SYBASE®

Enterprise JavaBeans User's Guide

EAServer

6.0

DOCUMENT ID: DC00428-01-0600-01

LAST REVISED: July 2006

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book ..... **vii**

CHAPTER 1 **Enterprise JavaBeans Overview** **1**
- About Enterprise JavaBeans components 1
  - EJB component types 3
  - EJB container services 5
  - EJB transaction settings 6
- EAServer EJB support 8
  - Deploying EJB components to EAServer 9
  - EJB clients connecting to EAServer 9
  - For more information 9
- EJB version levels 10
  - EJB 2.1 differences from 2.0 10
  - EJB 2.0 differences from 1.1 14
  - EJB 1.1 differences from EJB 1.0 16

CHAPTER 2 **Deploying and Configuring EJB Components** **19**
- Deploying an EJB-JAR file 19
- Configuring EJB component properties 20
  - Structure of the configuration file 21
  - Updating component properties 22
- Commonly configured properties 24
  - ejb.accessControl 25
  - ejb.allowedPorts 27
  - ejb.rolePrefix 27
  - ejb.automaticFailover 27
  - ejb.clusterPartition 28
  - ejb.copyValues 28
  - ejb.logExceptions 29
  - ejb.enableProfiling 29
  - ejb.enableTracing 30
  - ejb.isolationLevel 30
  - ejb.localNamePrefix 34

ejb.localNameSuffix ... 34
ejb.localThreadMonitor ... 34
ejb.remoteNamePrefix ... 35
ejb.remoteNameSuffix ... 35
ejb.remoteThreadMonitor ... 35
ejb.serviceThreadMonitor ... 36
ejb.transactionBatch ... 36
ejb.transactionRetry ... 37
ejb.passivateTimeout ... 38
ejb.removeTimeout ... 38
ejb.poolTimeout ... 39
jca.connectionFactory ... 39
sql.createTables ... 39
sql.dataSource ... 40
sql.isolationLevel ... 41

CHAPTER 3 Developing EJB Clients ... 43
Client runtime requirements ... 43
EJB client program flow ... 44
Instantiating home interface proxies ... 45
Obtaining an initial naming context ... 45
Resolving JNDI names ... 51
Instantiating remote or local interface proxies ... 52
Calling remote interface methods ... 54
Calling local interface methods ... 54
Managing transactions ... 55
Serializing and deserializing bean proxies ... 56
Using EJB providers ... 57
Running EAServer 5.x clients against 6.0 or later servers ... 59

CHAPTER 4 Creating Application Clients ... 61
About application clients ... 61
Deploying application clients ... 62
Configuring application client properties ... 64
General tab ... 64
Configuration tab ... 65
Advanced tab ... 65
Running application clients ... 65
Setting up a client's workstation ... 65
Starting the runtime container ... 66

CHAPTER 5 Interoperability ... 67

Intervendor EJB interoperability 67
Interoperable naming URLs 69
Classes for RMI/IIOP connections from third-party containers 70
Invoking EJB components from PowerBuilder clients 70
Invoking EJB components from CORBA/C++ clients 71
Supported datatypes 71
Generating required header files 72
Calling the home interface 73
Serializing and deserializing instance references 75

CHAPTER 6 Tutorial: Creating Enterprise JavaBeans Components and Clients 77
Overview of the sample components 77
Tutorial requirements 77
Creating the application 78
Start EAServer and the Management Console 78
Create the EJB-JAR file 79
Deploy the EJB-JAR file to EAServer 80
Create the glossary database and data source 81
Create a user account 85
Create the client application 85
Run the client application 85
Automating deployment and configuration 87

Index 89

About This Book

Subject

This book describes how to deploy and configure Enterprise JavaBeans (EJB) components to EAServer and create Enterprise JavaBeans clients that connect to EAServer.

Audience

This book is intended for EJB developers and application providers as well as administrators that deploy packaged EJB applications to EAServer.

How to use this book

Chapter 1, “Enterprise JavaBeans Overview,” summarizes the Enterprise JavaBeans component model and how it is supported in EAServer.

Chapter 2, “Deploying and Configuring EJB Components,” describes how to deploy EJB components to EAServer and configure EJB modules after deployment.

Chapter 3, “Developing EJB Clients,” describes the EJB client model supported by EAServer.

Chapter 4, “Creating Application Clients,” describes how to deploy EJB clients packaged using the J2EE application client packaging model.

Chapter 5, “Interoperability,” describes interoperability between EAServer and other EJB application servers, and interoperability between EJB components and other component models supported by EAServer.

Chapter 6, “Tutorial: Creating Enterprise JavaBeans Components and Clients,” walks you through the creation of sample EJB entity and session beans and an EJB client.

Related documents

Core EAServer documentation The core EAServer documents are available in HTML and PDF format in your EAServer software installation and on the SyBooks™ CD.

What’s New in EAServer 6.0 summarizes new functionality in this version.

The *EAServer API Reference Manual* contains reference pages for proprietary EAServer Java classes and C routines.

The *EAServer Automated Configuration Guide* explains how to use Ant-based configuration scripts to:

- Define and configure entities, such as EJB modules, Web applications, data sources, and servers
- Perform administrative and deployment tasks

The *EAServer CORBA Components Guide* explains how to:

- Create, deploy, and configure CORBA and PowerBuilder™ components and component-based applications
- Use the industry-standard CORBA and Java APIs supported by EAServer

The *EAServer Enterprise JavaBeans User's Guide* (this book) describes how to:

- Configure and deploy EJB modules
- Develop EJB clients, and create and configure EJB providers
- Create and configure applications clients
- Run the EJB tutorial

The *EAServer Feature Guide* explains application server concepts and architecture, such as supported component models, network protocols, server-managed transactions, and Web applications.

The *EAServer Java Message Service User's Guide* describes how to create Java Message Service (JMS) clients and components to send, publish, and receive JMS messages.

The *EAServer Migration Guide* contains information about migrating EAServer 5.*x* resources and entities to an EAServer 6.0 installation.

The *EAServer Performance and Tuning Guide* describes how to tune your server and application settings for best performance.

The *EAServer Security Administration and Programming Guide* explains how to:

- Understand the EAServer security architecture
- Configure role-based security for components and Web applications
- Configure SSL certificate-based security for client connections
- Implement custom security services for authentication, authorization, and role membership evaluation
- Implement secure HTTP and IIOP client applications

- Deploy client applications that connect through Internet proxies and firewalls

The *EAServer System Administration Guide* explains how to:

- Start the preconfigured server and manage it with the Sybase Management Console
- Create, configure, and start new application servers
- Define database types and data sources
- Create clusters of application servers to host load-balanced and highly available components and Web applications
- Monitor servers and application components
- Automate administration and monitoring tasks with command line tools

The *EAServer Web Application Programming Guide* explains how to create, deploy, and configure Web applications, Java servlets, and JavaServer Pages.

The *EAServer Web Services Toolkit User's Guide* describes Web services support in EAServer, including:

- Support for standard Web services protocols such as Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Uniform Description, Discovery, and Integration (UDDI)
- Administration tools for deployment and creation of new Web services, WSDL document creation, UDDI registration, and SOAP management

The *EAServer Troubleshooting Guide* describes procedures for troubleshooting problems that EAServer users may encounter. This document is available only online; see the EAServer Troubleshooting Guide at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.eas_5.2.eastg/html/eastg/title.htm.

jConnect for JDBC documents EAServer includes the jConnect™ for JDBC™ 6.0.5 driver to allow JDBC access to Sybase database servers and gateways. The *jConnect for JDBC 6.0.5 Programmer's Reference* is available on the Sybase Product Manuals Web site at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.jconnjdbc_6.05.prjdbc/html/prjdbc/title.htm&toc=/com.sybase.help.jconnjdbc_6.05/toc.xml.

Sybase Software Asset Management User's Guide EAServer includes the Sybase Software Asset Management license manager for managing and tracking your Sybase software license deployments. The *Sybase Software Asset Management User's Guide* is available on the Getting Started CD and in the EAServer 6.0 collection on the Sybase Product Manuals Web site at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.eas_6.0/title.htm.

Conventions

The formatting conventions used in this manual are:

Formatting example	To indicate
commands and methods	When used in descriptive text, this font indicates keywords such as: • Command names used in descriptive text • C++ and Java method or class names used in descriptive text • Java package names used in descriptive text • Property names in the raw format, as when using Ant or jagtool to configure applications rather than the Management Console
variable, *package*, or *component*	Italic font indicates: • Program variables, such as *myCounter* • Parts of input text that must be substituted, for example: *Server*.log • File names • Names of components, EAServer packages, and other entities that are registered in the EAServer naming service
File | Save	Menu names and menu items are displayed in plain text. The vertical bar shows you how to navigate menu selections. For example, File | Save indicates "select Save from the File menu."
`package 1`	Monospace font indicates: • Information that you enter in the Management Console, a command line, or as program text • Example program fragments • Example output fragments

Other sources of information

Use the Sybase Getting Started CD, the SyBooks CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.

- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

  Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

  Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.

- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://sybooks.sybase.com/nav/base.do.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**

1 Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.

2 Select Products from the navigation bar on the left.

3 Select a product name from the product list and click Go.

4 Select the Certification Report filter, specify a time frame, and click Go.

5 Click a Certification Report title to display the report.

❖ **Creating a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1 Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.

2 Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ **Finding the latest information on EBFs and software maintenance**

1. Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.
2. Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
3. Select a product.
4. Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

   Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the "Technical Support Contact" role to your MySybase profile.
5. Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Accessibility features

EAServer has been tested for compliance with U.S. government Section 508 Accessibility requirements. The online help for this product is also provided in Eclipse help formats, which you can navigate using a screen reader.

The Web console supports working without a mouse. For more information, see "Keyboard navigation" in Chapter 2, "Management Console Overview," in the *EAServer System Administration Guide*.

The Web Services Toolkit plug-in for Eclipse supports accessibility features for those that cannot use a mouse, are visually impaired, or have other special needs. For information about these features see the Eclipse help:

1. Start Eclipse.
2. Select Help | Help Contents.
3. Enter `Accessibility` in the Search dialog box.

4 Select Accessible User Interfaces or Accessibility Features for Eclipse.

Note You may need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For additional information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1

Enterprise JavaBeans Overview

EAServer supports version 2.1 of the EJB specification. For compatibility with previous EAServer versions and other EJB servers, EAServer supports earlier EJB versions, from EJB 1.1 onwards.

For more details on the EJB architecture, see the EJB specifications from Sun Microsystems at http://java.sun.com/products/ejb/.

Topic	Page
About Enterprise JavaBeans components	1
EAServer EJB support	8
EJB version levels	10

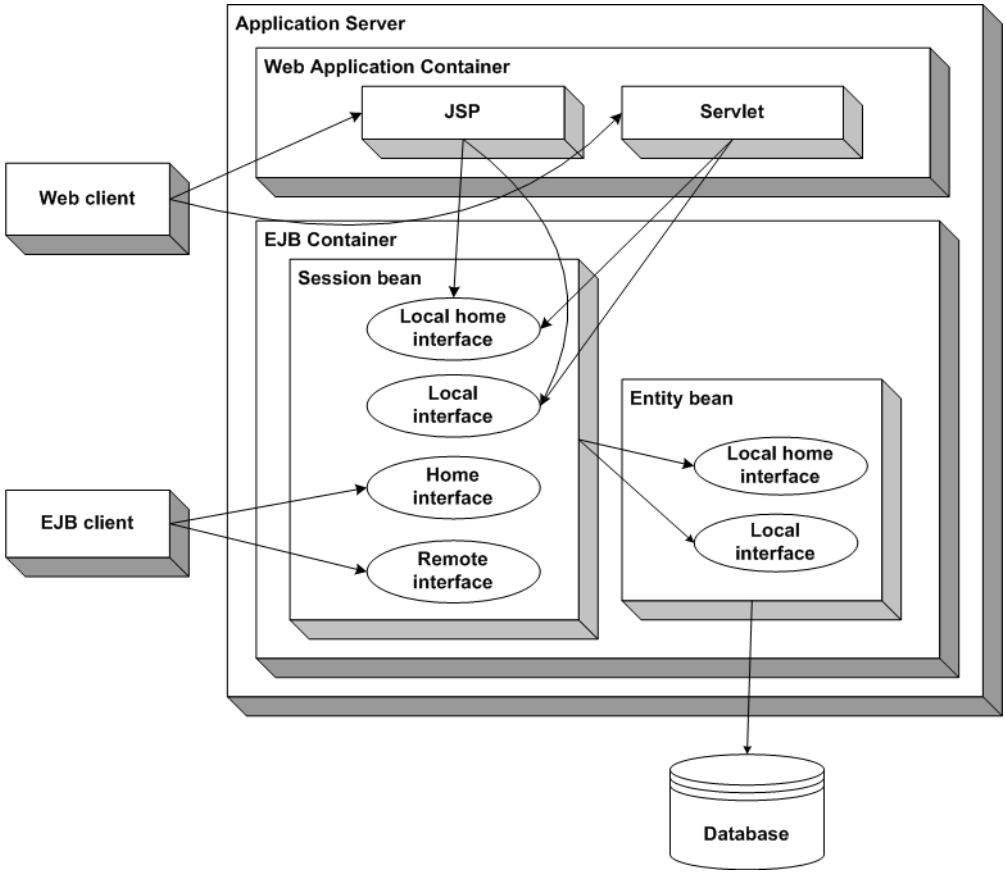
About Enterprise JavaBeans components

The Enterprise JavaBeans (EJB) technology defines a model for the development and deployment of reusable Java server components, called **EJB components**.

An EJB component is a nonvisual server component with methods that typically provide business logic in distributed applications. A remote client, called an EJB client, can invoke these methods, which typically results in the updating of a database. Since EAServer uses CORBA IIOP protocols, EJB components running in EAServer can be called by any other type of EAServer client or component, and even CORBA clients using ORBs from other vendors that are compatible with CORBA 2.3.

Figure 1-1 shows the components in the EJB architecture.

Figure 1-1: EJB Architecture



Application server The application server contains the EJB container and can contain other features such as a Web application container to host Web components such as JavaServer Pages (JSPs) and Java servlets. EAServer is an application server.

EJB container The EJB container runs EJB components and provides required services such as enforcing application security constraints, managing transactions that involve multiple components and databases, and providing the naming service that maps components and resources to logical names. The container provides the infrastructure required to run distributed components, allowing client and component developers to focus on programming business logic, and not system-level code.

EJB client An EJB client usually provides the user-interface logic on a client machine. The EJB client makes calls to remote EJB components on a server and needs to know how to find the EJB server and how to interact with the EJB components. EJB components and Web components can act as an EJB client by calling methods in another EJB component.

Session Beans and Entity Beans EJB components can be of two types. Most applications include *session beans* to be invoked directly by clients. Session beans provide the interface between base clients and the server-side implementation of the application. *Entity beans* provide an abstraction of an object that is stored in a database. *Entity beans* are not typically invoked directly by base clients. “EJB component types” on page 3 describes the EJB component types in more detail.

All types of EJB are implemented by a Java class that contains implementations of the remote interface methods and additional methods for lifecycle management.

Client interfaces An EJB client does not communicate directly with an EJB component. The container provides proxy objects that implement the components home and remote interfaces. The component’s *remote interface* defines the business methods that can be called by the client. The client calls the *home interface* methods to create and destroy proxies for the remote interface. Clients use the Java name service to resolve components; the name service maps the component’s logical name to a proxy objects that implement the home and remote interfaces. The proxies contain generated code required to invoke the component over the network.

Beginning in EJB version 2.0, EJB and Web components running in the same server can execute EJB components using the component’s *local interface*. Using the local interface can improve performance. If an EJB component is not meant to be invoked directly by client applications, you can include only local interfaces for the component.

EJB component types

You can implement four types of EJB component, each for a different purpose:

- Stateful session beans
- Stateless session beans
- Entity beans
- Message driven beans

Stateful session beans

A stateful session bean manages complex processes or tasks that require the accumulation of data, such as adding items to a Web catalog's shopping cart. Stateful session beans have the following characteristics:

- They manage tasks that require more than one method call to complete, but are relatively short-lived. For example, a session bean might manage the process of making an airline reservation.
- They typically store session state information in class instance data, and do not survive server crashes unless they are run in a cluster that has persistent storage enabled for the component.
- There is an affinity between each instance and one client from the time the client creates the instance until it is destroyed by the client or by the server in response to an expired instance timeout limit.

For example, if you create a session bean on a Web server that tracks a user's path through the site, the session bean is destroyed when the user leaves the site or idles beyond a specified time.

Stateless session beans

A stateless session bean manages tasks that do not require the keeping of client session data between method calls. Stateless session beans have the following characteristics:

- Method invocations do not depend on data stored by previous method invocations.
- There is no affinity between a component instance and a particular client. Each call to a client's proxy may invoke a different instance.
- From the client's perspective, different instances of the same component are identical.

Unlike stateful session beans, stateless session beans can be pooled by the server, improving overall application performance.

Entity beans

An entity bean models a business concept that is a real-world object. For example, an entity bean might represent a scheduled airplane flight, a seat on the airplane, or a passenger's frequent-flyer account. Entity beans have the following characteristics:

- Each instance represents a row in a persistent database relation, such as a table, view, or the results of a complex query.
- The bean has a primary key that corresponds to the database relation's key, and is represented by a Java datatype or class.

Message-driven beans

A Message Driven Bean (MDB) responds asynchronous messages delivered by the application server's message engine. The MDB model integrates the EJB component architecture with the Java Message Service (JMS) asynchronous messaging API and other messaging APIs.

An MDB component is similar to an EJB stateless session bean, but the MDB component responds only to asynchronous messages and has no direct client interface. Rather than having remote or local interfaces, an MDB that responds to JMS messages must implement the javax.jms.MessageListener interface. EAServer calls the onMessage method to deliver messages to the MDB.

An MDB must be attached to a message source such as a JMS message queue or topic. You can invoke the MDB asynchronously by posting a message to this queue or topic.

For more information, see "Message-driven beans" in the *JMS User's Guide*.

EJB container services

The EJB container provides services to EJB components as listed below.

Transaction support An EJB container must support transactions. EJB specifications provide an approach to transaction management called declarative transaction management. In declarative transaction management, you specify the type of transaction support required by your EJB component. When the bean is deployed, the container provides the necessary transaction support.

Persistence support An EJB container can provide support for persistence of EJB components. An EJB component is persistent if it is capable of saving and retrieving its state. A persistent EJB component saves its state to some type of persistent storage (usually a file or a database). With persistence, an EJB component does not have to be re-created with each use.

An EJB component can manage its own persistence (by means of the logic you provide in the bean) or delegate persistence services to the EJB container. Container-managed persistence means that the data appears as member data and the container performs all data retrieval and storage operations for the EJB component.

Naming support An EJB container must provide an implementation of Java Naming and Directory Interface (JNDI) API to provide naming services for EJB clients and components. Naming services provide:

- **Location transparency** Clients can instantiate components by name, and do not need to know the details about the server hosting the component.
- **Deployment flexibility** Beginning in EJB version 1.1, EJB components can be configured with naming aliases for components and resources such as databases, JavaMail sessions, and JMS message queues. Using aliases simplifies the procedure to deploy the component on a server where the accessed components and resources use different JNDI names.

EJB transaction settings

In the deployment descriptor you must configure the transaction settings to specify whether EAServer manages transactions automatically, based on the method transaction attributes, or the bean manages transactions programmatically.

Container-managed transactions versus bean-managed transactions

When you design an EJB component, you must decide how the bean will manage transaction demarcation: either programmatically in the business methods, or whether the transaction demarcation will be managed by the container based on the value of the transaction attribute in the deployment descriptor.

Session beans and message-driven beans can use either bean-managed transaction demarcation or container-managed transaction demarcation; you cannot create a session bean where some methods use container-managed demarcation and others use bean-managed demarcation. An entity bean must use container-managed transaction demarcation.

When using bean-managed transactions, you must explicitly begin, commit, and roll back new, independent transactions by using the javax.transaction.UserTransaction interface. Transactions begun by the component execute independently of the client's transaction. If the component has not begun a transaction, the component's database work is performed independently of any EAServer transaction.

Method transaction attributes

When using container-managed transactions, the method transaction attributes specify how the component participates in transactions. When a client or another component calls a method, the transaction attribute determines whether the method executes in the same transaction context, in a new transaction context, or cannot execute in transactions at all.

Table 1-1 lists the transaction attribute values. Requires, Supports, Requires New, or Mandatory are the values that specify container-managed transaction demarcation. You can set the Transaction Attribute for the component and for individual methods in the home and remote interfaces. Values set at the method level override the component setting.

Table 1-1: Transaction attribute values

Attribute	Description
NotSupported	(The component-level default.) The EJB component's methods never execute as part of a transaction. If the EJB component is activated by a client that has a pending transaction, the EJB component's work is performed outside the existing transaction. Since entity beans are almost always involved in transactions, this value is not usually used for an entity bean.
Supports	The EJB component can execute in the context of an EAServer transaction, but a transaction is not required to execute the component's methods. If a method is called by a base client that has a pending transaction, the method's database work occurs in the scope of the client's transaction. Otherwise, the EJB component's database work is done outside of any transaction.
Required	The EJB component always executes in a transaction. Use this option when your EJB component's database activity needs to be coordinated with other components, so that all components participate in the same transaction.
RequiresNew	Whenever the EJB component is instantiated, a new transaction begins.
Mandatory	EJB component methods must be called in the context of a pending transaction. If a client calls a method without an open transaction, the EAServer ORB throws an exception.
Never	The component's methods never execute as part of a transaction, and the component may cannot be called in the context of a transaction. If a client or another component calls the component with an outstanding transaction, EAServer throws an exception.

EAServer EJB support

EAServer can host Enterprise JavaBeans (EJB) components developed according to version 2.1, 2.0, or 1.1 of the EJB specification, in other words, all EJB versions between 1.1 and 2.1, inclusive.

Deploying EJB components to EAServer

EJB components can be deployed in an EJB-JAR file. The EJB-JAR file packages the component classes and a *deployment descriptor* in the standard Java archive (JAR) format. The deployment descriptor describes the components included in the JAR and sets standard properties such as the home and remote interface names, transaction attribute values, required user roles, and so forth.

You can use several tools to create EJB-JAR files, including Ant, Eclipse, and popular IDEs such as Borland JBuilder. You can deploy the EJB-JAR to EAServer using several tools including:

- The Management Console
- The deploy utility
- The jagant and jagtool utilities.

EAServer also supports the Enterprise JavaBeans client model. You can generate EJB-style proxies for any IDL interface, and use the proxies to call methods on components that implement that interface.

EJB clients connecting to EAServer

EAServer also supports the Enterprise JavaBeans client model by generating EJB proxies and providing an EJB-compliant implementation of the JNDI Context interface. The Context implementation allows clients to connect to EAServer, look up proxy stubs for the home interface, and invoke EJB components. In the server, EJB and Web components also use this same API for intercomponent calls.

For more information

For information about	See this chapter or section
Creating, importing, and exporting EJB components.	Chapter 2, “Deploying and Configuring EJB Components”
Creating EJB clients, generating EJB stubs, instantiating home and remote interface proxies, managing transactions, and serializing and deserializing bean proxies.	Chapter 3, “Developing EJB Clients”

For information about	See this chapter or section
Invoking non-EJB components from EJB clients and invoking EJB components from non-EJB clients, and using EAServer with EJB 2.0 containers from other vendors.	Chapter 5, “Interoperability”
Running the EJB tutorial	Chapter 6, “Tutorial: Creating Enterprise JavaBeans Components and Clients”

EJB version levels

The current version of the EJB specification supported by EAServer is 2.1. Past EJB versions include 2.0, 1.1, and 1.0. If migrating from an earlier version, consider the version differences described below.

EJB 2.1 differences from 2.0

EJB 2.1 introduces the following enhancements:

- Web services support
- Message driven beans support additional message types
- Timer service
- EJB-QL enhancements

Web services support

In EJB 2.1, stateless session beans can be exposed as Web services. EJB components of any type can declare Web service references to alias a Web service proxy to a JNDI name.

Exposing stateless session beans as Web services

This feature allows a stateless session bean to be invoked using standard Web services protocols, specifically XML-based web service invocations using WSDL 1.1 and SOAP 1.1 over HTTP 1.1 in conformance with the requirements of the JAX-RPC specification.

To expose a stateless session bean as a Web service, you must:

- Define a Web service endpoint interface that satisfies the requirements described below.
- Declare the endpoint interface in the deployment descriptor using a `service-endpoint` element.

The *Web service endpoint interface* defines the methods in the bean that can be invoked by Web services clients. Each method in the endpoint interface describes a Web service operation to be exposed in the WSDL interface for the Web service.

The mapping of EJB methods to WSDL interface operations is based on the JAX-RPC specification. Specifically, the Bean implementation and interface must satisfy these requirements:

- The endpoint interface must extend the java.rmi.Remote interface.
- The methods in the endpoint interface must follow the rules for JAX-RPC service endpoint interfaces. Specifically:
    - Their argument and return values must be of valid types for JAX-RPC, and their throws clause must include java.rmi.RemoteException (in addition to the application exceptions to match those thrown by the equivalent bean implementation method).
    - You cannot pass references to the bean instance, other beans, or other object references that are not valid outside the scope of the bean's implementation. These references include EJB instance references (class EJBObject), EJB local instance references (EJBLocalObject), timer and timer handle references, local, remote, home, and local home classes, and managed collections that are used for entity beans with container-managed persistence. These types cannot be used as a parameter, return type, or in arrays or complex types used as parameters and return types.

    To support WSDL output and input-output operations, use JAX-RPC holder classes. Holder classes implement the javax.xml.rpc.holders.Holder interface.
- The web service endpoint interface must not include constant (as `public final static`) declarations.

- The bean implementation class must contain an equivalent method for each method in the endpoint interface. The methods must have the same name, take the same argument list, and return the same type. The service interface method must throw all exceptions listed in the throws clause of the implementation method (in addition to java.rmi.RemoteException).

Using Web-services references

EJB components of any type can declare Web service references to alias a Web service proxy to a JNDI name.

In the EJB-JAR file, declare Web service references using the `service-ref` element as shown in the following example:

```
<service-ref>
  <description>
  This is a reference to the stock quote
  service used to estimate portfolio value.
  </description>
  <service-ref-name>service/StockQuoteService</service-ref-name>
  <service-interface>com.example.StockQuoteService
  </service-interface>
</service-ref>
```

In the corresponding implementation code, the Web service proxy is bound to the name *java:comp/env/service/StockQuoteService* and can be retrieved as shown in the Java code below:

```
// Obtain the default initial JNDI context.
Context initCtx = new InitialContext();

// Look up the stock quote service in the environment.
com.example.StockQuoteService sqs =
    (com.example.StockQuoteService)initCtx.lookup(
    "java:comp/env/service/StockQuoteService");

// Get the stub for the service endpoint
com.example.StockQuoteProvider sqp =
sqs.getStockQuoteProviderPort();

// Get a quote
float quotePrice = sqp.getLastTradePrice(...);
```

After deploying an EJB-JAR that contains Web service references, you must configure the references to specify a local implementation of the Web service proxy class and other details required to connect to the Web service's host server.

Message driven beans support additional message types

In EJB 2.1, message driven beans support other message types besides JMS. You can provide a J2EE Connector Architecture (JCA) 1.5 resource adapter to pull messages from other messaging systems. For more information, see "Message-driven beans" in the *JMS User's Guide*.

Timer service

The EJB Timer Service provides methods to allow callbacks to be scheduled for time-based events. Stateless session beans and entity beans may call the timer API to register for timer event notifications. Notifications may be scheduled to occur at a specific time, after a specific elapsed duration, or at specific recurring intervals.

The EJB Timer Service is a coarse-grained timer notification service that is designed for use in the modeling of application-level processes. It is not intended for the modeling of real-time events. While timer durations are expressed in millisecond units, this is because the millisecond is the unit of time granularity used by the APIs of the J2SE platform. It is expected that most timed events will correspond to hours, days, or longer.

An enterprise bean accesses the timer service through its EJBContext interface. The timer service provides methods for the creation and cancellation of timers, as well as for locating the timers that are associated with a bean. A timer is created to schedule timed callbacks. The bean class of an enterprise bean that uses the timer service must implement the javax.ejb.TimedObject interface. This interface has a single method, the timer callback method, ejbTimeout. When the time specified at timer creation elapses, the container invokes the ejbTimeout method of the bean. A timer may be cancelled by a bean before its expiration. If a timer is cancelled, the ejbTimeout method is not called. A timer is cancelled by calling its cancel method.

Invocations of the methods to create and cancel timers and of the ejbTimeout method are typically made within a transaction. The timer service is intended for the modelling of long-lived business processes. Timers survive container crashes and the activation/passivation and load/store cycles of the enterprise beans that are registered with them.

EJB-QL enhancements

For EJB 2.1 EJB-QL has been enhanced with:

- Addition of ORDER BY clause

- Addition of Aggregate Functions AVG, MIN, MAX, SUM, COUNT to SELECT clause
- Addition of MOD Function to WHERE clause
- Clarification of requirements for returning null values from queries and finder and select methods

EJB 2.0 differences from 1.1

EJB 2.0 introduces support for message driven beans, new home interface method syntax, local interfaces, and inter-vendor interoperability. EJB 2.0 also enhances the container managed persistence model defined in EJB 1.1.

Message-driven beans

EJB 2.0 integrates the EJB component architecture with the Java Message Service (JMS) asynchronous messaging API. EJB 2.0 allows you to define message-driven bean (MDB) components to respond to JMS messages. An MDB component is similar to an EJB stateless session bean, but the MDB component responds only to JMS messages and has no direct client interface.

For information on JMS, see For more information, see “Message-driven beans” in the *JMS User’s Guide*.

Home interface methods

EJB 2.0 allows you to define business methods in the home interface for an entity bean and changes the syntax of create methods.

Create method syntax

Previously, create methods were restricted to methods named create. In EJB 2.0, you can use any name that begins with create, such as createNewAccount.

Home interface business methods

You can add business methods to the home interface for an entity bean to perform operations that are not specific to a single instance. For example, a home business method might return the average employee salary. For each home business method, the entity bean’s implementation class must have a method with the same name, except for the prefix ejbHome, and the same signature. For example, if the home interface declares:

```
public double averageSalary();
```

Then the implementation class must contain:

```
public double ejbHomeAverageSalary();
```

Local interfaces

The EJB 2.0 architecture introduces local interfaces for calls to an EJB component from within the same Java Virtual Machine. In EAServer, you can use local interfaces for intercomponent calls, and for component invocations made from servlets and JSPs hosted by the same server as the component. To use local interfaces, you must configure a local EJB reference for the JSP or EJB component that issues the call.

Using local interfaces can improve performance for calls to components hosted in the same server, but in coding you must be aware of the restrictions listed in "Calling local interface methods" on page 54.

CMP enhancements

EJB 2.0 enhances the Container Managed Persistence (CMP) model for entity beans as follows:

- The deployment descriptor more fully describes the persistent fields in the bean and the required database queries, making for less work after deploying an EJB JAR file that contains CMP entity beans.
- CMP entity beans in the same EJB JAR (which maps to an EAServer module) can have container-managed relationships. For example, an *Order* bean may have an *items* field that consists of a collection of *Inventory* bean instances representing the items being purchased. Or, an *Employee* bean may be related to itself, with *manager* and *employees* fields that contain Employee instances.

EJB 2.0 interoperability

EAServer 4.0 complies with the interoperability requirements in the EJB 2.0 specification to allow interoperability with other EJB 2.0 servers. EAServer continues to support CORBA-2.2 based interoperability, for interacting with other CORBA-based application servers and to allow interoperability between EJB components hosted by EAServer and EAServer components of other types. For more information, see Chapter 5, "Interoperability."

EJB 1.1 differences from EJB 1.0

The main change in EJB 1.1 involves the packaging of components. EJB 1.1 uses an XML deployment descriptor, and allows abstraction of container-specific resource references used within the source code. In addition, there are minor changes to the Java interfaces and classes.

Component differences

JNDI names in deployment descriptors

EJB 1.1 and later deployment descriptors do not specify JNDI names for deployed EJB components. Consequently, EJB 1.1 components imported into EAServer use the default JNDI name assigned by the deployment tool. After deploying, you can reconfigure the component to change the JNDI bindings.

Environment properties

EJB 1.1 allows environment properties to be accessed using JNDI, and the EJBContext.getEnvironment method is now deprecated. Environment properties can also contain values of types other than String.

Environment properties used within a bean must be cataloged in the bean's deployment descriptor. After deployment, you can edit the property values in the component properties—see Chapter 2, "Deploying and Configuring EJB Components."

You must call the JNDI Context.lookup method to access environment properties. To locate the naming context, create a javax.naming.InitialContext object for *java:comp/env*. In this example, the application retrieves the value of the environment property *maxExemptions* and uses that value to determine an outcome:

```
Context initContext = new InitialCopntext();
Context myEnv =
   (Context)initContext.lookup("java:comp/env");

// Get the maximum number of tax exemptions
Integer max=(Integer)myEnv.lookup("maxExemptions");

// Get the minimum number of tax exemptions
Integer min = (Integer)myEnv.lookup("minExemptions");

// Use these properties to customize the business logic
if (numberOfExemptions > max.intValue() ||
```

```
(numberOfExemptions < min.intValue())
 throw new InvalidNumberOfExemptionsException();
```

.

EJB and resource references

EJB 1.1 allows components to use logical names to access database connections, JavaMail sessions, and the home interfaces of other components. These names must be catalogued in the bean's deployment descriptor. After deployment, you can configure the target of the reference in the component properties—see Chapter 2, "Deploying and Configuring EJB Components."

Security access-control changes

The getCallerIdentity and isCallerInRole(java.security.Identity) methods in the EJBContext interface are deprecated in EJB 1.1. Instead of getCallerIdentity, call getCallerPrincipal. Instead of isCallerInRole(java.security.Identity), call isCallerInRole(java.lang.String).

You can configure role references for your component. Role references allow you to map names used in isCallerInRole(java.lang.String) calls to role names that exist on the server. Role references allow your component to be deployed on servers that do not have the same security configuration. After deploying, you can configure the component to assign server role names to the role reference names used in the component.

Declarative access control for EJB 1.1 and later version components uses method-level settings. You configure access restrictions for methods with <method-permission> elements in the deployment descriptor, listing the methods that the permission constraint applies to inside the <method-permission> element. When you deploy the EJB-JAR to EAServer, EAServer creates role configurations and security settings to apply the security constraints. For details, see the description of the ejb.accessControl Ant configuration property on page 25.

❖ Configuring role references

Role references in the EJB JAR deployment descriptor allow you to map logical role names used in the isCallerInRole Java method. While the EJB specification encourages declarative security constraints, you can programmatically check role membership with this method. The role reference allows you to logical role names when calling isCallerInRole, then map them to roles that exist on the deployment server.

When you deploy an EJB-JAR that contains role references, EAServer generates commands in the module configuration file to alias the role to another role. You can edit or reconfigure the role reference as follows:

1 If necessary, define new EAServer roles to be used by callers of the component.

2 Edit the EJB module configuration file as described in "Configuring EJB component properties" on page 20.

For each role reference in the deployment descriptor, EAServer generates the following in the configuration target:

```
<addRoles toRole="LinkedRole">
RoleUsedInCode
</addRoles>
```

Where *RoleUsedInCode* is the logical role name used in the EJB code (specified by `role-name` in the deployment descriptor `security-role-ref` element) and *LinkedRole* is the role name specified by `link-role` in the deployment descriptor. This command makes *RoleUsedInCode* have the same included and excluded users as *LinkedRole*.

Transaction isolation level

In EJB 1.1 and later version EJB deployment descriptors, you cannot declare an EJB component's transaction isolation level. However, when deploying to EAServer, you can modify the component property that specifies the isolation level (by setting the sqlIsolationLevel attribute of the transaction property).

Client model differences

Except for the differences below, the EJB 1.1 client model is identical to the EJB 1.0 model:

- **Finder method return types** Finder methods in EJB 1.1 clients can return java.util.Collection or java.util.Enumeration. Finder methods in EJB 1.0 must return java.util.Enumeration. The use of java.util.Collection is recommended for new development.
- **Home interface serialization** You can call the Home.getHandle method to serialize a home interface proxy in an EJB 1.1 client.
- **EJBMetaData enhancements** The EJBMetaData interface, used by development tools to dynamically inspect EJB components, provides an isStatelessSession method that returns true if the component is a stateless session bean.

CHAPTER 2

Deploying and Configuring EJB Components

Topic	Page
Deploying an EJB-JAR file	19
Configuring EJB component properties	20
Commonly configured properties	24

Deploying an EJB-JAR file

EJB components can be deployed in an EJB-JAR file. The EJB-JAR file packages the component classes and a *deployment descriptor* in the standard Java archive (JAR) format. The deployment descriptor describes the components included in the JAR and sets standard properties such as the home and remote interface names, transaction attribute values, required user roles, and so forth.

You can use several tools to create EJB-JAR files, including Ant, Eclipse, and popular IDEs such as Borland JBuilder. You can deploy the EJB-JAR to EAServer using several tools including:

- The Management Console, as described below.
- The deploy utility, as described in Chapter 12, “Command Line Tools,” in the *System Administration Guide*.
- The jagant and jagtool utilities.

❖ **Deploying EJB-JARs with the Management Console**

1. Start the Management Console and connect to EAServer as described in Chapter 1, “Getting Started,”in the *System Administration Guide.*
2. In the Management Console, right-click the EJB Modules folder in the left pane and choose Deploy. The Deploy wizard pages appear in the right pane. Fill in the Wizard settings as listed in the table below.

Setting	Explanation
File Name	Enter the full path to the EJB-JAR file, or browse to select the file.
Module Name	Enter a name for the package to be created. The default is the base name of the EJB-JAR file.
Overwrite if Package Already Exists.	Enable to overwrite an existing package that has the same name.
Do Validation During Deployment	Enable to enforce validation of the XML deployment descriptor. In accord with the J2EE specification, this option is enabled by default. Disabling validation may allow you to deploy archives that have invalid XML but are otherwise correctly packaged.
Server Name	The name of the server to install the module to. The default is the name of the server that you are connected to.
Install Module Into Selected Server	Deselect if you don't want the package installed into any server. The package must be installed in a server before clients of that server can call the components.
Directory Name	If you specify a directory name, EAServer creates a copy of the archive file in the specified directory. The copy is identical to the source, except that an EAServer configuration file is added to the *META-INF* directory. The directory must exist and be a full path or a path relative to the EAServer *bin* directory.

Click Finish on the final wizard page. The Management Console deploys the EJB-JAR and shows the deployment status in the right pane. When your browser finishes downloading the status page, scroll to the bottom. You see Build Successful if everything went ok.

Configuring EJB component properties

When deploying an EJB-JAR, EAServer creates an Ant build script that contains targets to configure the component properties and generate the classes required to run the components in EAServer.

The configuration script contains settings read from the EJB-JAR deployment descriptor. You may need to customize settings such as those listed below to match the server configuration:

- For entity beans that use container-managed persistence, the Persistence settings.
- Role mappings and method-level permissions.
- Resource references.
- EJB references to components that are not installed with the JAR file or when multiple beans use the same home and remote interfaces. It is impossible to infer EJB references if more than one bean uses the home and remote interfaces specified by the reference properties in the deployment descriptor.
- Environment properties.
- Resource Environment Refs properties.
- Run As Identity properties.

Structure of the configuration file

For an introduction to configuration with Ant, see Chapter 2, “Ant-Based Configuration,” in the *Automated Configuration Guide*.

For EJB components, the configuration file includes the following:

- Ant property definitions for commonly configured properties, described in “Commonly configured properties” on page 24.
- A `configure-default` target, which sets the package and component properties. This target executes when you configure the component using the Management Console or the configure command-line tool.
- A `recompile-default` target, which regenerates the classes required to run the components in EAServer. This target executes when you recompile the component using the Management Console or the recompile command-line tool. This target also executes the `configure-default` target.
- A `refresh-default` target, which loads or reloads the implementation classes and generated classes into the server. This target also applies changes to properties that affect run-time behavior, such as JNDI name bindings or security constraints.

Updating component properties

You can update component properties in the Management Console or by using command-line tools.

For deployed EJB modules, the Management Console displays properties on these tabbed pages:

- **General** Displays the module name and an optional description
- **Configuration** Displays the configuration script that deployment generated based on the settings in the deployment descriptor.
- **ejb-jar.xml** Displays the contents of the deployment descriptor for reference only.
- **User Configuration** Displays the contents of the user configuration script if present. You can create a user-configuration script as described below.
- **Advanced** Allows you to modify properties in the configuration scripts graphically.

You should modify component properties from a user configuration script rather than the default configuration script that is generated by deployment. If you modify the default script, redeployment overwrites your changes.

❖ **Creating a user-configuration script with the Management Console**

1 Start the Management Console and connect to EAServer as described in Chapter 1, “Getting Started,”in the *System Administration Guide*.

2 In the Management Console, expand the EJB Modules folder in the left pane and highlight the icon for your EJB package. The right frame displays the properties for the package.

3 If the properties do not include a User Configuration tab, create the user configuration script as described in “Creating user-configuration scripts” in Chapter 2, “Ant-Based Configuration,” in the *Automated Configuration Guide*.

Note You can also embed a user-configuration script in your EJB-JAR file so that the settings are applied automatically when deploying. For details, see “Embedding configuration scripts in J2EE archives” in Chapter 2, “Ant-Based Configuration,” in the *Automated Configuration Guide*.

❖ **Modifying user-configuration properties with the Management Console**

1. Click the User Configuration tab to display the contents of the user-configuration script. Make required edits and click Apply to save your changes. For property descriptions, see "Commonly configured properties" on page 24.
2. Follow the steps in "Configuring, recompiling, and refreshing components with the Management Console" on page 23 to reload the components so the property changes take affect.

❖ **Using the Advanced tab in the Management Console**

The Advanced tab allows you to modify properties in the configuration scripts graphically.

1. Click the Advanced tab to display the graphical controls. Controls are provided for commonly configured properties such as name bindings. For property descriptions, see "Commonly configured properties" on page 24.
2. If you change settings, click Apply to save the changes to the EAServer repository, then click Synchronize to apply the same changes to the configuration script. If you do not synchronize changes to the configuration script, your property changes can be undone by running the configure or recompile targets in the configuration scripts.
3. Follow the steps in "Configuring, recompiling, and refreshing components with the Management Console" on page 23 to reload the components so the property changes take affect.

❖ **Configuring, recompiling, and refreshing components with the Management Console**

After modifying component properties, follow the steps below to update the generated code and reload the component implementation. You must do this before property changes take affect:

1. In the left frame, right-click the icon for your EJB package and choose one of the following:
   - Configure, to apply the XML configuration file to the module and component properties.
   - Recompile, to apply the XML configuration file to the module and component properties and recreate generated classes.
2. In the left frame, right-click the icon for your EJB package and choose Refresh. EAServer reloads the implementation classes and generated classes.

❖ **Updating component properties with the command line**

1 Use a text or XML editor to edit the contents of the user configuration file for your package, *ejbjar-module-user.xml* in the EAServer *config* directory, where *module* is the name of the package. For property descriptions, see "Commonly configured properties" on page 24.

2 Run the recompile utility to apply the changes to the component and recreate generated classes. Alternatively, run the configure utility to update the component properties without regenerating affected code. See Chapter 12, "Command Line Tools," in the *System Administration Guide* for details on these utilities.

3 Run the refresh utility to reload the implementation classes and generated classes.

Commonly configured properties

The default EJB configuration script defines top-level Ant properties for settings that are most commonly configured. For example, since all beans in a JAR file typically connect to the same database, a `sql.dataSource` property is defined as follows:

```
<property name="sql.dataSource" value="default"/>
```

This property specifies the default for the data source name that is bound to any resource reference names used in the package. It also specifies the default data source used for field storage in CMP entity beans.

To override the Ant property values in the default configuration script, define the same property in your user-configuration script, above any target definitions.

In some cases, modifying the Ant property may not suffice because more than one value is required in the component configuration. For example, two enterprise beans in the module may connect to different data sources bound to JNDI name. In this case, you must override the default <bind> command in the module configuration by running a different <bind> command in the user-configuration script.

For an introduction to configuration with Ant, see Chapter 2, “Ant-Based Configuration,” in the *Automated Configuration Guide*. For details on the syntax of Ant commands, see the following file in your EAServer installation:

```
html\help\en\index.html
```

ejb.accessControl

Specifies the type of security access control for components in the module. The Ant property value sets the default for <accessControl> subcommands in the <setProperties> commands for the EJB components and the EJB module. Allowable values are:

- `default`, to specify use of EAServer’s default role- and port-based access control mechanism.
- `jacc`, to specify use of the JACC (Java™ Authorization Contract for Containers) mechanism, with policies enforced the Java system implementation or by a custom JACC policy provider implementation.
- `none`, to specify that no access control restrictions. Any user can call the component methods.

Using the default access control type

When using the `default` access control type, EAServer enforces port-based access control based on the EAServer roles that the client user belongs to and the port number that the client has connected with.

To apply role-based constraints, the default configuration runs <permitAccess> commands to configure component and method role constraints based on the method permissions defined in the deployment descriptor. The default configuration also creates these roles for each role name defined in the EJB deployment descriptor:

- ***`role`***, where *role* is the role name used in the deployment descriptor.
- ***`ejb-role-prefix.role`***, where *ejb-role-prefix* is the value of the ejb.rolePrefix Ant property.

For evaluating a method role constraints, a user is considered a member of a role if they are a member of either the role with a matching name, or the role *ejb-role-prefix.role* where *ejb-role-prefix* is the value of the ejb.rolePrefix Ant property for the EJB module configuration. In other words, to allow access to a user, you can add them to either role.

Port-based access cannot be specified in the EJB deployment descriptor. By default, EAServer allows access through any port. In your user configuration, you can set the ejb.allowedPorts Ant property to restrict access to clients that connect through the specified port numbers. Changing the Ant property affects access to all methods unless you override the default security configuration commands.

In your user-configuration, you can override the default <accessControl> and <permitAccess> commands to fine tune the security settings. For example, you can enable auditing of permitted or denied access, or disable access to remote interface methods to allow only local-interface invocations. See the reference for the <permitAccess> and <denyAccess> commands for more information.

For example, to enable auditing:

```
<target name="configure-user">
   <setProperties package="ejb.components.example">
      <accessControl
           type="default"
           auditDeny="true"
           auditPermit="true"
     />
  </setProperties>
</target>
```

To enable auditing, auditing must also be enabled for the security domain (the security domain auditAccessDenied and auditAccessPermitted properties must both be true).

Using the JACC access control type

The JACC (Java™ Authorization Contract for Containers) mechanism authorizes method and component access using the Java system implementation or by a custom JACC policy provider implementation.

The JACC mechanism must be configured by deploying the EJB-JAR with the `-jacc` command-line option. Deployment with this option creates an additional configuration script to configure the JACC settings for the module. You must also configure the JACC settings for the EAServer security domain. For details, see “JACC (JSR-115) support” in Chapter 10, “Security Configuration Tasks,” in the *Security Administration and Programming Guide*.

ejb.allowedPorts

Specifies port numbers to restrict client access when using the default security access control mechanism. Allowable values are `all`, to allow access through any port, or a comma-separated list of port numbers to restrict access to just those ports. See ejb.accessControl for more information.

ejb.rolePrefix

Specifies a prefix for EAServer security role names created to apply role-based access constraints from the EJB deployment descriptor. See ejb.accessControl for more information.

ejb.automaticFailover

Specifies whether proxies for components in the module can transparently fail over when deployed in a cluster. The default configuration applies this Ant property value to the <automaticFailover> subcommand in the <setProperties> command for each component in the module. To override the setting for an individual component, add a <setProperties> command to your user configuration that include the <automaticFailover> subcommand. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.myjar.MyCompRemote">
     <automaticFailover enable="true"/>
  </setProperties>
</target>
```

For more information on failover, see Chapter 6, “Clusters and Synchronization,” in the *System Administration Guide*.

ejb.clusterPartition

If deploying to a cluster, specifies a global cluster partition name applied to the the properties of all components in the module. The component runs only on servers that are in the partition. The default configuration applies this Ant property value to the <clusterPartition> subcommand in the <setProperties> command for each component in the module. To override the setting for an individual component, add <setProperties> command to your user configuration to configure the component's that correspond to the enterprise bean's public interfaces. Include the <clusterPartition> subcommand. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.myjar.MyCompRemote">
     <clustePartition name="salesPartition"/>
  </setProperties>
  <setProperties component="ejb.components.myjar.MyCompHome">
     <clustePartition name="salesPartition"/>
  </setProperties>
</target>
```

For more information on clusters and partitioning, see Chapter 6, "Clusters and Synchronization," in the *System Administration Guide*.

ejb.copyValues

Specifies a global value to enable or disable by-reference parameter passing for in-server remote interface invocations of the module's components. A value of true means parameters are passed by creating temporary copies, making in-server remote interface calls behave like calls to a remote server. A value of `false` enables by-reference parameter passing, effectively making in-server remote interface invocations behave like local interface invocations. By-reference parameter passing can improve server performance, but may change the behavior of component code that modifies or retains references to variables passed as parameters to remote interface invocations.

To override this setting for individual components, create a <setProperties> command in your user configuration that runs the <copyValues> subcommand. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.myjar.MyCompRemote">
     <copyValues enable="true"/>
  </setProperties>
</target>
```

ejb.logExceptions

Specifies a global value to enable or disable logging of exceptions thrown by components in the module. If set to `true`, EAServer logs application and system exceptions that are thrown by the business methods for any component in the EJB module.

To override this setting for individual components, create a <setProperties> command in your user configuration that runs the <logExceptions> subcommand. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.myjar.MyCompRemote">
     <logExceptions enable="true"/>
  </setProperties>
</target>
```

You can disable logging of exceptions for all components in the server by setting the server Log System Exceptions and Log Application Exception properties. If exception logging is disabled in the server properties, the component settings have no affect.

ejb.enableProfiling

Specifies a global value to enable or disable profiling of home, remote, and local interface method invocations. A value of true causes generation of additional method profiling code. However, to collect and view profiling statistics, the server's Disable Statistics property (disableStatistics) must be set to false.

To override this setting for individual components, create a <setProperties> command in your user configuration that runs the <profilePublicMethods> subcommand. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.myjar.MyCompRemote">
     <profilePublicMethods enable="true"/>
  </setProperties>
</target>
```

Profiling data is included in the server statistics output. There are several ways to view this data, including spreadsheet software, the Management Console, and as text files output periodically by the server. For more information, see "Viewing server statistics" in Chapter 11, "Runtime Monitoring," in the *System Administration Guide*.

If you change this setting for the EJB module, recompile the module and restart the server for the change to take affect. If you change the server Disable Statistics setting, restart the server for the change to take affect.

ejb.enableTracing

Specifies a global value to enable or disable tracing of home, remote, and local interface method invocations. A value of true causes generation of additional tracing code. EAServer writes trace messages to the server log if tracing is enabled for the server. To enable EJB method tracing for the server, specify the -ejbTrace option when starting the server with the start-server command, or set the Enable EJB Tracing (ejbTrace) server property.

To override this setting for individual components, create a <setProperties> command in your user configuration that runs the <tracePublicMethods> subcommand. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.myjar.MyCompRemote">
     <tracePublicMethods enable="true"/>
  </setProperties>
</target>
```

ejb.isolationLevel

Specifies a global logical transaction isolation level for EJB-CMP entity beans in the module. This setting specifies the effective transaction isolation level for transactions managed using EAServer's query caching and optimistic concurrency control (OCC) mechanisms. This setting allows the performance benefits of OCC and caching, while also enforcing an effective transaction isolation level as you would use with pessimistic concurrency control. For more information on configuring concurrency control mechanisms, see Chapter 4, "EJB CMP Tuning," in the *Performance and Tuning Guide*.

The sql.isolationLevel Ant property specifies the database transaction isolation level used on the connection to the remote database.

To override this setting for individual components, create a <setProperties> command in your user configuration that runs the <persistentObject> subcommand. Specify the isolation level in the isolationLevel attribute. You can supply additional attributes to further tune the settings, for example the cacheTimeout attribute specifies a timeout for cached data.

Table 2-1 lists the allowable isolation levels.

Table 2-1: Logical isolation level values

Setting	Effect
`ReadCache`	Queries are satisfied by reading from the local query cache if possible. Otherwise, data is loaded from the remote database. Not recommended, as use of this isolation level can result in "lost" updates: updates made from the entity bean may go through even if the data has been modified since first read from the database. If lost updates are unacceptable, use `ReadCacheVerifyUpdates` and verify that the concurrency control configuration allows for locking or OCC version control.
`ReadCacheVerifyUpdates`	Queries are satisfied by reading from the object cache if possible. Otherwise, data is loaded from the remote database. If the entity is changed or removed, the corresponding SQL update or delete verifies that the data was not changed after it was loaded from the DBMS. This setting is suitable when it is acceptable for a read-only transaction to use stale cache data. To limit the use of stale data, specify a cache timeout by setting the cacheTimeout attribute for the <persistentObject> command, or by setting the cacheTimeout attribute for <queryMethod> commands.
`ReadCommitted`	Queries are satisfied by reading from the remote database. If the entity is changed or removed, the corresponding SQL update or delete does not verify that the data was not changed after it was loaded from the DBMS. Not recommended, as use of this isolation level can result in "lost" updates. If lost updates are unacceptable, use `ReadCommittedVerifyUpdates` and verify that the concurrency control configuration allows for locking or OCC version control.
`ReadCommittedVerifyUpdates`	Queries are satisfied by reading from the remote database. If the entity is changed or removed, the corresponding SQL update or delete verifies that the data was not changed after it was loaded from the DBMS. This setting provides a good balance of data integrity and performance. However, for some application data models, the maintenance of full data integrity requires a higher isolation level such as `RepeatableRead`. `ReadCommittedVerifyUpdatesWithCache` may provide better performance.

Setting	Effect
`ReadCommittedWithCache`	Queries are satisfied by reading from the object cache if possible. Otherwise, data is loaded from the remote database. If the entity is changed or removed, the corresponding SQL update or delete does *not* verify that the data was not changed after it was loaded from the DBMS. Otherwise, for read-only access, and only if data was loaded from the local cache, a commit-time verify step ensures that the data has not changed since it was originally loaded from the DBMS. This ensures that any cached data that was used is still current at commit time, but does not prevent concurrent or conflicting updates. This setting is not recommended, as it can result in lost updates. Instead, use `ReadCommittedVerifyUpdatesWithCache`.
`ReadCommittedVerifyUpdates WithCache`	Queries are satisfied by reading from the local cache if possible. Otherwise data is loaded from the remote database. If the entity is changed or removed, the corresponding SQL update/delete verifies that the data was not changed after it was loaded from the DBMS. Otherwise, for read-only access, and only if data was loaded from the local cache, a commit-time verify step ensures that the data has not changed since it was originally loaded from the DBMS. This ensures that any cached data that was used is still current at commit time. This setting does not prevent concurrent updates but does prevent conflicting updates. This setting is suitable when it is not acceptable for a read-only transaction to use stale data, and where commit-time verification is cheaper than satisfying queries from the DBMS; in particular, where a table timestamp is specified with the <persistentObject> tableVersion attribute, or where a JDBC/JIT driver wrapper is used (the JIT driver wrappers can batch verification statements together at commit-time).

Setting	Effect
`RepeatableRead`	Queries are satisfied by reading from the remote database. If the entity is changed or removed, the corresponding SQL update or delete will verify that the data was not changed after it was loaded from the DBMS. Otherwise, for read-only access, a commit-time verification ensures that the data has not changed since it was loaded from the DBMS. If pessimistic locking is enabled with the <persistentObject> selectWithSharedLock or selectWithUpdateLock attribute, and is supported by the DBMS, verification is skipped as the shared/exclusive locks that are obtained at load time will prevent conflicting updates. **Warning!** Pessimistic locking may increase the occurrence of deadlock. This setting is suitable for cases where uncontrolled concurrent updates may result in data integrity problems (even for read-only access). `RepeatableReadWithCache` may provide better performance, although if many transactions are updating the same rows, pessimistic locking with no cache is probably preferable.
`RepeatableReadWithCache`	Queries are satisfied by reading from the local cache if possible. Otherwise, data is loaded from the remote database. If the entity is changed or removed, the corresponding SQL update or delete verifies that the data was not changed after it was loaded from the DBMS. Otherwise, for read-only access, a commit-time verification ensures that the data has not changed since it was originally loaded from the DBMS. This setting is suitable for cases where uncontrolled concurrent updates may result in data integrity problems (even for read-only access), where it is not acceptable for a read-only transaction to use stale cache data, and where commit-time verification is cheaper than satisfying queries from the DBMS; in particular, where a table timestamp is specified with the <persistentObject> tableVersion attribute, or where a JDBC/JIT driver wrapper is used (the JIT driver wrappers can batch verification statements together at commit-time). If many transactions from other sources are updating the same rows, you may get better performance using `RepeatableRead` with pessimistic locking.
`Serializable`	Like `RepeatableRead`, except guarantees that the set of rows satisfying the query cannot change before the current transaction completes. **Note** This isolation level is not supported when using Oracle databases. Use `RepeatableRead` instead, or add additional application-level data version checking code.

Setting	Effect
SerializableWithCache	Like RepeatableReadWithCache, except guarantees that the set of rows satisfying the query cannot change before the current transaction completes. **Note** This isolation level is not supported when using Oracle databases. Use RepeatableReadCache instead, or add additional application-level data version checking code.

ejb.localNamePrefix

The ejb.localNamePrefix and ejb.localNameSuffix Ant properties define the suffix and prefix, respectively, for JNDI names bound to local interfaces in the module. Each local interface is bound to a JNDI name comprised of the prefix, the suffix, and the name of the enterprise bean.

To add additional name bindings, add <bind> commands to your user configuration script. For example:

```
<target name="configure-user">
  <setProperties package="ejb.components.myjar">
     <bind name="myapp/MyComp"
           component="ejb.components.example.MyCompRemoteHome"/>
     <bind name="myapp/MyOtherComp"
           interface="com.example.bank.MyOtherHome"/>
     <bind name="java:comp/env/ejb/MyComp"
           component="ejb.components.example.MyCompRemoteHome"/>
  </setProperties>
</target>
```

ejb.localNameSuffix

See ejb.localNamePrefix.

ejb.localThreadMonitor

Specifies the name of a thread monitor to govern local interface invocations. For details on thread monitors, see “Monitoring threads” in Chapter 3, “Creating and Configuring Servers,” in the *System Administration Guide*.

To override this setting for individual components, create a <setProperties> command in your user configuration that runs the <threadMonitor> subcommand. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.myjar.MyCompLocal">
     <threadMonitor name="myMonitor"/>
  </setProperties>
</target>
```

ejb.remoteNamePrefix

The ejb.remoteNamePrefix and ejb.remoteNameSuffix Ant properties define the suffix and prefix, respectively, for JNDI names bound to remote interfaces in the module. Each remote interface is bound to a JNDI name comprised of the prefix, the suffix, and the name of the enterprise bean.

To add additional name bindings, add <bind> commands to your user configuration script. See ejb.localNamePrefix for an example.

ejb.remoteNameSuffix

See ejb.remoteNamePrefix

ejb.remoteThreadMonitor

Specifies the name of a thread monitor to govern remote interface invocations. For details on thread monitors, see “Monitoring threads” in Chapter 3, “Creating and Configuring Servers,” in the *System Administration Guide*.

To override this setting for individual components, create a <setProperties> command in your user configuration that runs the <threadMonitor> subcommand. See ejb.localThreadMonitor for an example.

ejb.serviceThreadMonitor

Not used in the default configuration. If you create a user configuration to run an enterprise bean as an EAServer service component, you can reference this property to specify the thread monitor name. For details on service components, see Chapter 4, "Creating Service Components," in the *Automated Configuration Guide*.

ejb.transactionBatch

Specifies the name of the transaction batch set in the transaction properties of components in the module. A value of "none" or "default" indicates no transaction batching will be performed. Transaction batching allows container-managed transactions from independent threads to be grouped into a single transaction.This can help to increase throughput, especially when using container-managed persistence and EAServer's query-caching features with a JIT data source driver wrapper. Transaction batches must be defined with a separate <setProperties> command, for example:

```
<target name="configure-user">
  <setProperties transactionBatch="MyTransactionBatch">
    <property name="maximumBatchSize" value="5"/>
    <property name="maximumBatchWait" value="20"/>
  </setProperties>
</target>
```

To override the global transaction batch setting for individual components, create a <setProperties> command in your user configuration that runs the <transaction> subcommand and sets the batch attribute in addition to other attributes copied from the component's <setProperties> command in the default configuration. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.example.MyCompRemote">
    <transaction type="Required" batch="myTxBatch"/>
  </setProperties>
</target>
```

Note The <transaction> type attribute specifies the transaction attribute from the EJB deployment descriptor. Copy the value from the default configuration so as not to change the transactional behavior of the application.

To override the transaction settings for individual methods, include the method signature as the method attribute value. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.example.MyCompRemote">
    <transaction method="buy(int, java.lang.String)" type="Required"
batch="myTxBatch"/>
    <transaction method="sell(int, java.lang.String)" type="Required"
batch="myTxBatch"/>
  </setProperties>
</target>
```

ejb.transactionRetry

Globally enables or disables automatic retry of container-managed transactions for components in the module. If automatic retry is enabled, the transaction manager may automatically retry transactions that are rolled back for any reason, including use of the setRollbackOnly API.

To override this setting for individual components or methods, create a <setProperties> command in your user configuration that runs the <transaction> subcommand and sets the retry, retryCount, and retryDelay attributes. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.example.MyCompRemote">
    <!-- enable retry for sell transactions, with a retry count of 2
         and a retry delay of 2 seconds (2000 milliseconds). -->
    <transaction method="sell(int, java.lang.String)"
                 type="Required"
                 retry="true"
                 retryCount="2"
                 retryDelay="2000"/>
    <!-- Disable transaction retry for other methods. -->
    <transaction type="Required"
                 retry="false" />
  </setProperties>
</target>
```

Note The <transaction> type attribute specifies the transaction attribute from the EJB deployment descriptor. Copy the value from the default configuration so as not to change the transactional behavior of the application.

ejb.passivateTimeout

Globally configures the passivation timeout for EJB stateful session beans in the module. The value is the allowed idle time in seconds. If an instance is idle for more than this amount of time, EAServer removes the instance from memory by serializing it to persistent storage. If the client calls a method on a passivated instance, EAServer deserializes the instance back into memory to service the invocation. You can also set the ejb.removeTimeout property to specify a timeout for complete removal of idle instances.

To override this setting for individual components, create a <setProperties> command in your user configuration that runs the <ejbPassivate> subcommand and sets the timeout attribute. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.myjar.MyCompRemote">
     <ejbPassivate timeout="100"/>
  </setProperties>
</target>
```

EAServer passivates instances by storing in the ejb_session database table using the `session.db` data source. In the default, as-installed configuration, the `session.db` data source is an alias for the `default` data source, which connects to the preconfigured default database server.

ejb.removeTimeout

Globally configures the removal timeout for EJB stateful session beans in the module. The value is the allowed idle time in seconds. If an instance is idle for more than this amount of time, EAServer removes the instance. Client invocations fail after the instance has been removed.

To override this setting for individual components, create a <setProperties> command in your user configuration that runs the <ejbRemove> subcommand and sets the timeout attribute. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.myjar.MyCompRemote">
     <ejbRemove timeout="1200"/>
  </setProperties>
</target>
```

ejb.poolTimeout

Globally configures an instance pool timeout value for the entity beans and stateless session beans in the module (stateful session beans are not pooled). If a pooled instance remains idle for more than this amount of time, EAServer discards the instance.

To override this setting for individual components, create a <setProperties> command in your user configuration that runs the <instancePool> subcommand and sets the timeout attribute. For example:

```
<target name="configure-user">
    <setProperties component="ejb.components.example.MyCompRemote">
      <instancePool timeout="100"/>
  </setProperties>
</target>
```

jca.connectionFactory

Specifies the default JCA administered object bound to any JCA resource references names used in the module. To override the JCA administered object bound to a JNDI name, create a <setProperties> command in your user configuration that contains a <bind> command specifying the JNDI name and the JCA administered object to bind it to. For example:

```
<target name="configure-user">
  <setProperties package="ejb.components.myjarname">
    <bind name="java:comp/env/jdbc/MyConnector"
          administeredObject="myConnector"/>
  </setProperties>
</target>
```

sql.createTables

Specifies whether database tables for CMP entity beans should be created by EAServer when the module is created. Automatic table creation can be useful for debugging during development of new CMP entity beans.

Automatic table creation is for testing only
For deployment to production servers, you or your DBA should create the tables, using an optimized index model and any other necessary optimizations, such as enabling row-level locking.

If you use automatic table creation, you can override the <loadComponents> command in the default configuration and add the dropTables attribute. This attribute causes old tables to be dropped when redeploying. For example:

```
<target name="configure-user">
  <loadComponents package="ejb.components.myjarname"
                  sqlCompile="true"
                  createTables="true"
                  dropTables="true"/>
</target>
```

If you do not specify dropTables as true, tables are not created unless the tables don't already exist.

When running against predefined tables, review the <persistentObject> and <persistentField> commands in the default configuration. If a table names do not match the EJB entity bean name, you must override the <persistentObject> setting in your user configuration. If column names do not match the corresponding persistent field name, you must override the <persistentField> setting in your user configuration.

sql.dataSource

Specifies the default for the data source name that is bound to any JDBC resource reference names used in the module. It also specifies the default data source used for field storage in CMP entity beans.

To override the data source bound to a JNDI name, create a <setProperties> command in your user configuration that contains a <bind> command specifying the JNDI name and the data source to bind it to. For example:

```
<target name="configure-user">
  <setProperties package="ejb.components.example">
    <bind name="java:comp/env/jdbc/MyDataSource" dataSource="myDB"/>
  </setProperties>
</target>
```

To override the data source for CMP entity bean field storage, create a <setProperties> command in your user configuration to configure the CMP entity bean. Copy and paste the <persistentObject> from the default configuration's <setProperties> command for this component, then modify the dataSource attribute. For example:

```
<target name="configure-user">
  <setProperties component="ejb.components.example.CustomerInventory">
    <persistentObject
```

```
        table="cust_inv"
        isolationLevel="RepeatableRead"
        dataSource="myDB"
        />
  </setProperties>
</target>
```

sql.isolationLevel

Specifies the global JDBC connection SQL isolation level used for container-managed transactions (by calling method setTransactionIsolation in class java.sql.Connection). The precise interpretation of this property, particularly `Serializable`, is database-specific. Allowable values are:

- `ReadUncommitted`
- `ReadCommitted`
- `RepeatableRead`
- `Serializable`

If you are using container-managed persistence, Sybase recommends you set the SQL isolation level to `ReadCommitted` and configure the CMP logical isolation level for <persistentObject> and <queryMethod> commands to best match the characteristics of the application. This allows the persistence manager more options for database concurrency management, and thus may help you to obtain better performance. For details on the CMP logical isolation level, see ejb.isolationLevel.

CHAPTER 3

Developing EJB Clients

Topic	Page
Client runtime requirements	43
EJB client program flow	44
Instantiating home interface proxies	45
Instantiating remote or local interface proxies	52
Calling remote interface methods	54
Calling local interface methods	54
Managing transactions	55
Serializing and deserializing bean proxies	56
Using EJB providers	57
Running EAServer 5.x clients against 6.0 or later servers	59

Client runtime requirements

To run EJB clients, you must have:

- A Java Runtime Environment (JRE) or JDK installation, version 1.4.2 or later. Compatible versions of JDK 1.4 and JDK 1.5 are included in the EAServer client runtime installation.
- An EAServer client runtime installation. The Java class path must contain the EAServer JAR file *lib/eas-client-14.jar* or *lib/eas-client-15.jar* to run in JRE 1.4 or JRE 1.5, respectively. If you package these classes in a different archive or location, you must set the djc.home Java system property to indicate the location of the EAServer installation. For example:

  ```
  java -Ddjc.home=%DJC_HOME% com.foo.MyClass
  ```

  If you do not set this property, the client runtime infers the default from the first class path entry that ends with *lib/eas-client-14.jar* or *lib/eas-client-15.jar*.
- Stubs for the components that your client invokes.

EAServer generates client stubs on demand. If stubs are not available, the client runtime generates and compiles stub classes when you first invoke a component.

You must have a full JDK installation to generate compiled stubs at runtime. To deploy client applications to run in a JRE installation, you must include the generated stubs. When you deploy an EJB-JAR file, EAServer automatically creates stubs under the *genfiles/java/classes* subdirectory of the EAServer installation. Stub classes are in the Java package named by appending iiop_stubs to the package name of the interface class. For example, for interface com.sybase.easerver.tutorials.ejb.Query, the Java stub package is com.sybase.easerver.tutorials.ejb.iiop_stubs.

You can also manually generate stubs with the stub-compiler command. For details, see Chapter 12, "Command Line Tools," in the *System Administration Guide*.

EJB client program flow

The steps in the table below describe the program flow in a typical EJB client:

Step	Action	For more information
1	Add code to create the initial naming context and instantiate the home interface proxies.	See "Instantiating home interface proxies" on page 45.
2	Add code to instantiate remote or local interface proxies.	See "Instantiating remote or local interface proxies" on page 52.
3	Add code to call remote or local interface methods.	See "Calling remote interface methods" on page 54 or "Calling local interface methods" on page 54.
4	Optionally add code to control transactions and serialize and deserialize instances.	See: • "Managing transactions" on page 55 • "Serializing and deserializing bean proxies" on page 56

Instantiating home interface proxies

EJB clients use the Java Naming and Directory Interface (JNDI) to resolve logical bean JNDI names to proxy instances for a bean's home interface. Each EJB container vendor provides an implementation of this interface that works with the vendor's server and network protocol.

Obtaining an initial naming context

The core JNDI interface used by client applications is javax.naming.Context, which represents the initial naming context used to resolve names to bean proxies. To obtain an initial naming context, initialize a java.util.Properties instance and set the properties listed in Table 3-1. Pass the properties instance to the javax.naming.InitialContext constructor. The code fragment below shows a typical call sequence:

```
import javax.naming.*;

static public Context getInitialContext() throws Exception {
    java.util.Properties p = new java.util.Properties();

    // Sybase implementation of InitialContextFactory
    p.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sybase.ejb.InitialContextFactory");

    // URL for the Server's IIOP port
    p.put(Context.PROVIDER_URL, "iiop://myhost:2000");

    // Username "pooh", password is "tigger2"
    p.put(Context.SECURITY_PRINCIPAL, "pooh");
    p.put(Context.SECURITY_CREDENTIALS, "tigger2");

    // Now create an InitialContext that uses the properties
    return new InitialContext(p);
}
```

EJB servers from different vendors require different InitialContext property settings. If you are creating a client application that must be portable to other EJB servers, use an external mechanism to specify properties rather than hard-coding values in the source code. For example, in a Java application use command-line arguments or a serialized Java properties file. To specify properties used by a Java applet, use parameters in the HTML Applet tag that loads the applet.

Sybase InitialContext properties

The Sybase InitialContext implementation recognizes the properties in the following table. You can create multiple contexts with different properties. For example, you might create one context for proxies that connect with plain IIOP and another for proxies that connect using SSL. When using the EAServer context factory, the com.sybase.ejb prefix is optional for context properties.

Table 3-1: Sybase EJB InitialContext Properties

Property name	Description
java.naming.factory.initial	Specifies the fully qualified Java class name of the class that returns javax.naming.InitialContext instances that interact with the naming provider. Use `com.sybase.ejb.InitialContextFactory` for EAServer EJB clients.
java.naming.provider.url	Specifies the URL to connect to the EAServer name server, in the format described in "Connection URL formats" on page 48.
java.naming.security.principal	Specifies the user name for the EAServer session. Required if user name/password authentication is enabled for your server.
java.naming.security.credentials	Specifies the password for the EAServer session. Required if user name/password authentication is enabled for your server.
com.sybase.ejb.socketTimeout	The timeout for the network connection, specified in seconds. The default is 600 (10 minutes). The timeout determines how long underlying network API calls to connect to and read data from the server may block before the client runtime throws an exception or fails over to another server in a cluster.
com.sybase.ejb.idleConnectionTimeout	Specifies the time, in seconds, that a connection is allowed to sit idle. When the timeout expires, the ORB closes the connection. The default is 0, which specifies that connections can never timeout. The connection timeout does not affect the life of proxy instance references; the client runtime may close and reopen connections transparently between proxy method calls. Specifying a finite timeout for your client applications can improve server performance. If many instances of the client run simultaneously, a finite client connection timeout limits the number of server connections that are devoted to idle clients. A finite timeout also allows rebalancing of server load in an application that uses a cluster of servers.
com.sybase.ejb.lookupCacheTimeout	Specifies the time, in seconds, that previously resolved names may be cached in the client runtime. The default is 600 (10 minutes). Caching can improve performance by avoiding network round trips to resolve previously resolved names.
com.sybase.ejb.dataCompression	Whether to compress data when reading and writing to the server connection. The default is false, which disables compression. When compression is enabled, EAServer compresses data using the GZIP format (provided by the Java package java.util.zip).
com.sybase.ejb.disableResolveHost	When set to true, disables resolution of host names in the naming context URL (java.naming.provider.url). The default is false. A setting of true can be useful when running the server in the same process as the client or running the server and client on the same machine using `localhost` as the server's listener host name as the host name in the naming context URL. In these scenarios, host name resolution may cause network errors if the host machine lacks an active network connection.

Property name	Description
com.sybase.ejb. automaticFailover	When running in a cluster, a value of true allows component proxies to automatically fail over to run component instances on another server regardless of whether failover is configured in the component properties. The default of false does not allow failover unless the component properties are configured to support failover. You can configure automatic failover with the default setting by enabling failover in the properties of the components that can support it. If the component properties support failover, EAServer adds automatic failover code to the component stubs.
com.sybase.ejb. HttpTunnelling	Forces use of HTTP tunnelling. A value of true causes the client runtime to use HTTP tunnelling without trying to use plain IIOP first. The default is false. With the default setting, the proxy tries to open a connection using plain IIOP, and switches to HTTP tunnelling if the plain IIOP connection is refused. The default is appropriate when some users connect through firewalls that require tunnelling and others do not; the same application can serve both types. If you know tunnelling is required, set this property to true. This setting eliminates a slight bit of overhead that is incurred by trying plain IIOP connections before tunnelling is used.
com.sybase.ejb. HttpExtraHeader *Applicable only to Java application clients.*	An optional setting to specify what extra information is appended to the header of each HTTP packet when connecting through a Web proxy. There is no need to set this property unless your HTTP proxy server has special protocol requirements. By default, the following line is appended to each packet::: `User-agent: Jaguar/major.minor` where *major* and *minor* are the major and minor version numbers of your EAServer client software, respectively. You can set this property to specify text to be included at the end of each HTTP header. If multiple lines are included in the setting, they must be separated by carriage return and line feed characters. If the setting does not include a `User-agent:` line, then the default setting above is included in the HTTP header. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.ejb. HttpTunnellingSecure	If set to true when using a a reverse proxy server, forces use of SSL for the connection to the reverse proxy. Set this property to true if the connection to the reverse proxy must use SSL (HTTPS) tunnelling, but the connection from the proxy to the server does not use SSL tunnelling. For more information, see Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide*.
com.sybase.ejb. UseJavaURLConnection	When using HTTP tunnelling, a setting of true causes the client runtime to use java.net.URLConnection rather than java.net.HTTPConnection to connect to the server. The default of false causes the client to connect using java.net.HTTPConnection.
com.sybase.ejb. ReverseProxyHost	Specifies the machine name or the IP address of a reverse proxy server. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.

Property name	Description
com.sybase.ejb. ReverseProxyPort	Specifies the port number of a reverse proxy server. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.ejb. WebProxyHost *Applicable only to Java application clients.*	Specifies the host name or IP address of a Web proxy server. Applies to Java applications only. Java applets running in a Web browser will use the proxy address specified by the browser's proxy configuration. In Java applications, there is no default for this property, and you must specify both the host name and port number properties. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.ejb. WebProxyPort *Applicable only to Java application clients.*	Specifies the port number at which the Web proxy server accepts connections. Applies to Java applications only. Java applets running in a Web browser will use the proxy address specified by the browser's proxy configuration. In Java applications, there is no default for this property, and you must specify both the host name and port properties. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.ejb. HttpExtraHeader *Applicable only to Java application clients.*	An optional setting to specify what extra information is appended to the header of each HTTP packet when connecting through a Web proxy. See Chapter 9, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.ejb. SSLCallback *Applicable only to Java application clients.*	Required if you are using SSL and you wish to provide a callback class to set required SSL settings on an as-needed basis. Specify the name of a Java class that implements the CtsSecurity.SSLCallbackIntf interface. For example: `com.acme.AcmeSSLCallback` Chapter 4, "Using SSL in Java Clients," in the *EAServer Security Administration and Programming Guide* describes how to code a callback class.
com.sybase.ejb. userData *Applicable only to Java application clients.*	Specifies user data (String datatype). This is an optional property. Client code can set user data during NamingContext initialization and access it using SSLSessionInfo::getProperty method in the SSL callback implementation. This may be useful as a mechanism to store context information that is otherwise not available through the SSLSessionInfo interface.
com.sybase.ejb. useJSSE	Use the Java Secure Sockets Extension (JSSE) classes for secure HTTP tunnelled (HTTPS protocol) connections. JSSE provides an alternative to the built-in SSL implementations when secure connections are needed from an applet running in a Web browser. Additional configuration may be required to use this option. See Chapter 4, "Using SSL in Java Clients," in the *EAServer Security Administration and Programming Guide* for more information.

Connection URL formats

The java.naming.provider. url initial context property specifies the URL to connect to the EAServer name server. There are several variations on the URL format:

- URLs to connect to a running server
- URLs to start the server
- URLs to specify an EJB Provider name
- URLs that use the corbaname format for EJB interoperability
- URLs that specify connection properties

URLs to connect to a running server

This is the most common format of URL for distributed applications, allowing you to specify the address to connect to a remote (or local) naming server. The format is:

`iiop://`***`hostname`***`:`***`iiop-port`***`/`***`initial-context`***

where:

- *hostname* is the host machine name for the server that serves as the name server for your application. If omitted, the default is `localhost`.
- *iiop-port* is the IIOP port number for the server.
- *initial-context* is the initial naming context. This can be used to set a default prefix for name resolution. For example, if you specify *USA/Sybase/*, all names that you resolve with the context are assumed to be relative to this location in the name hierarchy. When specifying the initial context, the trailing slash is optional; it is added automatically if you do not specify an initial context that ends with a slash.

If you do not set this property, the default is `iiop://`***`hostname`***`:2000/`, where *hostname* is the name of the machine that you are running the client on.

URLs to start the server

This URL format allows you to start the server when your client program needs to connect to it. You can specify that the server runs in process with the client or in a separate process on the same machine. This feature is useful when you have applications that users must be able to run when disconnected from the network. You must deploy the client application with an export configuration that includes the files required to run the server and all server-side components hosted in the server. For more information, see Chapter 7, “Exporting Server Modules,” in the *System Administration Guide*.

To start the server in-process, use the URL format:

```
run-server:server
```

Where *server* is the name of the server to run. To start the server in a separate process, use the URL format:

```
start-server:server
```

URLs to specify an EJB Provider name

You can create EJB providers using the Management Console or an XML configuration file. The EJB provider contains a client property configuration that can be referenced by name in connection URLs. The format to reference the provider is:

```
ejb-provider:provider
```

Where *provider* is the EJB provider name. See “Using EJB providers” on page 57 for more information.

URLs that use the corbaname format for EJB interoperability

The EJB client runtime supports corbaname URLs as described in “Interoperable naming URLs” on page 69.

URLs that specify connection properties

In any URL format, EAServer properties can also be set by appending them to the URL passed to the java.naming.provider.url, for example:

```
iiop://myhost:2000?dataCompression=true&socketTimeout=20
```

Configuring error output

The client runtime writes errors to the console by default. In Java applications, you can modify this behavior by specifying the profile name as the Java system property djc.logFile. For example:

```
java -Ddjc.rmiTrace=true "-Ddjc.logFile=%DJC_HOME%\logs\rmiClientTrace.log"
```

For more information, see “Configuring system logging” in Chapter 3, “Creating and Configuring Servers,” in the *System Administration Guide*.

Running in Java applets

EJB clients that run as applets can set the APPLET parameter for the javax.naming.InitialContext instance used to connect to EAServer. For example:

```
java.util.Hashtable p = new java.util.Hashtable();
p.put(Context.APPLET, this);
```

```
// Sybase implementation of InitialContextFactory
p.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sybase.ejb.InitialContextFactory");

// URL for the Server's IIOP port. Host defaults to
// the applet download host.
p.put(Context.PROVIDER_URL, "iiop://:2000");

// Username "Guest", password is "GuestPassword"
p.put(Context.SECURITY_PRINCIPAL, "Guest");
p.put(Context.SECURITY_CREDENTIALS, "GuestPassword");

// Now create an InitialContext that uses the
// properties.
InitialContext ic = new InitialContext(p);
```

Setting the APPLET parameter activates the following convenient features:

- The host name can be omitted in the initial context URL that is specified as the PROVIDER_URL context parameter. The default host is the applet download host.
- You can set the com.sybase.ejb.autoProxy property and it will work as documented in Chapter 9, “Deploying Applications Around Proxies and Firewalls,” in the *EAServer Security Administration and Programming Guide*.

Resolving JNDI names

Call the Context.lookup method to resolve a bean’s JNDI name to a proxy for the bean’s home interface. If the server or cluster where the bean is installed has a name context configured, pass the server’s name context as part of the bean JNDI name, in the format:

```
Server-name-context/Bean-home
```

Where *Server-name-context* is the server’s initial naming context, and *Bean-home* is the component’s JNDI name, or, for server-side code executing in EJB or Web components, the aliased JNDI name in the calling component’s EJB reference properties.

Call javax.rmi.PortableRemoteObject.narrow to narrow the returned object to the bean's home (or local home) interface class. narrow requires as parameters the object to be narrowed and a java.lang.Class reference that specifies the interface type to returned. To obtain the java.lang.Class reference, use *Home*.class, where *Home* is the bean's home interface type. Cast the object returned by the narrow method to the bean's Java home interface.

The lookup method throws javax.naming.NamingException if the bean JNDI name cannot be resolved or the home interface proxy cannot be created. This can happen for any of the following reasons:

- *The server address* specified with the Context.PROVIDER_URL property is incorrect or the server is not running.
- *Authentication* with the specified credentials failed.
- *The bean* is incorrectly configured on the server. For example, a skeleton has not been generated, or the bean's properties specify the wrong implementation class.

Check the server's log file if the cause of the error is not clear from the exception's detail message.

The call below instantiates a proxy for a bean with Java home interface test.p1.Stateless1Home and bean JNDI name of test/p1/Stateless1:

```
import test.p1.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

try {
  Object o = ctx.lookup("test/p1/Stateless1");
  Stateless1Home home = (Stateless1Home)
    PortableRemoteObject.narrow(o, Stateless1Home.class);
} catch (NamingException ne) {
  System.out.println("Error: Naming exception: "
    + ne.getExplanation());
}
```

Instantiating remote or local interface proxies

Use the home interface create and finder methods to create proxies for session beans and entity beans.

Instantiating proxies for a session bean

A session bean's home interface can have several create methods. Each creates an instance with different initial-value criteria. The fragment below shows a typical call:

```
try {
  Inventory inv = invHome.create();
} catch (CreateException ce)
{
  System.out.println("Create Exception:"
    + ce.getMessage());
}
```

Instantiating proxies for an entity bean

Each instance of an entity bean represents a row in an underlying database table. An entity bean's home interface may contain both finder methods and create methods.

Finder methods Finder methods return instances that match an existing row in the underlying database.

A home interface may contain several finder methods, each of which accepts parameters that constrain the search for matching database rows. Every entity bean home interface has a findByPrimaryKey method that accepts a structure that represents the primary key for a row to look up.

Finder methods throw javax.ejb.FinderException if no rows match the specified search criteria.

Create methods Create methods insert a row into the underlying database.

When instantiating an entity bean proxy, call a finder method first if you are not sure whether an entity bean's data is already in the database. Create methods throw a javax.ejb.CreateException exception if you attempt to insert a duplicate database row.

Example: instantiating an entity bean This example instantiates an entity bean that represents a customer credit account. The primary key class has two fields: *custName* is a string and *creditType* is also a string. The example looks for a customer named Morry using the findByPrimaryKey method. If FinderException is thrown, the example calls a create method to create a new entity for customer Morry:

```
String _custName = "Morry";
String _creditType = "VISA";

custCreditKey custKey = new custCreditKey();
custKey.custName = _custName;
custKey.creditType = _creditType;
custMaintenance cust;
```

```
try {
  System.out.println(
    "Looking for customer " + _custName);
  cust = custHome.findByPrimaryKey(custKey);
} catch (FinderException fe) {
  System.out.println(
    "Not found. Creating customer " + _custName);
  try {
    cust = custHome.create(_custName, 2000);
  } catch (CreateException ce)
    System.out.println(
      "Error: could not create customer "
     + _custName);
  }
}
```

Calling remote interface methods

After instantiating a proxy for the bean, call the remote interface methods to invoke the bean's business logic. You can call the proxy methods as you would invoke methods on any other object.

Calling local interface methods

You can use EJB local invocations in servlet, JSP, or EJB component code to call EJB components hosted on the same server. Proxies for a local bean can be instantiated with almost the same code that would be used to instantiate remote proxies. The differences are:

- You must create a local EJB reference for the called EJB component, and use the aliased JNDI name defined in the EJB local reference.
- Parameters that are not primitive types are passed by reference, not by value. Changes to a parameter in the component implementation affect the variable passed from the caller.
- You must narrow to the local home interface type, not the home interface type.

- Local interfaces are available only to EJB components, Java servlets, and JSPs hosted on the same server as the target component.
- If local interfaces are used, both the caller and the called component must be loaded by the same custom class loader.

When an EJB 2.0 component provides local interfaces, any other component or Web application that calls the local interface must use the same class loader. ClassCastException errors occur when local interface calls are made from entities that use a different class loader. Configure the custom class lists for the calling and called components and parent entries to allow sharing of the class instances as described in Chapter 10, "Configuring Java Class Loaders," in the *System Administration Guide*.

Managing transactions

EJB clients can begin transactions using the javax.transaction.UserTransaction interface. Obtain an instance from the initial naming context by resolving the name javax.transaction.UserTransaction. For example:

```
import javax.transaction.*;
import javax.naming.*;

Context ctx;

... ctx has been initialized ...
UserTransaction uTrans =
   (UserTransaction) ctx.lookup(
     "javax.transaction.UserTransaction");
```

You can call the begin(), commit(), and rollback() methods to begin and end transactions. You can enlist multiple component methods in a transaction, with these restrictions:

- *Each method* must allow inheritance of an existing transaction context. That is, the method's transaction attribute must be Supports, Requires, or Mandatory. Methods with other transaction attributes run outside the scope of your transaction. See "EJB transaction settings" on page 6 for more information.
- *All components* must be on the same server, and all must use the same transaction coordinator.
- *All methods* must be invoked by the thread that began the transaction.

Serializing and deserializing bean proxies

Serialization allows you to save a bean proxy as a file. Deserialization allows you to extract the proxy from the file in another process or on another machine, and, if the component instance is still active, reestablish your session with the component.

To serialize a proxy

Call the getHandle method on the remote interface, which returns a javax.ejb.Handle instance. You can serialize the Handle instance using the standard Java serialization protocol, as shown in the example below:

```
String _serializeTo; // Name of file to save to
Stateful1 proxy;     // Active proxy instance

try {
  System.out.println("Serializing to " + _serializeTo);
  Handle handle = proxy.getHandle();
  FileOutputStream ostream = new
  FileOutputStream(_serializeTo);
  ObjectOutputStream p = new
  ObjectOutputStream(ostream);
  p.writeObject(handle);
  p.flush();
  ostream.close();
} catch (Exception e)
    {
  System.out.println("Serialization failed. Exception "
    + e.toString());
  e.printStackTrace();
  return;
    }
```

To deserialize the proxy

Use the standard Java deserialization protocol to extract the Handle instance, then call getEJBObject to restore the proxy, as shown in the example below:

```
String _serializeFrom; // Name of file to read from
Stateful1 proxy;

try {
  System.out.println("Deserializing proxy from "
    + _serializeFrom);
  FileInputStream istream = new
  FileInputStream(_serializeFrom);
  ObjectInputStream p = new ObjectInputStream(istream);
  Handle handle = (Handle)p.readObject();
  proxy = (Stateful1) handle.getEJBObject();
  istream.close();
```

```
} catch (Exception e)
{
  System.out.println(
    "Deserialization failed. Exception "
    + e.toString());
  e.printStackTrace();
  return;
}
```

Using EJB providers

An EJB provider allows you to define a set of client property settings that can be referenced by name when running clients. For example, you might wish all clients to run with data compression enabled. You can define EJB providers with the Management Console or using an XML configuration file. To set up an EJB provider for use by clients, you must:

- Deploy the client using an export configuration that includes the EJB provider definition, as described in Chapter 7, “Exporting Server Modules,”in the *System Administration Guide*.
- Add the EAServer *lib* subdirectory to the operating system library search path. On Windows, add it to the PATH environment variable. On UNIX or Linux, add it to the environment variable that spefies the directory search order to load shared libraries, for example LD_LIBRARY_PATH for Solaris and Linux.

To run clients with the EJB provider settings, specify the name in the value of the java.naming.provider. url property using the format described in “URLs to specify an EJB Provider name” on page 50.

Properties set in the EJB provider definition take precedence over those set in the EJB client code or in Java system properties.

EJB providers cannot be used in applets, because this feature requires access to configuration files in the EAServer installation directory.

❖ Creating EJB providers in the Management Console

1 Start the Management Console and connect to EAServer as described in Chapter 1, “Getting Started,”in the *System Administration Guide*.

2 Expand the Naming Providers folder and EJB Providers folder beneath it.

3 Right-click the EJB Providers folder and choose Add. Run the Add wizard to create the EJB provider with a unique name.

4 Configure the properties listed in Table 3-2.

❖ **Configuring EJB providers in the Management Console**

1 Start the Management Console and connect to EAServer as described in Chapter 1, "Getting Started,"in the *System Administration Guide*.

2 Expand the Naming Providers folder and EJB Providers folder beneath it.

3 Highlight the name of your EJB provider to display its properties in the right frame. Configure the properties listed in Table 3-2. Each property configures the indicated initial context property. See Table 3-1 on page 46 for descriptions of the initial context properties.

Table 3-2: EJB Provider settings

Display name	Initial context property
JNDI Initial Context Factory	java.naming.factory.initial
JNDI Provider URL	java.naming.provider.url
JNDI Security Principal (User name)	java.naming.security.principal
JNDI Security Credentials (Password)	java.naming.security.credentials
Idle Connection Timeout	com.sybase.ejb. idleConnectionTimeout
Lookup Cache Timeout	com.sybase.ejb. lookupCacheTimeout
Socket Timeout	com.sybase.ejb. socketTimeout
Enable Automatic Failover	com.sybase.ejb. disableAutomaticFailover
Enable Data Compression	com.sybase.ejb. enableAutomaticFailover

❖ **Creating and configuring EJB providers in an Ant configuration file**

You can create and configure an EJB provider in Ant configuration files, such as that for the EJB package that contains the components that your application invokes.

1 Edit the XML configuration file. In a configuration target, add an invocation of the setProperties task with the following format:

```
<setProperties ejbProvider="provider">
  <prop1="value1"/>
  <prop2="value2"/>
  ...
```

```
</setProperties>
```

Where:

- *provider* is the EBJ provider name.
- *prop1*, *value1*, *prop2*, *value2* are the names and values, respectively, of the properties to be set. Use the initial context property names listed in Table 3-2 on page 58.

2 Configure or recompile the entity that uses the configuration file. EAServer creates the EJB provider if it does not exist then sets the properties to match the values given in the configuration file.

Running EAServer 5.*x* clients against 6.0 or later servers

Client code from applications running in an EAServer 5.*x* installation can be recompiled to run in EAServer 6.0. You may need to change the initial context properties used in your code—see “Sybase InitialContext properties” on page 46.

You can also connect to EAServer 6.0 servers using the runtime classes from an EAServer 5.*x* installation (those classes in *easclient.jar*). To use both the 5.*x* runtime classes (*easclient.jar*) and 6.0 runtime classes (*eas-server-14.jar* or *eas-server-15.jar*) in the same application:

- Put *easclient.jar* ahead of *eas-server-14.jar* or *eas-server-15.jar* in the CLASSPATH setting
- Use initial context factory `com.sybase.ejb.InitialContextFactory` for 5.*x* client access, and `com.sybase.ejb.client.InitialContextFactory` for 6.0 client access.

If only *eas-server-14.jar* or *eas-server-15.jar* is present in the class path, both context factory names work, and use 6.0 client runtime classes.

CHAPTER 4

Creating Application Clients

EAServer supports the J2EE application client model. An application client is a standalone Java application that uses the EJB client interface to invoke components on EAServer and is run by the EAServer application client container. This model simplifies the deployment of standalone EJB client applications by allowing you to configure the client's component references, database connection references, and environment properties in the deployment descriptor.

Topic	Page
About application clients	61
Deploying application clients	62
Configuring application client properties	64
Running application clients	65

About application clients

An application client uses JNDI to look up and gain access to EJB components, resources, and environment properties defined in an XML deployment descriptor. You can code the application client like a standalone EJB client. However, since the application client is packaged with a deployment descriptor, you can alias the JNDI names used in code to different names when deploying to different servers.

An application client connects to an EAServer component using a JNDI environment naming context. Here is a simple implementation of an application client:

```
InitialContext initCntxt = new InitialContext();

Object acctRef =
   initCntxt.lookup("java:comp/env/ejb/acctBean");
acctBeanHome home = (acctBeanHome)
   PortableRemoteObject.narrow(acctRef,
   acctBeanHome.class);
Account acct = home.findByPrimaryKey(new
```

```
    AcctPK(1));
String name = acct.getName();
System.out.println(name);
```

The application client JAR file includes a deployment descriptor that defines the JNDI environment naming context entries. This example defines the EJB reference for an acctBean:

```
<application-client>
 <display-name>MyClient</display-name>
 <ejb-ref>
   <ejb-ref-name>ejb/AcctBean</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <home>com.sybase.acct.acctBeanHome</home>
   <remote>com.sybase.acct.Account</remote>
 </ejb-ref>
</application-client>
```

Deploying application clients

You can create application clients with Ant, Eclipse, and popular IDEs such as Borland JBuilder. Code your application clients as an EJB client or a JMS client. For more information, see:

- Chapter 3, “Developing EJB Clients”
- Chapter 3, “Developing JMS Clients,” in the *JMS Users Guide*

Application clients can be packaged as a standalone JAR file or as part of an enterprise application archive (EAR) file. You can deploy the JAR file to EAServer using several tools including:

- The Management Console
- The deploy utility, as described in Chapter 12, “Command Line Tools,” in the *System Administration Guide*
- The jagant and jagtool utilities

❖ **Deploying application client JAR files with the Management Console**

1 Start the Management Console and connect to EAServer as described in Chapter 1, “Getting Started,”in the *System Administration Guide*.

2 In the Management Console, right-click the Application Clients folder in the left pane and choose Deploy. The Deploy wizard pages appear in the right pane. Fill in the Wizard settings as listed in the table below.

Setting	Explanation
File Name	Enter the full path to the EJB-JAR file, or browse to select the file.
Module Name	Enter a name for the application to be created. The default is the base name of the JAR file. This name will appear in the Management Console and in configuration files.
Overwrite If This Name Already Exists.	Enable to overwrite an existing application client that has the same name.
Do Validation During Deployment	Enable to enforce validation of the XML deployment descriptor. In accord with the J2EE specification, this option is enabled by default. Disabling validation may allow you to deploy archives that have invalid XML but are otherwise correctly packaged.
Server Name	The name of the server to install the module to. The default is the name of the server that you are connected to.
Install Module Into Selected Server	Deselect if you don't want the package installed into any server. The package must be installed in a server before clients of that server can call the components.
Directory Name	If you specify a directory name, EAServer creates a copy of the archive file in the specified directory. The copy is identical to the source, except that an EAServer configuration file is added to the *META-INF* directory. The directory must exist and be a full path or a path relative to the EAServer *bin* directory.

Click Finish on the final wizard page. The Management Console deploys the EJB-JAR and shows the deployment status in the right pane. When your browser finishes downloading the status page, scroll to the bottom. You see Build Successful if everything went ok.

Configuring application client properties

You can configure an application client's properties in the Management Console. If you have deployed application clients as an Enterprise archive (EAR) file, most properties are set correctly during the import process.

❖ **Configuring application client properties**

1 In the Management Console, expand the Application Clients folder in the left pane, then highlight the icon that represents your application client.

2 The application client properties display in the right pane, on these pages:

- General tab
- Configuration tab
- Advanced tab

On each page, click Apply to save any changes made before leaving that page.

General tab

The controls on this page configure high-level properties for the application client, including:

- **Description** An optional text description of the application client.
- **Classloader** The Java class loader for the application client. For more information, see Chapter 10, "Configuring Java Class Loaders" in the *System Administration Guide*.
- **Client Type** The type of client, EJB or JMS. This setting must be set before you can configure an EJB Provider or JMS Provider.
- **EJB Provider** If the client type is EJB, allows you to configure an EJB Provider for the client. The EJB provider allows you to configure a group of client properties. For more information, see "Using EJB providers" on page 57.
- **JMS Provider** If the client type is JMS, allows you to configure a JMS Provider for the client. The JMS provider allows you to configure a group of client properties. For more information, see Chapter 3, "Developing JMS Clients," in the *JMS Users Guide*.
- **Main Class** The main Java class of the application client in dot notation; for example, `com.sybase.appclient.Myclient`.

Configuration tab

This page allows you to view and edit the generated Ant configuration file for the application client. As an alternative to using the other pages, you can edit the configuration file to modify the application properties. For more information, see Chapter 2, "Ant-Based Configuration," in the *Automated Configuration Tools Guide*.

Advanced tab

This page provides an alternate way to configure properties set by the configuration script.

Running application clients

You can run application clients on the same machine as the server, or on client workstations.

❖ **To run an application client:**

1 If running on a client workstation rather than the server host, set up the environment—see "Setting up a client's workstation" on page 65.

2 Copy the application client JAR file to the client's machine and import the JAR file—see "Deploying application clients" on page 62.

3 Start the application client's runtime container—see "Starting the runtime container" on page 66.

Setting up a client's workstation

To set up a client's workstation, install the EAServer client runtime files, as described in the *EAServer Installation Guide* for your platform. UNIX scripts and Windows batch files are provided to configure and launch the container runtime, as described below.

Starting the runtime container

The runtime container enables the application client to look up EJB and resource references. The container also provides security and authenticates the client when the application is started.

Run application clients using the script *run-appclient.bat* (on Windows) or *run-appclient.sh* (on UNIX). The script takes the following arguments:

-client <*appclient_name* | *appclient_jar* | *application_ear*> [-name <*appclient_name*> [-login <*boolean*>] [-systemProperties *propsFile*]

Option	Description
`-client`	Required, to specify one of the following: • The application client name, as displayed in the Management Console. This option works for all clients and does not require the deployment archive to be present. • The application client JAR file name and path, if the client was deployed from a standalone JAR file. • The application EAR file name and path, if the client was deployed from the EAR file. Also specify the application client name with the `-name` option.
`-name`	When you specify an application EAR file for the `-client` option, specify the name of the application client inside the EAR with this option. The name must match the base name of the application client JAR file inside the EAR file. For example: `-client C:\apps\myapp.ear -name myappclient`
`-login`	If you specify true for this option, the script prompts the user for login credentials. If you do not specify this option, the default is false.

`-systemProperties`	Specifies the name and path to a file that contains InitialContext property settings for the client. For example, to specify a user name and password, create a file that contains this text:
	`java.naming.security.principal=Guest`
	`java.naming.security.credentials=Guest123`
	An alternate way to specify properties is to configure an EJB or JMS provider on the General tab properties for the client—see “General tab” on page 64.

This example illustrates the command-line syntax to start an application client’s runtime container, where *my_client* is the name of the client. On Windows, the command is:

```
runclient -client my_appclient -login true
```

On UNIX, the command is:

```
runclient.sh -client my_appclient -login true
```

CHAPTER 5

Interoperability

EAServer not only hosts EJB components, it provides interoperability between EJB clients and components and other technologies. There are two areas of interest for EJB interoperability:

- Intervendor EJB interoperability, or how you can use EAServer with other EJB application servers.
- Intercomponent interoperability, or how you combine EJB components hosted in EAServer with components of other types in the same application.

This chapter describes:

Topic	Page
Intervendor EJB interoperability	67
Classes for RMI/IIOP connections from third-party containers	70
Invoking EJB components from PowerBuilder clients	70
Invoking EJB components from CORBA/C++ clients	71

Intervendor EJB interoperability

EAServer complies with the interoperability requirements in the EJB 2.0 specification, allowing you to interoperate with EJB 2.0 compliant servers from other vendors. There are two approaches to inter-vendor interoperability:

- **Using CORBA 2.2 client interfaces** This option allows interoperability between EAServer and other vendors that support CORBA 2.2.

  Using the EAServer Java or C++ CORBA client model, you can call another vendor's CORBA 2.2 compliant application server (the server must support IIOP 1.0 or 1.1). Similarly, you can use another vendor's CORBA 2.2 compliant client ORB to call any component hosted by EAServer (the client ORB must support IIOP 1.0 or 1.1).

This option is simpler than the EJB 2.0 RMI/IIOP option, but does not support some EJB 2.0 interoperability features such as transaction and security context propagation.

- **Using EJB 2.0 RMI/IIOP interoperability** This option allows interoperability between EJB 2.0 compliant application servers, but can be more complex to program, particularly in languages other than Java.

  RMI/IIOP interoperability depends on CORBA 2.3 IDL Valuetypes, which has the following implications:

  - Valuetypes and other IIOP 1.3 features cannot be used by pre-CORBA-2.3 client ORBs.
  - At the time of this writing, standard support for RMI/IIOP clients (specifically Valuetypes) in languages other than Java is lacking.

  RMI/IIOP interoperability supports some features not supported by CORBA 2.2 interoperability, such as:

  - Interoperable naming, when using the interoperable name formats described in “Interoperable naming URLs” on page 69.
  - Transaction propagation to other EJB servers.
  - Security context propagation in accordance with the CSIv2 requirements outlined in the EJB 2.0 specification. For more information on this feature, see “Intercomponent authentication for EJB 2.0 components” in the *EAServer Security Administration and Programming Guide*.
  - Parameter and exception type inheritance and null value propagation in method invocations.

EAServer supports RMI/IIOP interoperability for EJB clients and components, without using CORBA 2.3 Valuetypes in the IDL interface definitions. The generated stub and skeleton code can marshall parameters in accord with the RMI/IIOP requirements, even though the IDL does not use Valuetypes. Since the IDL does not use Valuetypes, EAServer EJB components remain compatible with components of other types and with CORBA 2.2 clients.

EAServer can simultaneously support RMI/IIOP and CORBA 2.2 clients. The client’s interoperability requirements are automatically detected at run time. To use RMI/IIOP from another vendor’s EJB 2.0 container, you must use the EAServer classes described in “Classes for RMI/IIOP connections from third-party containers” on page 70.

Interoperable naming URLs

You can use interoperable naming URLs for EJB 2.0 components and clients. EAServer versions prior to 6.0 required that you use interoperable naming URLs to enable the RMI/IIOP protocol, which is required for EJB 2.0 interoperability features such as caller credential propagation. EAServer 6.0 and later always use the RMI/IIOP protocol, and you can use the simplified URL formats described in "Connection URL formats" on page 48. Interoperable naming URLs are not required to connect to EAServer.

Interoperable naming URLs may be useful when integrating other vendor's application servers into an EAServer application, for example, when pulling messages from another vendor's JMS implementation.

To use RMI/IIOP as the network protocol, an EJB client must specify a `corbaname` interoperable naming URL as the value of the JNDI context's PROVIDER_URL property. When using `corbaname` URLs, you must specify the user name and password using the JAAS API.

When using the EAServer EJB client runtime, the URL syntax is:

corbaname:iiop:*ver*@*host*:*port*/NameService*[rmi]*

Or to use the default IIOP version number:

corbaname:iiop:*host*:*port*/NameService*[rmi]*

Where:

ver	Is an optional version number. Supported versions are 1.1 and 1.2. The default version is 1.1, unless you append the `#rmi:/` suffix, which forces the IIOP version to 1.2.
host	Is the server host name.
port	Is the server's IIOP port number.
[rmi]	Is the optional naming prefix `#rmi:/`, which specifies RMI Valuetype semantics. Valuetype semantics are required to propagate null parameter values in method calls. If connecting to EAServer 5.x, you must specify append `#rmi` for this part of the URL. EAServer 6.0 and later EJB clients always connect with valuetype semantics enabled. Using this option forces the IIOP version to 1.2.

For example, this URL specifies a connection to the host moxy at port 2000, using IIOP 1.2:

corbaname:iiop:1.2@moxy:2000/NameService

To connect to an EAServer 5.x server, which does not use valuetype semantics by default, append `#rmi` to the URL:

```
corbaname:iiop:1.2@moxy:2000/NameService#rmi:/
```

This URL identifies a connection to the host moxy at port 2000, using IIOP 1.1:

```
corbaname:iiop:moxy:2000/NameService
```

The string `/NameService` is optional in all `corbaname` URLs. For example:

```
corbaname:iiop:1.2@moxy:2000#rmi:/
```

Or:

```
corbaname:iiop:1.2@moxy:2000
```

Classes for RMI/IIOP connections from third-party containers

To connect to EAServer using another vendor's EJB 2.0 client, application client, EJB, or servlet or JSP within a Web container, add *easportable.jar* to the CLASSPATH. *easportable.jar* is located in the EAServer *lib* subdirectory and contains the classes in the com.sybase.ejb.portable package. These classes are:

- EJBMetaData
- Handle
- HomeHandle

Adding *easportable.jar* to the CLASSPATH enables you to call these methods on a `javax.ejb.EJBHome` or `javax.ejb.EJBObject` instance residing on EAServer:

- getEJBMetadata
- getHandle
- getHomeHandle

Invoking EJB components from PowerBuilder clients

There are two ways to call EJB components from PowerBuilder:

- If using PowerBuilder, you can call EJB components hosted in EAServer by generating proxies for the home, local, and remote interfaces then calling the lookup method on the PowerBuilder Connection object to instantiate the home interface proxy. Call the appropriate home interface create method to instantiate a proxy for the remote interface, then call the business methods as you would for any other EAServer component.
- If using PowerBuilder 9.0 or later, you can use the PowerBuilder EJB client interfaces. These interfaces use Java and Sybase-provided PowerBuilder extensions to invoke EJBs on any J2EE compatible application server. While this approach allows interoperability with servers from multiple vendors, the deployed client files are larger due to the need for a Java Runtime Environment and additional PowerBuilder libraries.

For more information, see the *Application Techniques* manual in the PowerBuilder documentation.

Invoking EJB components from CORBA/C++ clients

CORBA C++ clients can instantiate an EJB component using a proxy for the EJB component's home interface, then call business methods using a proxy for the EJB component's remote interface.

For details on the C++/CORBA client interface, see Chapter 9, "Developing CORBA/C++ Clients," in the *CORBA Components Guide*.

Supported datatypes

C ++ clients can call methods that are defined using only IDL datatypes. EAServer allows serializable Java classes to be used as parameters and return values. IDL operations that use Java classes as a parameter or return value cannot be called from C++ clients.

Generating required header files

For deployed EJB modules, EAServer generates IDL for each home and remote interface (unless you have disabled IDL generation with the `-noidl` deploy option). To call EJB components from C++ clients, you must generate C++ header files from the IDL types that define the component's home and remote interfaces.

You can find IDL for deployed components in the *Repository/idl* subdirectory. The IDL files use a folder structure that matches the IDL module structure, which corresponds to the Java package structure. For example, the Java package com.sybase.easerver.tutorials.ejb translates to the IDL module com::sybase::easerver::tutorials::ejb, and translated IDL types for classes in this Java package are in the subdirectory *com/sybase/easerver/tutorials/ejb*.

Use the idl-compiler tool to generate C++ stubs for each IDL home and remote interface. For example:

```
%DJC_HOME%\bin\idl-compiler.bat -v com\sybase\easerver\tutorials\ejb\Query.idl
-f %DJC_HOME%\include -cpp
%DJC_HOME%\bin\idl-compiler.bat -v
com\sybase\easerver\tutorials\ejb\QueryHome.idl -f %DJC_HOME%\include -cpp
```

For more information, see the idl-compiler reference page in Chapter 12, "Command Line Tools," in the *System Administration Guide*.

In the case of nested IDL modules, EAServer generates several header files. For example, for the IDL interfaces com::foo::interfaces::MyInterface and com::foo::interfaces::MyHomeInterface, these files are generated:

- *com.hpp* includes *com_foo.hpp* and headers for other modules nested within module com.
- *com_foo.hpp* includes *com_foo_interfaces.hpp* and headers for any other nested modules.
- *com_foo_interfaces.hpp* declares the C++ classes for com::foo::interfaces::MyInterface and com::foo::interfaces::MyHomeInterface, as well as any other types declared in module com::foo::interfaces.

In your client program, you must include only those header file that define types or interfaces used by your program. For example, if you use the types com::foo::interfaces::MyInterface and com::foo::interfaces::MyHomeInterface, include the header file *com_foo_interfaces.hpp*.

Calling the home interface

The C++ representation of the home interface follows the standard IDL-to-C++ language mappings. In EAServer's interface repository, the EJB FinderException and CreateException exceptions are represented by the IDL exceptions javax::ejb::FinderException and javax::ejb::CreateException, respectively.

Instantiating a proxy for the home interface

To instantiate a home interface, use a SessionManager::Manager instance to create a SessionManager::Session instance, then call the SessionManager::Session::lookup method, passing the EJB component's home interface name. Narrow the returned object to the C++ class for the EJB component's home interface.

In this example, the IDL home interface is bookStore::custMaintenanceHome and the EJB component's home interface name is *bookStore/custMaintenance*:

```
 // Initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, 0);

// Obtain a SessionManager::Manager instance using the URL:
CORBA::Object_var obj =
    orb->string_to_object(url);
SessionManager::Manager_var manager =
  SessionManager::Manager::_narrow(obj);

// Create an authenticated session for user Guest
// using password GuestPassword
SessionManager::Session_var session =
    manager->createSession("Guest", "GuestPassword");


// Look up the EJB component's home interface
obj = session->lookup(component_name);
bookStore::custMaintenanceHome_var home
    = bookStore::custMaintenanceHome::_narrow(obj);
```

Instantiating a session bean

To instantiate a session bean, call one of the home interface create methods as shown in the example below. All create methods can raise javax::ejb::CreateException. The example below instantiates the home of a bean with home interface name *bookStore/inventory*. The IDL remote interface is bookStore::inventory:

```
try {
    bookStore::inventory_var inventory = home->create();
}
catch (javax::ejb::CreateException &ce)
{
    cout << "CreateException for component " << component_name << "\n"
         << "Message:" << ce.message << "\n";
}
```

Instantiating an entity bean

An entity bean represents a row in a database relation. In the home interface, create methods create a row in the database, and finder methods return one or more instances that represent existing rows. All create methods can raise javax::ejb::CreateException, and finder methods can raise javax::ejb::FinderException. The example below first tries to find an existing row using findByPrimaryKey, and creates a row if javax::ejb::FinderException is thrown. The entity bean in this example represents customer credit data. The primary key, bookStore::custCreditKey, has two string fields, *custName* and *creditType*. The IDL remote interface is bookStore::custMaintenance:

```
// Initialize a primary key for the bean
bookStore::custCreditKey custPk;
custPk.custName = CORBA::string_dup(customer_name);
custPk.creditType = CORBA::string_dup(credit_type);

bookStore::custMaintenance_var customer;
long balance = 2000;

// Look for an existing instance.
try {
    cout << "Looking for customer named " << customer_name << "\n";
    customer = home->findByPrimaryKey(custPk);
} catch (javax::ejb::FinderException &fe)
{
    // Instance does not exist. Create it.
    cout << "Customer " << customer_name << " does not exist. "
        << "Creating " << customer_name << " with initial balance of "
```

```
        << balance << ".\n";
    customer = home->create(customer_name, balance);
}catch (javax::ejb::FinderException &fe)
{
    cout << "Error creating account for customer " << customer_name ;
}
```

Serializing and deserializing instance references

An EJB client is allowed to obtain a **handle** for a remote interface instance. The handle is a binary encoding of the session state between the client and the bean. The client can obtain a handle, save it to disk or mail it to another location, then reestablish the session at a later time.

In a CORBA client, you can obtain the same functionality using the Orb.object_to_string and Orb.string_to_object methods. The same restrictions apply when deserializing bean proxies that apply to any other remote object.

CHAPTER 6

Tutorial: Creating Enterprise JavaBeans Components and Clients

In this tutorial, you will create two Enterprise JavaBeans (EJB) components and a simple client to verify their operation.

Topic	Page
Overview of the sample components	77
Tutorial requirements	77
Creating the application	78
Automating deployment and configuration	87

Overview of the sample components

The application consists of two components used to manage a glossary of terms and components, and a relational database that manages the glossary data. You will create an EJB entity bean that allows you to search for keyword entries, create and delete keyword entries, and modify definitions. You will also create an EJB stateless session bean that can be used to run arbitrary select queries against the database.

Tutorial requirements

To create the tutorial application, you need:

- The EAServer software

  The *EAServer Installation Guide* for your platform describes how to install the software.

- Java development environment

The tutorial steps use the JDK software and Apache Ant software that is included with your EAServer installation. You can also use Eclipse, JBuilder, or any other development tool that is compatible with JDK 1.4 or later.

- Adaptive Server Anywhere, version 9.0 or later

  The sample database requires Adaptive Server Anywhere, version 9.0 or later. This software is optionally installed with EAServer on Windows and supported UNIX platforms such as Solaris, HP-UX, Linux, and IBM AIX. See the *EAServer Installation Guide* for installation instructions.

  If you are using another platform, see the *EAServer Release Bulletin* for your platform.

Creating the application

To create and run the sample application:

1 Start EAServer and the Management Console.

2 Create the EJB-JAR file.

3 Deploy the EJB-JAR file to EAServer.

4 Create the glossary database and data source.

5 Create a user account.

6 Create the client application.

7 Run the client application.

Start EAServer and the Management Console

Start the Management Console and connect to EAServer as described in Chapter 1, “Getting Started,”in the *System Administration Guide*.

Create the EJB-JAR file

The source for the EJB components and the EJB-JAR manifest are provided in the *samples/tutorials* directory of your EAServer installation. Copy the contents of the *ejb* subdirectory to a location where you can modify the files. If necessary, change the permissions on this copy to allow write access.

Review the project layout component source files

You will use Ant project builds the EJB-JAR from the provided Java source files and the deployment descriptor. Source files are beneath the *src* directory in package subdirectory *com/sybase/easerver/tutorials/ejb*. The master copy of the deployment descriptor is *manifest/META-INF/ejb-jar.xml*.

Table 6-1 describes the source files for the Glossary entity bean, which interacts with the database to manage a glossary of terms and definitions:

Table 6-1: Glossary entity bean source files

File	Purpose
Glossary.java	The remote interface
GlossaryHome.java	The home interface
GlossaryBean.java	Implements the component methods

Table 6-2 describes the source files for the Query stateless session bean, which implements methods to lookup definitions by calling the Glossary bean.

Table 6-2: Query stateless session bean source files

File	Purpose
Query.java	The remote interface
QueryHome.java	The home interface
QueryBean.java	Implements the component methods

The Ant project, defined by *build.xml*, does the following:

- Compiles the component source files to the *jar-image* directory, creating the directory if necessary.
- Copies the deployment descriptor to the same location.
- Archives the contents of the *jar-image* directory to create the EJB-JAR file, *ejbtut.jar*.

If you wish, you can review the Ant commands to perform these tasks in the definition of the `ejbjar` target in *build.xml*. However, you don't need to know Ant to run the tutorial. Documentation for Ant can be found on the Apache Web site at http://ant.apache.org/.

Build the EJB-JAR file

Edit the file *settings.bat* using a text editor. Change the setting of the `DJC_HOME` setting to match the location of your EAServer install. At a command prompt, change to your working directory for the tutorial and run the following command to build the EJB-JAR file:

```
./build
```

This command runs build.bat, which runs Ant to build the default target in the project, the ejbjar target in *build.xml*. If successful, you should see *ejbtut.jar* created in the same directory.

Deploy the EJB-JAR file to EAServer

We use the Management Console to deploy the EJB-JAR file. You could also use the deploy command-line tool, or jagant, or jagtool.

❖ **Deploying the EJB-JAR file**

1 If you haven't already, start EAServer and connect to the preconfigured server with the Management Console as described in Chapter 1, "Getting Started,"in the *System Administration Guide*.

2 In the Management Console, right-click the EJB Modules folder in the left pane and choose Deploy. The Deploy wizard pages appear in the right pane. Fill in the Wizard settings as follows:

   a *File Name* – Enter the full path to the *ejbtut.jar* file, or browse to select the file.

   b *Module Name* – Accept the default of `ejbtut`.

   c *Server Name* – Accept the default, which should be the name of the server that you are connected to. Verify that the check box is selected to install the module into this server.

   d Optional location to store archive – Leave this blank.

Click Finish on the final wizard page. The Management Console deploys the EJB-JAR and shows the deployment status in the right pane. When your browser finishes downloading the status page, scroll to the bottom. You see Build Successful if everything went ok.

Create the glossary database and data source

The glossary data is stored in an Adaptive Server Anywhere database. To make the data available to the components, we must define the data source that the components use to connect to the database.

Copy the database file to the EAServer installation

If your EAServer installation does not include a *data* subdirectory, create this directory. Copy *gloss.db* from the *data* directory of the EJB tutorial files to the EAServer *data* subdirectory.

In the tutorial, we configure EAServer to start and stop the database server automatically. Make sure the *data* subdirectory allows read, write and execute for the user account that you use to run EAServer. Verify that the *gloss.db* file permissions allow read and write access.

Create the data source

A data source maintains a pool of connections to a database server, increasing performance by allowing connection sharing and reuse.

❖ **Creating the data source**

1 In the Management Console, expand the Resources folder and locate the JDBC Data Sources folder beneath it. Right-click on JDBC Data Sources folder and choose Add.

2 In the New Data Source wizard, enter Glossary as the name. Finish the wizard to display the properties for the new Glossary data source.

3 Configure the properties on the General tab listed in Table 6-3, then click Apply to save the changes. Leave properties that are not listed in Table 6-3 at their default setting.

Note Be sure to click Apply to save changes when switching between tabs in property pages.

The Database Type setting determines the default settings for other data source properties. In this case, setting it to `Sybase_ASA` (to indicate Sybase Adaptive Server Anywhere) causes EAServer to use Sybase jConnect as the database driver as well as making the default URL format the one used by jConnect. You can define and modify database types to change the default data source property values for that database type.

Table 6-3: Data source General tab settings

Setting	Value
Database Type	`Sybase_ASA`
Server name	localhost
Port Number	`2640` **Port numbers must match** Make sure the port number specified here matches the port number you specify in the start command.
User name	`dba`
Password	`sql`

4 Configure the Database tab settings listed in Table 6-4 and click Apply. These settings allow EAServer to start and stop the server automatically. Leave settings that are not listed at their default value.

Table 6-4: Data source Database tab settings

Setting	Value
Database File	~/data/gloss.db
Database Create Command	Leave blank. This setting can be configured to create a database file for EJB CMP entity beans. We have already created the database file.
Database Start Command	`asa-start${.bat} -x tcpip(ServerPort=2640) -n ${dataSource} ${databaseFile}` This command runs asa-start (located in the EAServer *bin* directory), starting the server to listen on port 2640, with engine name matching the data source name (`Glossary`), and database file matching the value of the Database File setting (*gloss.db*). **Port numbers must match** Make sure the port number used in the General tab properties matches the port number you specify in the start command. If you have a port number conflict, change the port number in both settings.
Database Stop Command	`asa-stop${.bat} -y -c "uid=dba;pwd=sql;eng=${dataSource}"` This command runs asa-stop (located in the EAServer *bin* directory), stopping the server. The command specifies the user name and password required to connect and the engine name.

Setting	Value
Database Command Echo	Enabled.
Database Command Destroy on Exit	Disabled.

5 Verify the data source properties as follows:

a Expand the Data Sources folder.

b Right-click the Glossary data source and choose Ping.

Since the server is starting the database, it will take some time before Ping completes. If the Ping operation fails, confirm that you have applied the settings correctly and that the database is running. You can see the database commands on the server console and in the server log file.

Modify the data source used by the EJB components

EJB components obtain database connections using JNDI. The EAServer component and package properties bind data source to the JNDI names used in the component implementation.

When we deployed the EJB-JAR file, EAServer bound the JNDI data source names to the default data source. We must reconfigure the properties to use the new data source that we just created.

❖ **Changing the data source used by the EJB components**

1 In the Management Console, expand the EJB Modules folder and click on the ejbtut icon beneath it. The package configuration properties display.

2 Display the Configuration tab. This tab displays the Ant configuration file that EAServer generated from the settings in the EJB-JAR deployment descriptor. Since this configuration file is regenerated if you redeploy the components, we will not modify it. Instead we will create a user-configuration file.

3 Right-click the ejbtut icon beneath the EJB Modules folder and choose Create User Configuration. You see a dialog saying the User Configuration File is created. Right-click the ejbtut icon again and choose Refresh to display the User Configuration tab.

4 Click the User Configuration Tab to display the file contents. The configuration information is the XML text of an Ant build file that sets the package and component properties.

Add the following line above the `import` elements to define the data source name that the components connect to:

```
<property name="sql.dataSource" value="Glossary"/>
```

Find the definition of the `configure-user` target, and add the statements below inside the start and end tags for the target. The target definition in your file should look like this:

```
<target name="configure-user">
  <echo level="info" message="configure: ejbjar-ejbtut-user"/>
  <echo level="info" message="sql.dataSource is ${sql.dataSource}"/>
</target>
```

Click Apply at the bottom of the page to save the changes.

Structure of the configuration file

The generated configuration file contains several Ant property elements. These define values that can be used in subsequent setProperties tasks. In this case, the sql.dataSource property is used in the setProperties task for the ejbtut package, to bind the JNDI name to the data source name. By setting this property in the user configuration file, you have overridden the definition of the same property in the generated configuration file. The commands in the generated configuration file bind the datasource name (specified by the Ant property) to the JNDI name used in the component code (specified by the EJB-JAR deployment descriptor).

5 Right-click on the ejbtut icon beneath the EJB Modules folder and choose Run Ant Recompile. The Management Console runs the Recompile task defined in the configuration file, applying the changes to the package and component properties and regenerating the runtime classes required to integrate the components into EAServer.

Review the output that displays in the right pane. You see Build Successful at the bottom of the output if everything goes ok.

Create a user account

You must have a user account the client application uses to connect to the server. If you don't already have a user account defined, create it as described here. Alternatively, edit the client application source code to use an existing account.

❖ **Creating the Guest user account**

1 In the Management Console, expand the Security folder and right-click the Users folder beneath it. Choose Add from the context menu.

2 In the New User wizard, enter `Guest` as the user name and click Finish.

3 An icon appears for the Guest wizard under the Users folder. Right-click this icon and choose Set Password.

4 In the Set Password wizard, enter `GuestPassword2` for the password and click Apply.

Create the client application

The source for the client application is in the *client-src* directory of the EJB tutorial files, file *TestClient.java* in the package subdirectory *com/sybase/easerver/tutorials/ejb/client*.

The client connects to EAServer and calls the Glossary component to create three entries. It then calls the Query component to display all the entries in the database.

No changes are required to the source code unless you need to connect with a different account than the one created in the step "Create a user account" on page 85. In that case, edit the source and change the user name and password values.

Build the test client with Ant by running this command in your copy of the EJB tutorial directory:

```
build client
```

Run the client application

If you have not refreshed or restarted your server since last modifying the tutorial components, data source, or user account, refresh the server now before running the client. Otherwise, verify that the server is running.

Run the client from using a batch file or UNIX shell script. The batch file or shell script configures the CLASSPATH environment variable, then runs the application using the JDK 1.4 java program included with your EAServer installation.

❖ **Creating the Windows batch file**

- Create a file named *runtest.bat* in your copy of the EJB tutorial directory, containing the commands below:

```
@SETLOCAL
@echo off
call settings.bat
call %DJC_HOME%\bin\djc-setenv.bat
set JAVAHOME=%DJC_JAVA_HOME_14%
set CLASSPATH=%DJC_HOME%\lib\eas-server-14.jar
set CLASSPATH=%CLASSPATH%;%DJC_HOME%\deploy\ejbjars\ejbtut
set CLASSPATH=%CLASSPATH%;.\client-classes
set CLASSPATH=%CLASSPATH%;%DJC_HOME%\genfiles\java\classes
"%JAVAHOME%\jre\bin\java"
com.sybase.easerver.tutorials.ejb.client.TestClient %*
```

❖ **Creating the UNIX shell script**

1 Create a file named *runtest* containing the commands below:

```
#!/bin/sh
. settings.sh
. $DJC_HOME/bin/djc-setenv.sh
set JAVAHOME=$DJC_JAVA_HOME_14
set CLASSPATH=${DJC_HOME}/lib/eas-server-14.jar
set CLASSPATH=$CLASSPATH:${DJC_HOME}/deploy/ejbjars/ejbtut
set CLASSPATH=$CLASSPATH:./client-classes
set CLASSPATH=$CLASSPATH:${DJC_HOME}/genfiles/java/classes
export CLASSPATH
export JAVAHOME
"$JAVAHOME/jre/bin/java"
com.sybase.easerver.tutorials.ejb.client.TestClient $*
```

2 Change the file permissions to allow the script to be executed. For example:

```
chmod 777 runtest
```

❖ **Running the client application**

- Run the batch or script file, specifying the server host name and IIOP port number on the command line as follows:

```
runtest iiop://host:iiop-port
```

For example:

```
runtest iiop://myhost:2000
```

The client application:

1 Creates a proxy for the Glossary entity bean's home interface, then calls the create method to populate the database with some glossary entries.

2 Creates a proxy for Query session bean's home interface, then calls the runQuery method to get a result set containing all the entries, then prints them.

If errors occur, check the server log file for information on how to correct the problem.

Automating deployment and configuration

This section demonstrates how to modify the Ant build to deploy the EJB-JAR file to EAServer and how to use a Sybase configuration file to automate the post-deployment configuration steps.

❖ **Running the automated deploy and configure**

1 In your working copy of the tutorial files, copy the file *sybase-ejbjar-config.xml* from the *example-config* subdirectory to the *manifest/META-INF* directory.

The *sybase-ejbjar-config.xml* contains definitions of a `configure-user` and `recompile-user` target. EAServer calls these targets after deploying the EJB-JAR file. The `configure-user` target creates the data source used to connect to the database and the sample user account that the client uses to login.

2 Edit *build.xml* with a text or XML editor. Find the definition of the `eas.host` and `eas.http.port` properties. These must match the host name and HTTP port number of the server that you deploy to. As shipped, the `eas.host` value matches the string returned by the HOSTNAME environment variable (on UNIX platforms) or COMPUTERNAME environment variable (on Windows platforms). Edit the values if you are deploying to a server on another machine or the server uses a different HTTP port number than the default 8000.

3 Create a wlogin session to the server that you are deploying to, using the default system account, admin@system, or another account with admin-role membership. You can use the same account that you use to login to the Management Console. For information on wlogin, see Chapter 12, "Command Line Tools," in the *System Administration Guide.*

Wlogin treats host names as case-sensitive
If you are running on Windows, and you did not change the value of the `eas.host` and `eas.http.port` properties in the Ant build file, specify the host name in all uppercase letters when running wlogin. As installed, this property uses the value of the Windows COMPUTERNAME environment variable, which is typically set in all uppercase letters.

4 Optionally undeploy the tutorial components from EAServer. You can do this with the undeploy command-line tool. For example:

```
undeploy -server host:8000 ejbjar-ejbtut
```

For information on undeploy, see Chapter 12, "Command Line Tools," in the *System Administration Guide*.

5 At a command prompt, change to your working directory for the tutorial and run the following command to rebuild and deploy the EJB-JAR file:

```
./build deploy
```

This command runs *build.bat*, which runs Ant to build the `deploy` target. This target:

a Rebuilds *ejbtut.jar* (because this target depends on the JAR file).

b Deploys *ejbtut.jar* to the server address specified by the `eas.host` and `eas.http.port` properties.

c Copies the database file, *gloss.db*, to the *data* subdirectory in the EAServer installation if it is not already there. If successful, you should see the output of the EAServer deployment.

6 Restart or refresh the server and run the test client as described in "Create the client application" on page 85 and "Run the client application" on page 85.

Index

A

access control
  and port numbers 27
  configuring 25
  JACC 26
  types of 25
Adaptive Server Anywhere
  starting 81
  using in EJB tutorial 78
addresses, network
  restricting access by 27
  specifying in EJB clients 46
Ant
  configuring properties with 20
  example project 79
  properties 24
  used in tutorial 79
application clients 61–66
  configuring properties for 64
  creating 61
  deploying 65
  general properties 64
  running 65, 66
  set up for 65
  starting the runtime container 66
application server, definition of 2
applications
  client 61
  EJB 78
  EJB client 43
architecture
  EJB 2
  EJB component 1
authorization
  configuring 25
  JACC 26
  roles created by deployment 25
automatic transaction retry 37

B

batching transactions 36
by-reference parameter passing 28

C

C++
  clients 71
  datatypes 71
C++ clients,invoking EJB components in 71
clients
  application 61
  C++ 71
  EJB 43, 85
  error logging for 50
  PowerBuilder 70
clusters
  partitioning 28
  properties for 28
code
  for C++ client 73
  for EJB client 85
com.sybase.ejb.ProxyHost
  InitialContext property name 47
com.sybase.ejb.ProxyPort
  InitialContext property name 48
components
  C++ 71
  EJB 1, 19, 79
  PowerBuilder 70
configuring
  authorization 25
  EJB modules 19
  failover 27
  JNDI names 34
  properties 24
configuring, role references 17
connection timeout

configuring for EJB clients 46
conventions x
CORBA
interoperability with EJB 67
creating
data sources 81

D

data sources
associating with components 40, 83
associating with EJB components 83
creating 81, 83
databases
for EJB tutorial 81
starting automatically 82
datatypes, C++ 71
deploying
database files 81, 88
EJB-JAR files 9, 19, 88
development environments, Java 77

E

EAServer
EJB component support in 8
interoperability 67
EJB
See also EJB clients, EJB components
architecture 2
clients 43, 77, 85
creating components 79
EAServer support for 8
entity beans 77
generating stubs for 43
interoperability 67
interoperability with PowerBuilder 70
overview of 1
requirements for 77
sample components 77
session beans 77
timer service 13
tutorial for 77
tutorial steps 78
version levels 10
EJB client
definition of 3
generating stubs for 43
runtime requirements for 43
EJB components
and EJB version 10
configuring 19
creating 19
deploying 9, 19
exposing as Web services 10
introduction to 1
invoking from C++ clients 71
invoking from PowerBuilder clients 70
JNDI names for 16
properties of 20, 24
security configuration for 17
tutorial for 77
types of 3
undeploying 88
using transactions in 6
version levels and 16
EJB container
definition of 2
services provided by 5
EJB modules
configuring 19
deploying 19
undeploying 88
ejb.accessControl Ant property 25
ejb.allowedPorts Ant property 27
ejb.automaticFailover Ant property 27
ejb.clusterPartition Ant property 28
ejb.copyValues Ant property 28
ejb.enableProfiling Ant property 29
ejb.enableTracing Ant property 30
ejb.isolationLevel
values for 31
ejb.isolationLevel Ant property 30, 41
ejb.localNamePrefix Ant property 34
ejb.localNameSuffix Ant property 34
ejb.localThreadMonitor Ant property 34
ejb.logExceptoins Ant property 29
ejb.passivateTimeout Ant property 38
ejb.poolTimeout Ant property 39
ejb.remoteNamePrefix Ant property 35

ejb.remoteNameSuffix Ant property 35
ejb.remoteThreadMonitor Ant property 35
ejb.removeTimeout Ant property 38
ejb.rolePrefix Ant property 27
ejb.serviceThreadMonitor Ant property 36
ejb.transactionBatch Ant property 36
ejb.transactionRetry Ant property 37
EJB-JAR files
  creating 79
  deploying 9, 19, 88
  undeploying 88
Enterprise JavaBeans
  See EJB
entity bean
  definition of 3, 4
  EJB component type 3
  used in tutorial 79
errors
  logging 29
  logging in clients 50
exceptions
  logging 29

F

failover, configuring 27

G

generating EJB stubs 43

H

home interface
  instantiating in clients 45
  invoking with C++ 73
home interface, definition of 3

I

interoperability
  overview of 67
  with C++ clients and components 71
  with other application servers 67
  with PowerBuilder clients and components 70
isolation level
  configuring 30, 41
  explanation of 30, 41

J

JACC 26
Java development environments 77
JavaBeans
  See also EJB
jca.connectionFactory Ant property 39
JNDI names
  configuring 34
  in deployment descriptor 16

L

load balancing
  and cluster partitions 28
  properties for 28
logging
  for clients 50
  of exceptions 29

M

Mandatory
  method transaction attribute 8
message driven bean
  definition of 5

N

naming, properties for 34
Never, method transaction attribute 8
NotSupported, method transaction attribute 8

P

parameters, passing by reference 28
passivation 38
pooling 39
port numbers
  restricting access with 27
  specifying in EJB clients 46
PowerBuilder, invoking EJB components in 70
profiling
  enabling 29
  viewing data 29
properties
  Ant 24
  configuring 24
  data source 81
  EJB component 20, 24, 83

R

ReadCache transaction isolation level 31
ReadCacheVerifyUpdates transaction isolation level 31
ReadCommitted transaction isolation level 31
ReadCommittedVerifyUpdates transaction isolation level 31
ReadCommittedVerifyUpdatesWIthCache transaction isolation level 32
ReadCommittedWithCache transaction isolation level 32
remote interface, definition of 3
RepeatableRead transaction isolation level 33
RepeatableReadWithCache transaction isolation level 33
Required, method transaction attribute 8
requirements
  for EJB client runtime 43
  for EJB tutorial 77
RequiresNew
  method transaction attribute 8
resource references
  deployment settings for 40
  for EJB components 83
role references, configuring 17
roles
  and isCallerInRole method 17
  created by deployment 25, 27
  referencing in code 17
run-appclient command-line tool 66

S

security roles
  created by deployment 27
Serializizable transaction isolation level 33
SerializizableWithCache transaction isolation level 34
session bean
  definition of 3
  EJB component type 3
  stateful 4
  stateless 4
sessions, timeouts for 38
socketTimeout EJB initial context property 46
sql.createTables Ant property 39
sql.dataSource Ant property 40, 83
stateful session beans
  definition of 4
  timeouts for 38
stateless session bean
  definition of 4
  used in tutorial 79
stubs
  generating for EJB clients 43
stubs, EJB
  generating 43
Supports, method transaction attribute 8

T

thread monitors 34, 35, 36
timeouts
  for clients 46
  for entity beans 39
  for stateful session beans 38
  for stateless session beans 39
timeouts, connection
  for EJB clients 46
timer service 13
trace data 30
tracing, deployment property for 30
transactions
  automatic retry of 37
  batching 36
  bean-managed 6
  container-managed 6
  isolation level 30

method attribute for 7
use in EJB components 6
tutorial, EJB
creating the client 85
steps for 78
tutorials
client 77
EJB 77
typographical conventions x

U

undeploying EJB-JAR files 88
user names, specifying in EJB clients 46

V

versions, of EJB specificiation 10

W

Web services 10