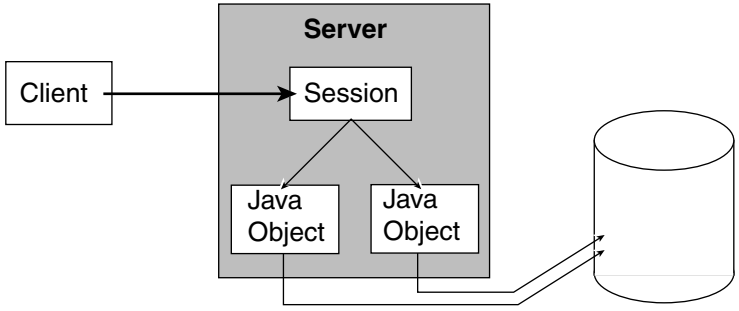
Disadvantages

- Not suitable for fine-grained entities
- Cannot have relationships between entities
- Communication overhead for each method call
- Transactional costs of each method call
- Client accesses many EJB interfaces (no single client interface or point of server entry)
- Much of the business logic must reside on the client

Remote Session beans

It is common practice to apply a Session bean layer in front of lightweight objects. This may take the form of the session façade pattern described below, with lightweight entities or other types of persistent objects being managed.

Figure A–2 Remote session beans architecture



Since Session beans do not themselves represent durable objects, often a session façade pattern will be used to converse directly with persistent Java objects, without the use of entities. Persistent data can be modeled using regular TopLink-enabled persistent Java objects that are managed by the Session beans and mapped using TopLink tools. Since few domain objects actually "live" on the client, client applications rarely need to access the domain objects directly, but if regular objects are used then they may be sent to the client if necessary since no such restriction exists for them. The Session beans are used to carry out most of the application logic. Stateful beans are used for those operations for which client-identity is important, while stateless Session beans can be used for "single-shot" operations. All of the EJB benefits of security, transactions and distribution are available through the Session beans.

The exclusive use of Session beans does not allow for overly complex client behavior - all client behavior is limited to services provided by the Session beans. Simple client behavior is a general characteristic of all thin client architectures.

Simplicity and fast client access are clear benefits of this approach. In addition, there is great flexibility in how the domain objects are designed, and how these objects are mapped to the underlying relational database tables.

Advantages

- Location transparency on session interface
- Fine-grained persistent object operations can be "batched" or combined into a single session bean call to reduce communication overhead
- High performance storage/retrieval of persistent objects
- EJB features available through single point of session bean entry to reduce Container overhead

- Can relate persistent objects
- Can pass persistent objects to the client side if necessary

Disadvantages

- Fine-grained client calls may require explosion of session interface calls
- Persistent Java objects not included in EJB specification

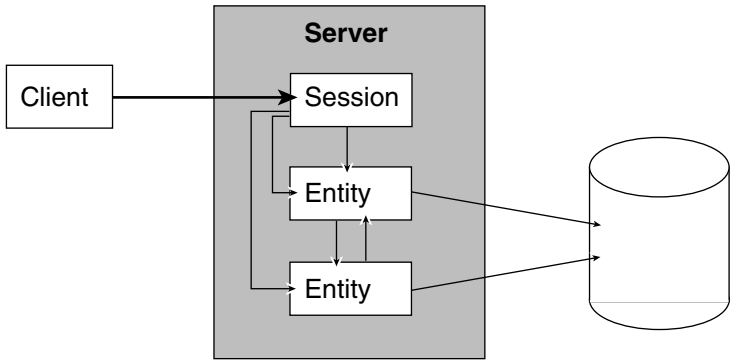
Session Façade - Combining Session and Entity beans

The majority of systems have relationships between application entities. This being the expected scenario, it is well observed and explained by J2EE designers. The EJB 2.0 specification dictates that all such related entities must be co-located within the same server, or running in the same VM. Such a requirement to use local interfaces to entities both enforces transactional integrity of relationships and opens the door to substantial optimization of entity invocation.

The problem of accessing the local entities on the server from a remote Java client is overcome by installing a session bean that exports the required client operations through a remote interface, as was described in the above section. The client application, then, converses only with the session bean and passes all of its data and requests to the session bean for service. The client may still retain much of the logic, if so desired, with the session bean making fine-grained operations on the local entities. More common, however, is to incorporate the domain logic into the session bean itself. This migrates the business logic from the client to the server which provides a number of well-known benefits including ease of maintenance, convenient upgradability, and increased access to server features.

Regardless of whether the session bean simply forwards operations to entities or actually includes the application logic as a façade that fronts the local entities it is a modular approach to remotely accessing server-side objects. It is also likely to be easier to maintain as the J2EE specification moves forward, since session beans tend to experience change to a lesser degree than other components, such as entities. The decision to use a stateful or stateless session bean will likely depend on the amount of business logic incorporated into the session bean.

Figure A–3 Session façade architecture



Advantages

- Location transparency on session interface
- Fine-grained entity operations can be "batched" or combined into a single session bean call to reduce communication and transaction overhead
- Inter-entity method calls are pass-by-reference
- Can maintain entity relationships
- All components described by EJB specification
- Flexibility to create new (local) transactions on specific method calls when required·

Disadvantages

- Fine-grained client calls may require explosion of session interface calls
- Some Container overhead still incurred on each local entity call

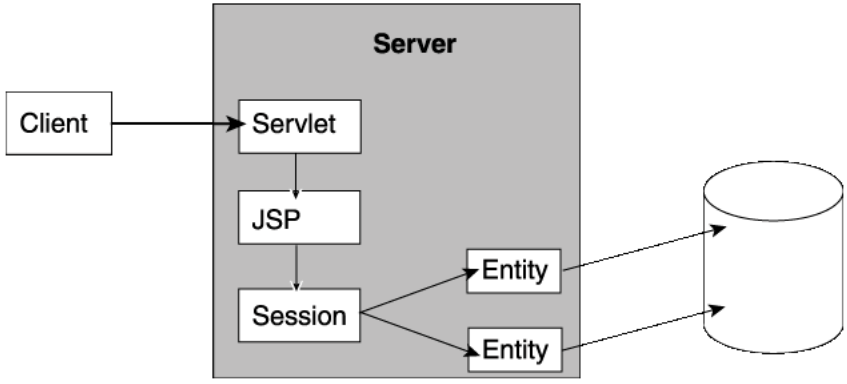
Thin Client

A prevalent B2B architecture uses an HTTP browser as its client and makes use of J2EE servlet and JSP components to contain the presentation and sometimes even business logic. These layers then typically invoke EJB's for transactional behavior. Although the EJB specification seems to treat server-side components as remote the J2EE specification does allow ejb-ref and ejf-local-ref elements to be added they are still, according to the EJB specification, remote clients to the EJB's. Some servers, however, support the definition of ejb-refs or ejb-local-refs in the descriptors of

these components, which will enable invocation of referenced EJB's. This allowance takes into consideration the fact that these components are all co-located and that local entity access could occur.

A stricter and more portable approach would be to retain the session façade layer described in the above section. This would require the JSP to remotely access a session bean, which in turn would invoke the local entities. This layered J2EE structure divides the responsibilities of the system up into its various components, and allows substitution or upgrading of a single component with little or no effect on the others.

Figure A–4 Thin client architecture



Dependent Lightweight Objects

There are a good many applications that make use of persistent data objects that are not autonomous, or that do not require their own transactional and security mechanisms. Instead they would be better suited to share, or "piggy-back" on top of the mechanisms of an existing object upon which they rely. This secondary or reliant object is called a dependent object.

Dependent objects may take one of three forms. They may appear as local entity beans residing only on the server. They may be simple persistent TopLink-enabled Java objects that can be shipped back and forth between the client and server. They may even be serializable "dependent value" objects that are persisted by simply attaching them to entities and letting their contents be serialized when the entity gets written back to its persistent store. Each of these three strategies has its advantages and may apply more readily in certain applications.

Local Entities

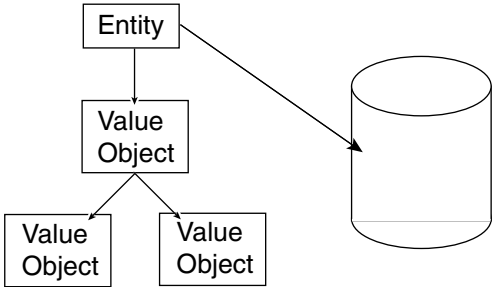
Strictly speaking, EJB local entities should not be considered lightweight since all of the container security, transactional and synchronization costs are still very much in effect. They are included in this category chiefly because they are more lightweight than remote entities and because they can, in practice be used as dependent objects. But they are not considered to be truly "dependent", in the sense that they can be "found" (using finder methods on the home interface) by any server-side component regardless as to whether they are related to the object or not.

Because EJB Containers can assume that local entities are always co-located in the same VM, communication optimizations can be effected by the Container. Local entity methods can be invoked using pass-by-reference semantics, alleviating unnecessary marshalling and communications infrastructure costs. This makes them lighter than remote entities both in the amount of code that gets generated by the Container, and the execution time of each method invocation.

Dependent Value Objects

The EJB 2.0 specification discusses a class of object called a dependent value object that can be assigned to an entity attribute. They are nothing more than regular serializable Java objects and must be serialized in their entirety and written out in BLOB form when any part of them changes. It is an integral part of the entity since it is stored as a "cmp" field in the entity. Although it can be sent across from client to server and back again, its persistence is coupled with the entity and it is not practical when multiple levels of dependent objects are required.

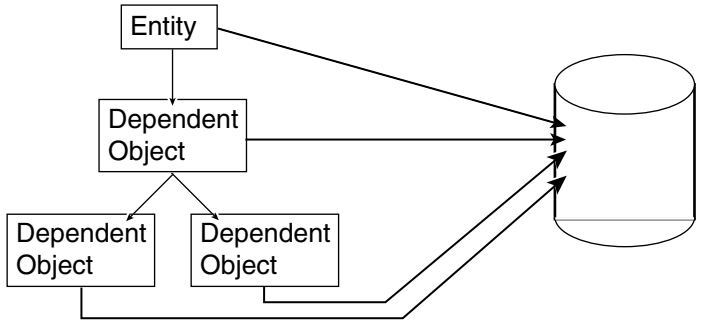
Figure A–5 Using dependent value objects in a system



Dependent Java Objects

Because of TopLink's ability to persist regular Java objects without the need for EJB container support these objects offer the most flexibility. They do not incur the entity costs of container management but can be persisted independent of the entity. This means that changes are detected at commit-time, but if nothing changes during the course of an entity update then the object will not be written back, and likewise only the object may be written out if the reverse is true. The dependency upon the entity is still intact, however, as any removal of the entity will automatically propagate to cause the dependent Java object to be removed from persistent storage. Like serializable value objects, they can be transported back and forth between the client and server, allowing for client interactions that refer to the owning bean, but operate on the dependent data. These objects are not supported in the EJB 2.0 specification, but do offer the benefits of managed, mapped persistence.

Figure A–6 Using dependent Java objects in a system



Conclusion

EJB provides developers with a great deal of infrastructure that makes building enterprise applications easier. This allows developers to build better applications by allowing them to focus on the business logic of their application rather than on distribution, security, and transactions. Even with everything that EJB provides, developing with EJB requires intelligent architectural choices to be made. Although EJB provides much, it also allows for flexibility so that developers can use it to meet their needs.

Regardless of the EJB architecture used, TopLink will support it by providing the right level of control and transparence appropriate to the architecture. The TopLink persistence framework also adds the value required to customize applications to

run at their best, and will play an essential role in enabling customers to successfully develop and deploy their enterprise applications.

B

The toplink-ejb-jar DTD

This appendix offers a listing of the `toplink-ejb-jar` document type description (DTD).

DTD listing

```
<!ELEMENT toplink-ejb-jar (session)>

<!-- The element that describes the TopLink session for the beans in a
particular jar.
Example:
<toplink-ejb-jar>
  <session>
    <name>ejb20_AccountDemo</name>
    <project-class>oracle.toplink.demos.ejb20.cmp.account.
    AccountProject</project-class>
    <login>
      <connection-pool>ejbPool</connection-pool>
    </login>
    <customization-class>oracle.toplink.demos.ejb20.cmp.account.
    AccountCustomizer</customization-class>
  </session>
</toplink-ejb-jar>
Used in: toplink-ejb-jar
-->
<!ELEMENT session (
  name,
  (project-class | project-xml),
  login,
  cache-synchronization?,
  use-remote-relationships?,
  customization-class?
```

```
)>

<!-- The unique name that will be used to identify this session.
  Valid values: User-chosen name unique amongst all deployed jars
  Used in:  session
-->
<!ELEMENT name (#PCDATA)>

<!-- The class name that this session will load to provide mapping information.
  Valid values: fully-qualified class name
  Example: <project-class>oracle.toplink.demos.ejb20.cmp
 .account.AccountProject</project-class>
  Used in: session
-->
<!ELEMENT project-class (#PCDATA)>

<!-- The XML project that this session will load to provide mapping information.
The xml can (and should) be located
     in the deployable jar file in order simplify portability across machines.
  Valid values:  A fully-qualified, or relative file name
  Example: <project-xml>META-INF/AccountProject.xml</project-xml>
  Used in: session
-->
<!ELEMENT project-xml (#PCDATA)>

<!-- Used to specify the login parameters such as the data source.
  Example:
    // using a connection pool
    <login>
      <connection-pool>ejbPool</connection-pool>
    </login>

    // using datasources
    <login>
      <datasource>ejbDataSourceJTS</datasource>
      <non-jts-datasource>ejbDataSourceNonJTS</non-jts-datasource>
    </login>

  Used in: session
-->
<!ELEMENT login (
  (connection-pool | (datasource, non-jts-datasource)),
  should-bind-all-parameters?,
  uses-byte-array-binding?,
  uses-string-binding?)>
```

```
<!-- The name of the WebLogic connection pool used by TopLink.
  Valid values:  Name of connection pool defined in WLS
  Used in: login
-->
<!ELEMENT connection-pool (#PCDATA)>

<!-- The name of the transactional data source used by TopLink to get db
     connections to use for writing.
  Valid values:  Name of transactional data source defined in WLS
  Used in: login
-->
<!ELEMENT datasource (#PCDATA)>

<!-- The name of the non-transactional data source used by TopLink to get db
     connections to use for reading.
  Valid values:  Name of non-transactional data source defined in WLS
  Used in: login
-->
<!ELEMENT non-jts-datasource (#PCDATA)>

<!-- Set to true if all queries should use parameter binding.
  Valid values:  "True", "False"
  Used in: login
-->
<!ELEMENT should-bind-all-parameters (#PCDATA)>

<!-- Set to true if byte arrays should be bound.
  Valid values:  "True", "False"
  Used in: login
-->
<!ELEMENT uses-byte-array-binding (#PCDATA)>

<!-- Set to true if strings should be bound.
  Valid values:  "True", "False"
  Used in: login
-->
<!ELEMENT uses-string-binding (#PCDATA)>

<!-- Configure how TopLink synchronizes its caches across all
     local and remote sessions.
  Example:
    <cache-synchronization>
      <is-asynchronous>True</is-asynchronous>
      <should-remove-connection-on-error>True</should-remove-connection-on-error
```

```
>
    </cache-synchronization>

  Used in: session
-->
<!ELEMENT cache-synchronization (is-asynchronous?,
should-remove-connection-on-error?)>

<!-- Set to true if synchronization should not wait until all sessions have been
     synchronized before returning.
  Valid values:  "True", "False"
  Used in: cache-synchronization
-->
<!ELEMENT is-asynchronous (#PCDATA)>

<!-- Set to true if a synchronization connection should be removed from
     the session if a communication error occurs.
  Valid values:  "True", "False"
  Used in: cache-synchronization
-->
<!ELEMENT should-remove-connection-on-error (#PCDATA)>

<!-- Defines whether the Session should use remote relationships for
     all bean-to-bean relationships.
  Valid values: "True", "False"
  Default: "False"
  Example:
    <use-remote-relationships>True</use-remote-relationships>
  Used in: session
-->
<!ELEMENT use-remote-relationships (#PCDATA)>

<!-- The class that will be called to customize the session
     before and after login.
  Valid values:  A fully-qualified class name
  Example:
    <customization-class>oracle.toplink.demos.ejb20.cmp
 .account.AccountCustomizer</customization-class>
  Used in: session
-->
<!ELEMENT customization-class (#PCDATA)>
```

Index

A

B

C

D

E

F

H

I

J

L

M

N

O

P

Q

R

S

T

U

W