



EJB Server User's Guide

Adaptive Server Enterprise

12.5

DOCUMENT ID: 33690-01-1250-01

LAST REVISED: June 2001

Copyright © 1989-2001 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase database management software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase, the Sybase logo, ADA Workbench, Adaptable Windowing Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Anywhere, Adaptive Server Enterprise, Adaptive Server Enterprise Monitor, Adaptive Server Enterprise Replication, Adaptive Server Everywhere, Adaptive Server IQ, Adaptive Warehouse, AnswerBase, Anywhere Studio, Application Manager, AppModeler, APT Workbench, APT-Build, APT-Edit, APT-Execute, APT-FORMS, APT-Translator, APT-Library, Backup Server, ClearConnect, Client-Library, Client Services, Data Pipeline, Data Workbench, DataArchitect, Database Analyzer, DataExpress, DataServer, DataWindow, DB-Library, dbQueue, Developers Workbench, Direct Connect Anywhere, DirectConnect, Distribution Director, E-Anywhere, E-Whatever, Embedded SQL, EMS, Enterprise Application Studio, Enterprise Client/Server, Enterprise Connect, Enterprise Data Studio, Enterprise Manager, Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, EWA, Financial Fusion, Financial Fusion Server, Gateway Manager, ImpactNow, InfoMaker, Information Anywhere, Information Everywhere, InformationConnect, InternetBuilder, iScript, Jaguar CTS, jConnect for JDBC, KnowledgeBase, MainframeConnect, Maintenance Express, MAP, MDI Access Server, MDI Database Gateway, media.splash, MetaWorks, MySupport, Net-Gateway, Net-Library, ObjectConnect, ObjectCycle, OmniConnect, OmniSQL Access Module, OmniSQL Toolkit, Open Client, Open ClientConnect, Open Client/Server, Open Client/Server Interfaces, Open Gateway, Open Server, Open ServerConnect, Open Solutions, Optima++, PB-Gen, PC APT Execute, PC DB-Net, PC Net Library, Power++, power.stop, PowerAMC, PowerBuilder, PowerBuilder Foundation Class Library, PowerDesigner, PowerDimensions, PowerDynamo, PowerJ, PowerScript, PowerSite, PowerSocket, Powersoft, PowerStage, PowerStudio, PowerTips, Powersoft Portfolio, Powersoft Professional, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, Report Workbench, Report-Execute, Replication Agent, Replication Driver, Replication Server, Replication Server Manager, Replication Toolkit, Resource Manager, RW-DisplayLib, RW-Library, S-Designer, SDF, Secure SQL Server, Secure SQL Toolset, Security Guardian, SKILS, smart.partners, smart.parts, smart.script, SQL Advantage, SQL Anywhere, SQL Anywhere Studio, SQL Code Checker, SQL Debug, SQL Edit, SQL Edit/TPU, SQL Everywhere, SQL Modeler, SQL Remote, SQL Server, SQL Server Manager, SQL SMART, SQL Toolset, SQL Server/CFT, SQL Server/DBM, SQL Server SNMP SubAgent, SQL Station, SQLJ, STEP, SupportNow, Sybase Central, Sybase Client/Server Interfaces, Sybase Financial Server, Sybase Gateways, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase User Workbench, SybaseWare, Syber Financial, SyberAssist, SyBooks, System 10, System 11, System XI (logo), SystemTools, Tabular Data Stream, Transact-SQL, Translation Toolkit, UNIBOM, Unilib, Uninull, Unisep, Unistring, URK Runtime Kit for UniCode, Viewer, Visual Components, VisualSpeller, VisualWriter, VQL, WarehouseArchitect, Warehouse Control Center, Warehouse Studio, Warehouse WORKS, Watcom, Watcom SQL, Watcom SQL Server, Web Deployment Kit, Web.PB, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, XA-Server and XP Server are trademarks of Sybase, Inc. 3/01

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

All other company and product names used herein may be trademarks or registered trademarks of their respective companies.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608.

Contents

About This Book	ix
------------------------------	-----------

PART 1 OVERVIEW

CHAPTER 1	About EJB Server	1
	About EJB Server.....	1
	Features	5
	The EJB Server execution engine	6
	Component support.....	7
	Network protocol support	8
	Administration and development tools	9
	Client-session and component-lifecycle management	10
	Naming services.....	12
	Connection caching.....	13
	Transaction management.....	13
	Thread-safety features	13
	Result-set support	14
	Permissions and roles	15
	PowerJ overview	15

CHAPTER 2	Getting Started	17
	Before you use EJB Server	17
	Terminology and concepts	18
	Terminology.....	18
	Concepts	19
	Developing an application	19
	The EJB Server runtime environment	20
	Basic tasks	21
	Using the Adaptive Server plug-in to Sybase Central	22
	Enabling EJB Server	23
	Disabling EJB Server	24
	Starting EJB Server automatically	25
	Starting EJB Server independently	25

Shutting down EJB Servers.....	26
Verifying the status of EJB Server.....	27

PART 2 INFORMATION FOR DEVELOPERS

CHAPTER 3	Enterprise JavaBeans Overview	31
	About Enterprise JavaBean components.....	32
	EJB component types	33
	EJB transaction attribute values.....	35
	EJB container services.....	37
	EJB support.....	38
	Running EJB components in EJB Server.....	38
	EJB clients connecting to EJB Server	39
CHAPTER 4	Creating Component-Based Applications.....	41
	Application architecture	42
	Designing the application	44
	Implementing components and clients.....	46
	Deploying the application	48
	Deploying components.....	48
	Developing clients	49
CHAPTER 5	Understanding Transactions and Component Lifecycles	51
	Component lifecycles	51
	The EJB Server transaction processing model	55
	How EJB Server transactions work.....	56
	Benefits of using EJB Server transactions	56
	Defining transactional semantics.....	57
	Example	63
	Dynamic enlistment in Bean-managed transactions	64
	OTS/XA transaction model.....	66
CHAPTER 6	Working with EJB Packages and Components	69
	Packages and Enterprise JavaBean components	69
	Importing Enterprise JavaBeans	71
	Installing components	74
	Modifying components	75
	Configuring component properties	75
	General component properties.....	76
	Transactions tab component properties	77
	Instances tab component properties	78

	Resources tab component properties	80
	Persistence tab component properties	82
	All Properties tab	84
	Generating stubs and skeletons	86
	Creating Enterprise JavaBeans	87
	Modifying packages	93
	Configuring package properties	94
	Exporting packages to EJB-JAR files.....	95
CHAPTER 7	Creating Enterprise JavaBean Clients	97
	Developing an EJB client	97
	Generating EJB stubs	98
	Java packages	99
	Compiling stubs	99
	Instantiating home interface proxies	100
	Obtaining an initial naming context	100
	Resolving Bean home names	103
	Instantiating remote interface proxies	104
	Calling remote interface methods	106
	Managing transactions	106
	Serializing and deserializing Bean proxies.....	107
CHAPTER 8	Managing Persistent Component State	109
	Persistence for entity Java Beans	109
	Using component-managed persistence	110
	Using automatic persistence	110
	Persistence for stateful components	114
	Using Java serialization.....	115
	Using automatic persistence	115
	Storage components	116
	Supported Java, IDL, and JDBC/SQL types	116
	Table schema for binary storage.....	117
CHAPTER 9	Developing Applications with PowerJ and EJB Server.....	119
	About the development process.....	119
	Creating workspaces, targets, and classes.....	122
	Designing the user interface.....	125
	Designing menus.....	126
	Accessing data	126
	Coding application logic	127
	Building distributed and Web applications that use EJB Server ..	128
	About EJB Server.....	128

Architecture of distributed and Web applications	129
Building EJB Server components with PowerJ	130
Building a Java client for a distributed or Web application	136
Building client/server applications using JDBC	137
Building the application	138
Building Enterprise JavaBeans 1.1 components.....	141

PART 3 INFORMATION FOR ADMINISTRATORS

CHAPTER 10	Configuring EJB Server	145
	Configuring an EJB Server.....	145
	General.....	146
	Log/Trace	147
	Naming Service	148
	All Properties	149
	Configuring server stack size	151
	Character sets.....	152
	Shared-memory connections	152
	Managing connection caches.....	153
	Creating and installing a new connection cache	153
	Modifying connection caches	154
	Modifying connection cache properties	154
	Connection cache refresh	157
	Connection cache ping.....	158
	Managing XA resources.....	159
	Setting up XA resources.....	159
	Creating XA resources	160
	Configuring Listeners	163
	Preconfigured listeners.....	163
	Configuring listeners.....	163
	Replacing an EJB Server	165
CHAPTER 11	EJB Server Naming Services.....	167
	How does the EJB Server naming service work?	167
	EJB Server initial context	168
	Name binding example.....	169
	Transient vs. persistent storage	170
	JNDI support	171
	JNDI J2EE features.....	171
	Configuring the EJB Server naming service	176
	Name binding password security	177
	Using an LDAP server with EJB Server	177

LDAP object schema and EJB Server objects 178
Storing EJB Server object bindings on an LDAP server 178

About This Book

This book describes how to create Enterprise JavaBean (EJB) clients and components for Sybase® EJB Server and Adaptive Server® Enterprise.

Audience

This book is intended for EJB component developers, Sybase System Administrators, and others interested in EJB components.

How to use this book

This book will assist you in creating EJB components and clients for Sybase EJB Server. It contains these parts and chapters:

- Part 1, “Overview,” provides a general description of the EJB Server and sufficient information to allow you to get started using it. Part 1 contains these chapters:
 - Chapter 1, “About EJB Server,” provides an overview of EJB Server, a summary of EJB Server features, and a description of Sybase PowerJ.
 - Chapter 2, “Getting Started,” describes basic concepts, terminology, and basic task information you need to use EJB Server
- Part 2, “Information for Developers,” describes EJBs and presents information about using the Adaptive Server plug-in for Sybase Central and PowerJ to create EJB clients and components.
 - Chapter 3, “Enterprise JavaBeans Overview,” describes Enterprise JavaBeans components.
 - Chapter 4, “Creating Component-Based Applications” describes the process of designing, building, and deploying applications with components executing in EJB Server.
 - Chapter 5, “Understanding Transactions and Component Lifecycles” explains the EJB Server component lifecycle and transaction processing models.
 - Chapter 6, “Working with EJB Packages and Components,” provides instructions for creating, deploying, and modifying EJB components and packages.

-
- Chapter 7, “Creating Enterprise JavaBean Clients,” describes how to implement EJB clients using the Sybase EJB client runtime.
 - Chapter 8, “Managing Persistent Component State” describes how to manage persistence for Enterprise JavaBeans.
 - Chapter 9, “Developing Applications with PowerJ and EJB Server,” gives an overview of how to develop applications using PowerJ and EJB Server
 - Part 3, “Information for Administrators,” describes how to set up and manage the EJB Server. It also describes the system procedures that support EJB Server.
 - Chapter 10, “Configuring EJB Server,” describes basic configuration tasks to customize your installation, such as creating new servers, changing server properties, and defining new connection caches
 - Chapter 11, “EJB Server Naming Services” describes how to use naming services to associate a logical name with an object.

Related documents

The following documents comprise the Sybase Adaptive Server Enterprise documentation:

- The release bulletin for your platform – contains last-minute information that was too late to be included in the books.

A more recent version of the release bulletin may be available on the World Wide Web. To check for critical product or document information that was added after the release of the product CD, use the Sybase Technical Library.

- The *Installation Guide* for your platform – describes installation, upgrade, and configuration procedures for all Adaptive Server and related Sybase products.
- *Configuring Adaptive Server Enterprise* for your platform – provides instructions for performing specific configuration tasks for Adaptive Server.
- *What’s New in Adaptive Server Enterprise?* – describes the new features in Adaptive Server version 12.5, the system changes added to support those features, and the changes that may affect your existing applications.

- *Transact-SQL User's Guide* – documents Transact-SQL, Sybase's enhanced version of the relational database language. This manual serves as a textbook for beginning users of the database management system. This manual also contains descriptions of the pubs2 and pubs3 sample databases.
- *System Administration Guide* – provides in-depth information about administering servers and databases. This manual includes instructions and guidelines for managing physical resources, security, user and system databases, and specifying character conversion, international language, and sort order settings.
- *Reference Manual* – contains detailed information about all Transact-SQL commands, functions, procedures, and datatypes. This manual also contains a list of the Transact-SQL reserved words and definitions of system tables.
- *Performance and Tuning Guide* – explains how to tune Adaptive Server for maximum performance. This manual includes information about database design issues that affect performance, query optimization, how to tune Adaptive Server for very large databases, disk and cache issues, and the effects of locking and cursors on performance.
- The *Utility Guide* – documents the Adaptive Server utility programs, such as isql and bcp, which are executed at the operating system level.
- The *Quick Reference Guide* – provides a comprehensive listing of the names and syntax for commands, functions, system procedures, extended system procedures, datatypes, and utilities in a pocket-sized book. Available only in print version.
- The *System Tables Diagram* – illustrates system tables and their entity relationships in a poster format. Available only in print version.
- *Error Messages and Troubleshooting Guide* – explains how to resolve frequently occurring error messages and describes solutions to system problems frequently encountered by users.
- *Component Integration Services User's Guide* – explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.
- *Java in Adaptive Server Enterprise* – describes how to install and use Java classes as datatypes, functions, and stored procedures in the Adaptive Server database.

-
- *Using Sybase Failover in a High Availability System* – provides instructions for using Sybase’s Failover to configure an Adaptive Server as a companion server in a high availability system.
 - *Using Adaptive Server Distributed Transaction Management Features* – explains how to configure, use, and troubleshoot Adaptive Server DTM features in distributed transaction processing environments.
 - *XA Interface Integration Guide for CICS, Encina, and TUXEDO* – provides instructions for using Sybase’s DTM XA interface with X/Open XA transaction managers.
 - *Glossary* – defines technical terms used in the Adaptive Server documentation.
 - *Sybase jConnect for JDBC Programmer’s Reference* – describes the jConnect for JDBC product and explains how to use it to access data stored in relational database management systems.
 - *Full-Text Search Specialty Data Store User’s Guide* – describes how to use the Full-Text Search feature with Verity to search Adaptive Server Enterprise data.
 - *Historical Server User’s Guide* – describes how to use Historical Server to obtain performance information for SQL Server and Adaptive Server.
 - *Monitor Server User’s Guide* – describes how to use Monitor Server to obtain performance statistics from SQL Server and Adaptive Server.
 - *Monitor Client Library Programmer’s Guide* – describes how to write Monitor Client Library applications that access Adaptive Server performance data.

Other sources of information

Use the Sybase Technical Library CD and the Technical Library Product Manuals Web site to learn more about your product:

- Technical Library CD contains product manuals and technical documents and is included with your software. The DynaText browser (included on the Technical Library CD) allows you to access technical information about your product in an easy-to-use format.

Refer to the *Technical Library Installation Guide* in your documentation package for instructions on installing and starting the Technical Library.

- Technical Library Product Manuals Web site is an HTML version of the Technical Library CD that you can access using a standard Web browser. In addition to product manuals, you'll find links to the Technical Documents Web site (formerly known as Tech Info Library), the Solved Cases page, and Sybase/Powersoft newsgroups.

To access the Technical Library Product Manuals Web site, go to Product Manuals at <http://www.sybase.com/support/manuals/>.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

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

- 1 Point your Web browser to Technical Documents at <http://www.sybase.com/support/techdocs/>.
- 2 Select a product from the product pick list and click Go.
- 3 Select the Certification Report filter, specify a time frame, and click Go.
- 4 Click a Certification Report title to display the report.

❖ **For the latest information on EBFs and Updates**

- 1 Point your Web browser to Technical Documents at <http://www.sybase.com/support/techdocs/>.
- 2 Select EBFs/Updates. Enter user name and password information, if prompted (for existing web accounts) or create a new account (a free service).
- 3 Specify a time frame and click Go.
- 4 Select a product.
- 5 Click an EBF/Update title to display the report.

❖ **To create 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.

Java syntax conventions

This book uses these font and syntax conventions for Java items:

-
- Classes, interfaces, methods, and packages are shown in Helvetica within paragraph text. For example:
 SybEventHandler interface
 setBinaryStream() method
 com.Sybase.jdbc package
 - Objects and parameter names are shown in italics. For example:
 “In the following example, *ctx* is a DirContext object.”
 “*eventHdler* is an instance of the SybEventHandler class that you implement.”
 “The *classes* parameter is a string that lists specific classes you want to debug.”
 - Java names are always case sensitive. For example, if a Java method name is shown as Misc.stripLeadingBlanks(), you must type the method name exactly as displayed.

Transact-SQL syntax conventions

This book uses the same font and syntax conventions for Transact-SQL as other Adaptive Server documents:

- Command names, command option names, utility names, utility flags, and other keywords are in Helvetica in paragraph text. For example:
 select command
 isql utility
 -f flag
- Variables, or words that stand for values that you fill in, are in italics. For example:
user_name
server_name
- Code fragments are shown in a monospace font. Variables in code fragments (that is, words that stand for values that you fill in) are italicized. For example:

```

Connection con = DriverManager.getConnection
("jdbc:sybase:Tds:host:port", props);

```
- You can disregard case when typing Transact-SQL keywords. For example, SELECT, Select, and select are the same.

Additional conventions for syntax statements in this manual are described in Table 1. Examples illustrating each convention can be found in the *System Administration Guide*.

Table 1: Syntax statement conventions

Key	Definition
{ }	Curly braces indicate that you choose at least one of the enclosed options. Do not include braces in your option.
[]	Brackets mean choosing one or more of the enclosed options is optional. Do not include brackets in your option.
()	Parentheses are to be typed as part of the command.
	The vertical bar means you may select only one of the options shown.
,	The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.

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.

Overview

This part provides an overview of the Enterprise JavaBeans Server (EJB Server) and the information you need to start using EJB Server.

About EJB Server

This chapter presents an overview of the Enterprise JavaBeans Server (EJB Server).

Topic	Page
About EJB Server	1
Features	5
The EJB Server execution engine	6
Component support	7
Network protocol support	8
Administration and development tools	9
Client-session and component-lifecycle management	10
Naming services	12
Connection caching	13
Transaction management	13
Thread-safety features	13
Result-set support	14
Permissions and roles	15
PowerJ overview	15

About EJB Server

Enterprise JavaBeans (EJB) Server is a component transaction server. It supports the EJB server-side component model for developing and deploying distributed, enterprise-level applications in a multi-tiered environment. It provides the framework for creating, deploying, and managing middle-tier business logic.

In a three-tier environment, the client provides the user interface logic, the business rules are separated to the middle tier, and the database is the information repository. The client does not access the database directly. Instead, the client makes a call to the EJB Server on the middle tier, which then accesses the database.

The three tiers can reside on different machines or on the same machines. EJB Server is designed to reside on the same machine as the database engines it serves. Because the servers are on the same machine, EJB Server can communicate with the database using Adaptive Server's high-speed, shared-memory JDBC driver. This approach ensures:

- High-speed communication and data transfer, even for large data sets
- Secure data transmission because the transfer of information from the third tier to the middle tier does not take place over the network

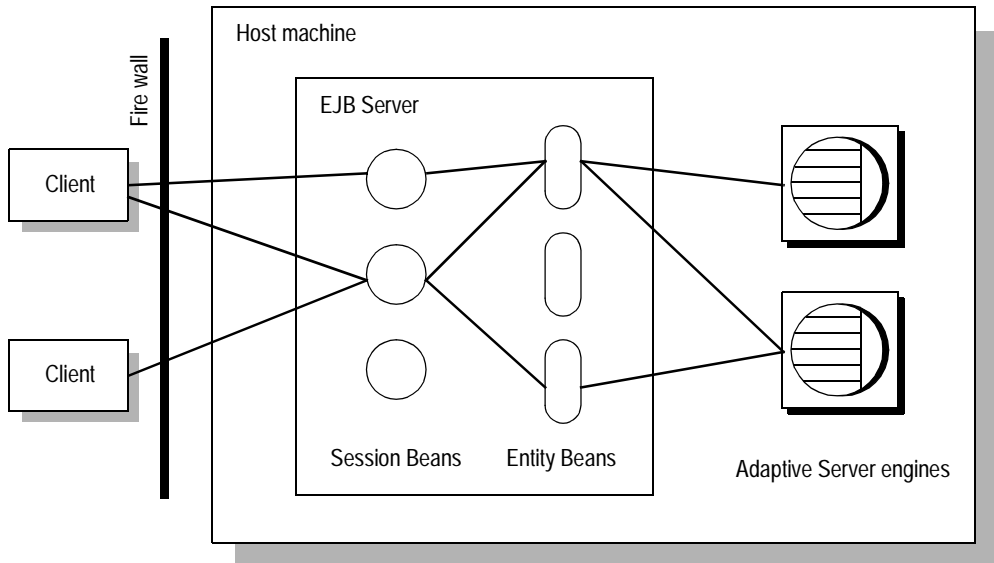
EJB components (or Beans) are reusable modules of code that combine related tasks (methods) into a well-defined interface. EJB components contain the methods that execute business logic and access data sources. You (or the Administrator) install the component's executable code on EJB Server. Any number of independent Java or EJB applications (clients) can use the EJBs.

There are three types of **Enterprise JavaBeans: stateful session Beans, stateless session Beans, and entity Beans**. Each type of bean is a set of methods and is responsible for different tasks on behalf of the client.

All session Bean instances are transient. They maintain a one-to-one relationship with the client. They perform tasks, and can store information in the database on the client's behalf. Stateful session Beans manage complex tasks that require the accumulation of data. Stateless session Beans manage tasks that do not store data between method calls. Entity Bean instances are persistent. They represent underlying objects, typically a particular row in a database. All three bean types work together to process a request and return information to the client.

Figure 1-1 shows how the client interacts with the EJB Server and the database.

Figure 1-1: EJB Server environment



The **stub** and the **skeleton** allow EJB Server to appear to run locally on the client. Every component instance has its own stub and skeleton created specifically for it. The stub resides on the client machine and is connected over the network to the skeleton, which resides on EJB Server. The stub acts as a surrogate for the client, transmitting requests to the skeleton. The skeleton listens on an IIOP port for requests from the stub.

When the skeleton receives a request, it determines which method is required and then invokes that method. Using the Sybase high-speed JDBC driver, EJB Server sends the request to Adaptive Server. If values are returned, the skeleton sends them to the stub, which returns them to the client application.

EJB Server provides efficient management of client sessions, threads, third-tier database connections, and transaction flow, without requiring specialized knowledge on the part of the component developer. As a consequence, developers can focus on solving business problems instead of programming the application's infrastructure.

Developers use classes and interfaces from the `javax.ejb` packages of the JavaSoft API to create and deploy components. To implement a component, the developer must define interfaces and classes:

- **Remote interface** – defines the Bean’s business methods and extends `javax.ejb.EJBObject`.
- **Home interface** – defines the Bean’s lifecycle methods and extends `javax.ejb.EJBHome`.
- **Bean class** – implements the Bean’s business methods and extends `javax.ejb.EnterpriseBean`.
- **Primary key** – provides a pointer into the Adaptive Server database and must implement `Serializable`. Necessary for entity Beans only.

Sybase provides a graphics-based management tool, the Adaptive Server plug-in to Sybase Central, for EJB Server developers and administrators. From this graphical interface, developers can deploy components and administrators can configure the server.

- For detailed information about creating and implementing EJB clients and components, see Chapter 6, “Working with EJB Packages and Components.”
- For detailed information about configuring EJB Server, see Chapter 2, “Getting Started,” and Chapter 10, “Configuring EJB Server.”

Feature summary

EJB Server features include the following:

- A scalable, multithreaded, platform-independent execution engine
- Dispatch and stub/proxy support for the EJB component model
- High-speed communication through Adaptive Server shared memory
- Graphical administration with the Adaptive Server plug-in to Sybase Central
- Easy integration with Sybase PowerJ development environment
- Transparent client-session and component lifecycle management
- Connection caching to allow reuse of database connections
- Industry-standard naming services to resolve components using logical names rather than server addresses
- Transaction management to simplify the design and implementation of an application’s transactions
- Transparent thread-safety features to simplify use of shared data and resources

- Result-set support to enable efficient retrieval of tabular data in client applications
- Support for Enterprise JavaBeans (EJB) components developed according to version 1.1 of the Enterprise JavaBeans specification.

The following sections explain these features and describe how EJB Server works.

Features

EJB Server is for deploying transaction-intensive business applications on the Internet. These applications move beyond one-way dynamic updates or data collection to real-time two-way updates of business critical information. You can also migrate traditional client/server transactional applications to multitier EJB Server applications.

EJB Server provides a framework for deploying the middle-tier logic of distributed component-based applications. EJB Server's high-performance transaction server provides efficient management of client sessions, threads, database connections, and transaction flow. EJB Server's scalability and platform independence allow you to develop your application on inexpensive uniprocessor machines, then deploy the application on an enterprise-grade multiprocessor server.

Client-side logic for enterprise applications must be as small and efficient as possible to conserve network bandwidth. To accomplish this goal, applications are partitioned into three parts: presentation logic, business logic, and database logic. The database resides on the bottom tier of the enterprise system to maintain and secure the organization's information assets. The business logic resides in the middle tier. The presentation logic is on the user's desktop, or top tier, or is dynamically downloaded to the user's desktop.

The EJB Server is responsible for executing and securing the vast majority of a corporation's business logic. This makes it a critical component in the emerging network-centric architecture. The Web browser connects to EJB Server or a Web server via HTTP to download an HTML page containing a Java applet that performs presentation functionality. The applet communicates with EJB Server, calling middle-tier components that perform business logic. Adaptive Server stores, processes, and protects the corporate data. EJB Server manages a pool of connections to the back-end database, coordinating the transaction processing to those servers.

Components are objects that reside on EJB Server and can be used by many different programs, regardless of the program's programming language. A client executes the methods in a component. Instead of creating one massive program, you create a client that contains the GUI and validation code and several individual components that contain the functionality (or business logic) of your program. By separating the functionality from the GUI, you can easily upgrade and change the functionality of your program without having to change the GUI. In addition, multiple clients can use components at the same time.

The EJB Server execution engine

EJB Server's runtime engine provides a scalable and platform-independent environment for the execution of component-based applications. EJB Server is scalable because it is multithreaded and multiprocessor safe. The EJB Server execution environment is the same across all platforms.

The EJB Server runtime engine provides these services:

- Network listeners for the connections on which clients send remote component invocations. EJB Server's core network server technology is based on Sybase's Open Client/Server™ technology.
- An execution environment for middle-tier components.
See "Server-side component support" on page 7.
- A built-in HTTP server. You can use EJB Server's HTTP support to deploy your application's Java applets and HTML pages.
- Ability to run with different Java virtual machines.
- Connection caching. You can define caches of connections for interacting with databases from EJB Server components.

See "Connection caching" on page 13 for more information.

In addition to these built-in services, you can install *service components* that run in the background and provide customized services to clients or other components.

Component support

Components are reusable modules of code that combine related tasks (methods) into a well-defined interface. EJB Server components are installed on an EJB Server and contain the methods that execute business logic and access data sources. You or your administrator install the component's executable code on the EJB Server. Components can be distributed to Adaptive Server databases residing on the same host. Once installed, components can be used by any number of independent applications.

Since EJB Server components reside on the server, components do not contain methods to display graphics or user interfaces—that is, EJB Server components are inherently nonvisual.

User-interface developers or other component developers can browse a component's interface in the Adaptive Server plug-in for Sybase Central; in their code, they use a client stub or proxy to invoke the component's methods. The stub or proxy acts as a local surrogate for the remote component, providing the same method signatures as the component and hiding the details of server communication.

EJB Server's server-side component support and client-side stub or proxy support are independent. Any EJB Server client can execute any component. Additionally, since EJB Server uses standard CORBA IIOP as its core network protocol, you can use CORBA client runtimes from other vendors to invoke components installed on an EJB Server.

All clients and components share a common interface repository. Component interfaces are stored in standard CORBA Interface Definition Language (IDL). Interfaces can be defined by importing compiled Java classes or standard-format EJB-JAR files.

Server-side component support

EJB Server supports Java components that follow the JavaSoft Enterprise JavaBeans (EJB) specification, version 1.1. An Enterprise JavaBean is a nonvisual, transactional component that is implemented in Java.

Chapter 6, “Working with EJB Packages and Components,” describes how to create EJB components.

Client stub/proxy support

Applications invoke an EJB Server component using a stub or proxy object. The stub or proxy acts as a local surrogate for the remote component; it provides the same method signatures as the component and hides the details of server communication. Stubs and proxies are available for:

- **Java (EJB)** Any component can be invoked via a Java stub class. The Adaptive Server plug-in generates source code for Java stubs. At runtime, your client program instantiates the stub. When you call methods on the stub class, the stub transparently invokes the component method on the EJB Server. Using HTML pages, Java applets, and EJB Server's built-in HTTP support, you can create "zero-install" applications that have no client-machine installation requirements other than the presence of a Java-capable Web browser.

EJB Server supports the EJB Java client model.

- **EJB** Your program uses the JavaSoft EJB (javax.ejb) classes and EJB Server's EJB stubs to call EJB Server component methods. This client model follows the EJB 1.1 Specification.

Chapter 7, "Creating Enterprise JavaBean Clients," describes how to implement EJB clients.

Network protocol support

EJB Server supports the following protocols:

- **Internet Inter-ORB Protocol (IIOP)** IIOP is the standard protocol for communication between CORBA ORBs over TCP/IP networks. The EJB client model uses IIOP. IIOP connections can also be tunneled inside of HTTP to allow connections through firewalls that do not allow passage of IIOP traffic, as discussed in "HTTP tunneling support" on page 9.
- **Hypertext Transfer Protocol (HTTP)** HTTP is used by Web browsers for file downloads and uploads. EJB Server provides HTTP support to allow you to deploy HTML pages and Java applets on the EJB Server itself.

To enable support for IIOP, you must define a *listener* in the Adaptive Server plug-in. The listener configuration specifies a server address (host name and port number) as well as the network protocol and security settings to be used by clients that connect to that listener.

To enable support for HTTP, you must use the standard configuration and port number.

HTTP tunneling support

Almost all network firewalls allow HTTP traffic to pass, but some reject IIOP packets. When IIOP traffic is tunnelled inside of HTTP, your clients can connect to the EJB Server through a firewall that does not allow IIOP traffic to pass.

EJB Server's Java client ORB performs HTTP tunnelling automatically using the designated IIOP port. No additional configuration or proxies are required. When connecting, the EJB Server client-side ORB first tries to open an IIOP connection to the specified address and port. If the IIOP connection fails, the ORB tries an HTTP-tunnelled connection to the same address and port. The default behavior 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 HTTP tunnelling is always required for a Java client, you can set the `ORBHttp` property to cause the ORB to use HTTP tunnelling without trying plain IIOP connections first..

Administration and development tools

Sybase Central is a common management framework for Sybase application and database servers. EJB Server provides the Adaptive Server plug-in to Sybase Central plug-in for developers and administrators.

The Adaptive Server plug-in provides graphical administration facilities for EJB Server, including support for development and deployment.

Development support

You can use Sybase PowerJ 3.6 with EJB Server. Using this IDE tool, you can develop, deploy, and debug EJB Server components entirely within the development environment. You can also generate the proxies required for client application development. For more information, see *Building Internet and Enterprise Applications* in the *Enterprise Application Studio* online books collection.

Interface definitions can be imported from existing Java classes or from standard CORBA IDL files.

The Adaptive Server plug-in also generates stub classes for use in Java client applications.

- Deployment support
- To simplify application deployment, the Adaptive Server plug-in defines the following basic, middle-tier application units:
- **Servers** A server represents one EJB Server runtime process. Each server has its own network addresses for client session connections and for HTTP (HTML) connections. All servers on one host machine share the same configuration repository. For administration purposes, you can connect to any server on the host machine to configure other servers on the same host.
 - **Packages** A package organizes components into cohesive, secure units that can be easily deployed on another EJB Server. Packages can be exported, or saved, as a Java archive (JAR) file. The package archive includes the definition of all components in a package, plus any supporting files (such as source code and client files) that you specify. Package archives exported from one server can easily be imported for redeployment on another server.
-
- Note**
Do not confuse EJB Server package names with Java package names.
-
- **Components** A component definition consists of the component's method signatures and other properties, such as component type, transaction support, threading model, and the name of the Java class or executable library that implements the component.
- Before a client application can execute a component, the component must be installed in an EJB Server package, and that package must be installed in the server to which the client connects.
- Hot refresh support
- EJB Server provides a Refresh menu item to refresh components, packages, and servers. This option lets you test and debug component implementation changes without restarting EJB Server.

Client-session and component-lifecycle management

EJB Server client sessions are established internally by the client stubs and proxies that applications use to invoke EJB Server component methods. A component's lifecycle determines how instances are allocated, bound to client sessions, and destroyed. EJB Server manages both client sessions and component lifecycles without requiring specialized knowledge on the part of the application developer.

Client-session management

Internally, the stub or proxy object establishes a network connection between the EJB Server and a remote client. The stub/proxy model discussed in “Client stub/proxy support” on page 8 requires user-authentication parameters to instantiate a stub or proxy object. The communication protocol is also determined when the stub or proxy object is instantiated. Once the stub or proxy object exists, all details of network communication are hidden from the application developer.

All stubs and proxies use the Inter-ORB Invocation Protocol (IIOP) to communicate with the EJB Server. See “Network protocol support” on page 8 discusses client protocols in detail.

For more information on stub and proxy objects, see Chapter 7, “Creating Enterprise JavaBean Clients.”

Component-lifecycle management

In the simplest case of lifecycle management, an instance is allocated for each stub or proxy created by the client and is destroyed when the client explicitly requests destruction or when it disconnects, whichever happens first.

More sophisticated components can be coded to support instance pooling. Instance pooling allows EJB Server to maintain a cache of component instances and bind them to client sessions on an as-needed basis. Instance pooling requires the following changes to your component:

- The component must provide activate and deactivate methods. EJB Server calls the activate method just before an instance is bound to a client session. activate must be able to reset the component to an as-allocated state. EJB Server calls deactivate just before an instance is unbound from a client session (that is, made idle again).
- Methods in the component must use the EJB Server transaction state primitives to request early deactivation.

For components that support EJB Server transactions, the time between EJB Server’s activate and deactivate calls coincides with the beginning and end of that instance’s participation in an EJB Server transaction.

Using components that support instance pooling increases the scalability of your application. Instance pooling eliminates execution time and memory consumption that would otherwise be spent allocating unnecessary component instances.

Coded character set conversions

EJB Server supports multiple coded character sets for clients and components. When a client and component use different coded character sets, the EJB Server automatically converts character data from one character set to another. For example, if the client uses the `roman8` character set and the component uses `iso_1`, EJB Server converts string parameters and return values automatically from `roman8` to `iso_1` when the client calls the component methods.

In accordance with the Java standards, Java components use 16-bit Unicode. Unicode contains mappings for all characters in all other known coded character sets.

Note EJB Server and its host Adaptive Server must use the same character set. If you change the character set on Adaptive Server, you must perform a similar change on EJB Server. See “Configuring an EJB Server” on page 145.

Naming services

When multiple servers are involved in your application, the naming service allows you to specify logical server names rather than server addresses. For example, instead of connecting to your finance component server at host `badger` using port 9000, you can specify the initial naming context for that server, such as `USA/MyCompany/FinanceServer`. Components are identified by specifying an initial server name context plus the package and component name. For example:

```
USA/MyCompany/FinanceServer/FinancePackage/PayrollAdmin
```

This layer of abstraction allows you to move a server to another host without affecting deployed client applications. Naming does require that one EJB Server use a well-known, stable host and port. This server acts as the name server for other servers that participate in your application, and clients connect to that server to resolve name requests.

You have the option of using persistent or transient storage for the naming database. For transient storage, the EJB Server builds the name database in memory when it starts, based on the contents of the EJB Server configuration repository.

Configuring naming services

Naming configuration for a multi-server application is briefly summarized as follows:

- 1 Choose one EJB Server to act as name server for the application. You can configure this server to store names in memory (transient storage), or to store names in a third-party directory server (persistent storage).
- 2 Configure each of the remaining servers to connect to the designated EJB Server naming server to resolve names. Each server will also update the name space when packages and components are added or deleted in the Adaptive Server plug-in.

Client APIs for naming

EJB Server provides industry standard client-side APIs for naming services. EJB Server also provides implementations of the CORBA standard CosNaming API and the Java Naming and Directory (JNDI) API.

See Chapter 7, “Creating Enterprise JavaBean Clients,” for more information.

Connection caching

Connection caching allows EJB Server components to share pools of preallocated connections to the database server, avoiding the overhead imposed when each instance of a component creates a separate connection. Components that support transactions must use a connection from an EJB Server connection cache to interact with the database.

Transaction management

EJB Server’s transaction management feature allows you to specify a transaction coordinator and define a component’s transactional semantics as part of the component interface.

Thread-safety features

Since EJB Server is a multithreaded environment, component instances that share resources and volatile data must be coded or configured to avoid problems with inconsistent state. For example, if all instances of a component write to the same file, you must take steps to ensure that the file is “locked” before each instance writes to it and “unlocked” when the write completes. If writes to the file are allowed to occur simultaneously, then output from two component instances may be mixed together within the file.

Whenever possible, avoid the use of static class variables. Also, avoid sharing stateful (being able to maintain information about the state of a resource) resources such as database connections or file descriptors. In cases where data and resources are shared, there are two ways to ensure thread safety in a component:

- Configure the component for single-threaded execution.

Each component defined in EJB Server has a Concurrency property. By default, components are multithreaded and instances are allowed to execute concurrently on different threads. You can also request single-threading; in a single-threaded component, each method invocation blocks other method invocations on instances of the same component.

For components in general, single-threading is the least desirable alternative because it increases the likelihood that clients will block each other's execution and increase the apparent response time of client applications. Single-threading makes sense for some specific problems; for example, to share an output file among component instances, you can create a single-threaded component with methods that write to the file (another alternative is to use explicit threading primitives when implementing code that writes to the file, such as the Java synchronized keyword).

- Store shared data on the database.

You can use connection caching and store the data on the database, letting the database server handle concurrency issues. (The component's transactional semantics may affect the interaction with the database.)

Result-set support

EJB Server methods can return tabular data to the calling client. This feature can be useful for the following reasons:

- **Use with data-aware controls** Some front-end tools provide objects that can automatically display a result set. For example, in a PowerJ application, you can pass the result set obtained from an EJB Server method invocation to a PowerJ Query object or DataWindow, and it will display the rows.
- **Efficiency** For tasks that require returning tabular data, using an EJB Server result set is the most efficient alternative. Common uses of result sets include menu and pick-list population. For example, in an online clothing catalog, you need to list in-stock sizes for each item.

The EJB Server result set allows data to be sent all at once (rather than requiring a get-next-row method and one client-server round trip per method). A large EJB Server result set can be sent with less overhead than is required to encapsulate tabular data as an object and send a serialized version of the object to the client.

Each component model provides an interface that allows you to define result sets from scratch or to forward results from a database query directly to the client.

Permissions and roles

EJB Server and its host Adaptive Server share user names and roles. If you create a user or a role in Adaptive Server, that user and role are also valid in EJB Server. To use the Adaptive Server plug-in with EJB Server, you must be a registered user in Adaptive Server and have the System Administrator role.

System Administrator roleSecurity for components is handled at the method level for each package. You include a J2EE role in the method you want to restrict, and then map that J2EE role to a role in Adaptive Server so that only users with that role can execute the method. The Adaptive Server plug-in allows you to map roles through each package for methods within the package. See “Configuring package properties” on page 94 for more information.

PowerJ overview

PowerJ is Sybase’s RAD tool for Java development. Even if you are not an experienced Java programmer, you can use PowerJ to write sophisticated programs. If you *are* an experienced Java programmer, you can use the full facilities of Java whenever necessary.

Because PowerJ provides facilities that let you build and deploy components directly to the middle tier, it is the ideal web development environment for EJB Server.

With PowerJ, you can:

- Write Java code that takes advantage of the PowerJ component library, a collection of Java classes that speeds the development process
- Use object-oriented programming features such as inheritance, encapsulation, and polymorphism to make your objects more reusable

- Create Java components that can be deployed directly to EJB Server
- Deploy Java nonvisual objects to the Sybase Java VM running in an Adaptive Server database
- Retrieve and update database information using Sybase's patented DataWindow technology
- Access a variety of industry-standard databases through JDBC, the Java standard for database access
- Build windows, menus, and other user interface components using visual programming tools

Getting Started

This chapter provides basic concepts, terminology, and the task information you need to get started using EJB Server.

Topic	Page
Before you use EJB Server	17
Terminology and concepts	18
Basic tasks	25

Note Your installation must have a valid license for Adaptive Server-EJB Server to use this product.

You can perform all the tasks necessary to use EJB Server from the Sybase Central, the graphics based management tool for Sybase products. This chapter presents a conceptual overview of the processes you will use to develop EJB Server applications. Refer to Chapter 6, “Working with EJB Packages and Components,” to create EJB component applications.

The installation process starts and preconfigures your Adaptive Server host, your EJB server, and Sybase Central. This chapter provides directions for enabling the EJB Server option, starting and stopping EJB Server, and setting up the Adaptive Server plug-in to Sybase Central. Refer to Chapter 10, “Configuring EJB Server,” for other configuration tasks.

Before you use EJB Server

To use EJB Server effectively, you should be able to create programs in the Java/Enterprise JavaBeans programming language and component model.

You should also know how to retrieve and update information in databases and be familiar with component technology concepts.

Terminology and concepts

This section explains some of the basic concepts and terminology associated with developing component-based EJB Server applications. It is intended primarily to provide you with enough information to begin using the Adaptive Server plug-in for Sybase Central. For detailed information on EJB application development, refer to Chapter 6, “Working with EJB Packages and Components.”

Terminology

An EJB application consists of one or more packages and a client application or applet. Packages consist of components, and components are made up of one or more methods.

- EJB Server hosts, manages, and executes JavaBean components. In the EJB environment, a **component** is simply an **application object** that consists of one or more methods. JavaBeans typically execute business logic, access data sources, and return results to the client. Clients (applets) create an instance of a component and execute methods associated with that component. Components run strictly within the EJB Server.
- A **package** is a collection of components that work together to provide a service or some aspect of your application’s business logic. A package defines a boundary of trust within which components can easily communicate. Each package acts as a unit of distribution, grouping together application resources for ease of deployment and management.
- A **stub** is a Java class stub generated by the Adaptive Server plug-in for Sybase Central and acts as a **proxy object** for an EJB component. A stub is compiled and linked with your Java applets or client application. A stub communicates with EJB Server to instantiate and invoke a method on a component in the middle tier. Stubs make a remote EJB component appear local to the client.
- A **skeleton** acts as the interface between the EJB runtime environment and the user code that implements the method. Skeletons are compiled and linked with each of the components, and at runtime they enable EJB Server to locate and invoke an appropriate method.

- EJB Server transparently maintains a **session** between a client application and the EJB Server. Unlike a typical HTTP scenario, where a new connection is created for each request and response, sessions allow a browser to maintain a connection with the server across a multiple request-response cycle.

Concepts

EJB Server implements distributed computing architecture. In this model, three sets of elements work together to give users access to data:

- Client-side applet or application
- EJB Server components
- Adaptive Server database

Java applets are downloaded to clients, which instantiate components on the server. Client applications are installed on client machines, from which they also instantiate components on the server.

An applet or application manages presentation and interaction with an end user. Components running on EJB Server handle much of the application processing. The database stores, manages, and processes the data.

If the client is an applet, users find and launch applications from traditional HTML pages. Instead of simply loading a static page, EJB Server downloads an executable applet to the individual's browser. If the client is an already-installed application, the user launches the application from his or her machine. Clients communicate directly with an application component running on EJB Server. Server components access data from one or more databases, apply business logic, and return results to the client applet for display.

When a proxy object is created on the client applet, it instantiates a corresponding component registered with the EJB Server. On the server side, a component is instantiated in response to a request from the proxy object running in the client environment. A method on a component is executed when it is invoked by a proxy object on the client applet.

Developing an application

There are three basic steps involved in creating and deploying an EJB Server application that employs a Java applet as a client.

❖ **To create and deploy an EJB Server application:**

- 1 Define packages, components, and methods. The Adaptive Server plug-in for Sybase Central is the EJB Server GUI interface. It allows you to easily define the packages, components, and methods that EJB Server clients use to run an application. The Adaptive Server plug-in for Sybase Central generates:

- **The client-side stub files** Stubs contain interface information used by the client to invoke EJB component methods.
- **The server-side skeleton files** Skeletons provide the interface information of each component method.

- 2 Create the applets and components. Once you have generated the stubs and skeletons, write the Java classes that, once linked with the stub files, form the basis of your downloadable applet.

In addition to the applet, you need to develop the server-side components that link with the skeletons to form the business logic of your servlet. EJB Server supports many of the integrated development environment (IDE) tools, such as PowerJ, available today.

- 3 Deploy the application. You register components on the EJB Server. Since EJB Server is also a Web server, you can write an HTML page for your applet and install it on EJB Server.

The EJB Server runtime environment

A typical EJB Server application has an applet or HTML page associated with it. Once you build and deploy such an application, it runs in the following fashion:

- 1 EJB Server receives an HTTP request and downloads the requested HTML page or applet. Included with the applet are the Java stubs, which through a proxy, instantiate components and invoke the methods on those components.
- 2 The client establishes a session with EJB Server. The session, unlike an HTTP connection, allows the client and EJB Server to maintain a connection throughout the transaction.

- 3 The client creates a component instance through a client-side proxy. The proxy used depends on the type of component being instantiated. EJB Server validates the user against the component's access list. If the user is validated, the dispatcher checks the location and status of the component and creates an instance.
- 4 The client invokes the component's business logic by executing its methods.
- 5 The component may interact with the database server. If it does, the component obtains a connection to the database using the Sybase high-speed JDBC driver.
- 6 EJB Server returns the results from the database to the client.
- 7 The client indicates that it has completed the operation. EJB Server destroys the component instance or returns it to a pool for future client instantiations. The client disconnects from EJB Server.

Basic tasks

The installation and configuration process will preconfigure and start both your Adaptive Server Enterprise and its EJB Server. As part of the post-installation process, you will start up Sybase Central, the graphical user interface that allows you to manage both Adaptive Server and EJB Server.

You can use EJB Server without further configuration, but to customize EJB Server for your installation, see Chapter 10, "Configuring EJB Server".

The following sections describe enabling and starting EJB Server and the Adaptive Server plug-in for Sybase Central so that you can perform these basic tasks yourself after installation.

The installation and configuration process will start Adaptive Server, enable the EJB Server option, start EJB Server, and start the Adaptive Server plug-in to Sybase Central.

Using the Adaptive Server plug-in to Sybase Central

The Adaptive Server plug-in runs within Sybase Central. Use the Adaptive Server plug-in for Sybase Central to configure EJB Server and to define and deploy software components and packages. An EJB Server must be running before Sybase Central can connect to it.

You must have the System Administrator role in Adaptive Server to use the Adaptive Server plug-in. The installation process creates this role.

EJB Server is preconfigured so that you can start up and run the server. You may need to configure EJB Server further to run your applications.

What you can do from the Adaptive Server plug-in

From the Adaptive Server plug-in you can:

- Replace EJB servers
- Shutdown and restart EJB Servers
- Add or drop connection caches
- Add, drop, export, or deploy packages
- Add or drop components
- Update object properties
- Generate stubs and skeletons for components

Each of these tasks is described in subsequent chapters.

Starting the Adaptive Server plug-in

First start Sybase Central and then start the Adaptive Server plug-in from within it.

Sybase Central

You can start Sybase Central from the UNIX command line or from a Sybase Central shortcut on the desktop.

❖ To start Sybase Central from the UNIX command line:

1 Enter:

```
source $SYBASE/SYBASE.csh
```

2 Then enter:

```
$SYBASE/sybcent32/java/scjview
```


The Adaptive Server plug-in

❖ **To start Sybase Central from Windows NT:**

- Select the Sybase Central Java Edition shortcut from the desktop. The installation process creates this shortcut for you.

Once the Sybase Central Java version is running, you can start the Adaptive Server plug-in.

❖ **To connect to the Adaptive Server host:**

- 1 Select Tools | Connect.
The Connect to Adaptive Server Enterprise screen displays.
- 2 Enter the sa user name and password.
- 3 Select the Adaptive Server host machine name or IP address.
- 4 Verify the Adaptive Server host port number.
- 5 Click Connect.

Disconnecting from the Adaptive Server host

dSybase Central allows you to disconnect the Adaptive Server plug-in from an Adaptive Server host so that you can connect to another server, or reconnect to the same server, without restarting the plug-in.

❖ **To disconnect from the Adaptive Server host:**

- 1 Highlight the Adaptive Server host.
- 2 Select File | Disconnect.

Enabling EJB Server

The installation process enables the EJB Server option in Adaptive Server. Later on, you can enable or disable an EJB Server in either of two ways:

- From the Adaptive Server plug-in to Sybase Central
- From the command line using isql

Enabling EJB Server is a dynamic process. You do not need to restart Adaptive Server for it to take effect. After you enable EJB Server, select File | Refresh All to display the Enterprise JavaBeans folder beneath the host Adaptive Server.

- ❖ **To enable EJB Server from the Adaptive Server plug-in:**
 - 1 Highlight the Adaptive Server host for the EJB Server.
 - 2 Select File | Configure.
 - 3 Select “enable enterprise java beans” from the alphabetical list of configuration parameters.
 - 4 Change the number in the Value column to 1.
 - 5 Click OK.

- ❖ **To enable EJB Server using *isql*:**
 - 1 Log in to Adaptive Server using *isql*.
 - 2 Enter:

```
sp_configure 'enable enterprise java beans', 1
```

Disabling EJB Server

You can disable the EJB Server option in either of two ways:

- From the Adaptive Server plug-in to Sybase Central
- From the command line using *isql*

Disabling EJB Server is a dynamic process. You do not need to restart Adaptive Server for it to take effect. After you disable EJB Server, select File | Refresh All to remove the Enterprise JavaBeans folder beneath the host Adaptive Server.

- ❖ **To disable EJB Server from the Adaptive Server plug-in:**
 - 1 Highlight the Adaptive Server host for the EJB Server.
 - 2 Select File | Configure.
 - 3 Select “enable enterprise java beans” from the alphabetical list of configuration parameters.
 - 4 Change the number in the Value column to 0.
 - 5 Click OK.

❖ To disable EJB Server using *isql*:

1 Log in to Adaptive Server using *isql*.

2 Enter:

```
sp_configure 'enable enterprise java beans', 0
```

Starting EJB Server automatically

When the installation process is complete:

- EJB Server is running.
- EJB Server is configured to start up automatically each time Adaptive Server starts up.

Later on, you can enable or disable automatic startup using the `sp_serveroption` system procedure.

For example, to disable automatic startup, enter:

```
sp_serveroption 'SYB_EJB',  
  'external engine auto start', 'false'
```

where `SYB_EJB` is the logical name of the EJB Server.

To enable automatic startup, enter:

```
sp_serveroption 'SYB_EJB',  
  'external engine auto start', 'true'
```

Starting EJB Server independently

Note Adaptive Server must be running before you can start EJB Server.

You can start or restart EJB Server in two ways:

- From the Adaptive Server plug-in to Sybase Central.
- Using the `sp_extengine` system procedure.

❖ To restart EJB Server from the Adaptive Server plug-in

1 Right-click on the EJB Server you want to restart.

2 Choose File | Restart.

- 3 Press View | Refresh Folder.

Note Restarting EJB Server may take a minute or two, depending on the load on Adaptive Server.

You can stop and then restart EJB Server from the Adaptive Server plug-in. To start an EJB Server that has been shut down in another way, you must use the `sp_extengine` system procedure.

❖ **To start EJB Server using the *sp_extengine* system procedure**

- 1 Log in to Adaptive Server using `isql`.
- 2 Enter this command:

```
sp_extengine 'SYB_EJB', 'start'
```

where SYB_EJB is the logical name of the EJB Server.

Shutting down EJB Servers

You can stop EJB Server in three ways:

- From the Adaptive Server plug-in
- Using the `sp_extengine` system procedure
- By shutting down Adaptive Server Enterprise

❖ **To shut down EJB Server from the Adaptive Server plug-in**

- 1 Highlight the EJB Server you want to shut down.
- 2 Press File | Stop EJB

❖ **To shut down EJB Server using *sp_extengine***

- 1 Log in to Adaptive Server using `isql`.
- 2 Enter this command:

```
sp_extengine 'SYB_EJB', 'stop'
```

where SYB_EJB is the logical name of the server.

❖ **To shut down both EJB Server and Adaptive Server**

- 1 Log in to Adaptive Server using `isql`.
- 2 Enter this command:

shutdown

You can also use shutdown with the no wait option.

Note Issuing a “kill -9” command on Adaptive Server will not shut down the associated EJB Server.

Verifying the status of EJB Server

To determine if EJB Server is running, use the `sp_extengine` system procedure.

Log in to Adaptive Server using `isql` and enter:

```
sp_extengine 'SYB_EJB', 'status'
```

where `SYB_EJB` is the logical name of the EJB Server.

PART 2

Information for Developers

This part provides an overview of Enterprise JavaBeans (EJBs) and information about using the Adaptive Server plug-in for Sybase Central and PowerJ to create EJB clients and components.

Enterprise JavaBeans Overview

EJB Server supports Enterprise JavaBean (EJB) 1.1 components.

For details on EJB architecture, see the EJB 1.1 specifications from Sun Microsystems at <http://java.sun.com/products/ejb/>.

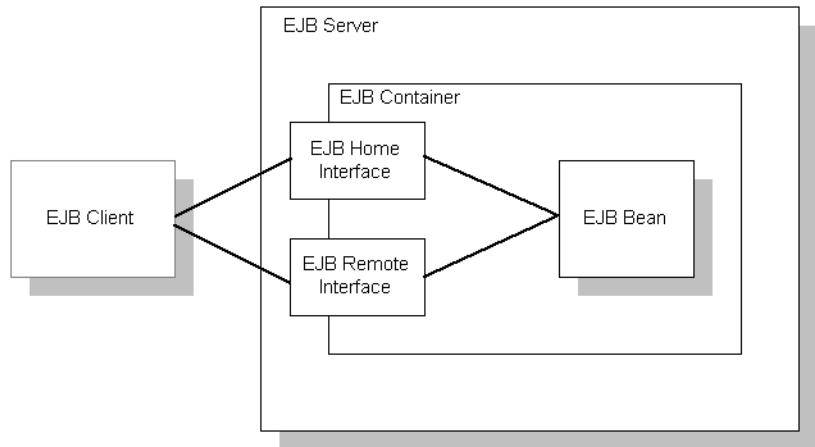
Topic	Page
About Enterprise JavaBean components	32
EJB support	38

About Enterprise JavaBean components

The 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.

The EJB architecture looks like this:



EJB server EJB Server holds the EJB container, which provides the services required by the EJB component.

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. An EJB component can act as a EJB client by calling methods in another EJB component.

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.

EJB container The EJB specification defines a container as the environment in which one or more EJB components execute. 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. In EJB Server, the container encapsulates:

- The client runtime and generated stub classes, which allow clients to execute components on a remote server as if they were local objects.
- The naming service, which allows clients to instantiate components by name, and components to obtain resources such as database connections by name.
- The EJB Server component dispatcher, which executes the component's implementation class and provides services such as transaction management, database connection pooling, and instance lifecycle management.

EJB component implementation The Java class that runs in the server implements the Bean's business logic. The class must implement the remote interface methods and additional methods for lifecycle management.

EJB component types

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

- Stateful session Beans
- Stateless session Beans
- Entity 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.
- 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.

EJB transaction attribute values

Each EJB component has a transaction attribute that determines how instances of the component participate in transactions. In EJB Server, you set the transaction attribute in the Transaction tab of the Component Properties dialog box.

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.

A session Bean 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.

Table 3-1 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 3-1: Transaction attribute values

Attribute	Description
Not Supported	<p>(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.</p> <p>Since entity Beans are almost always involved in transactions, this value is not usually used for an entity Bean.</p>
Supports	<p>The EJB component can execute in the context of an EJB Server 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.</p>
Required	<p>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.</p>
Requires New	<p>Whenever the EJB component is instantiated, a new transaction begins.</p>
Mandatory	<p>EJB component methods must be called in the context of a pending transaction. If a client calls a method without an open transaction, the EJB Server ORB throws an exception.</p>
Never	<p>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, EJB Server throws an exception.</p>
Bean Managed	<p>(For EJB session Beans only.) The EJB component can explicitly begin, commit, and roll back new, independent transactions by using the <code>javax.transaction.UserTransaction</code> 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 EJB Server transaction.</p>
Default to component	<p>(Method-level default) In the Transactions tab of the Method properties window, choose this option if the method should inherit the transaction attribute set in the component properties.</p>

EJB container services

The EJB container provides services to EJB components. The services include transaction and persistence support.

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. See Chapter 8, “Managing Persistent Component State,” for more information.

EJB support

EJB Server can host EJB components developed according to version 1.1 of the Enterprise JavaBeans specification. EJB Server supports session Beans and entity Beans with Bean-managed persistence or container-managed persistence.

See the complete EJB 1.1 specifications from Sun Microsystems at <http://java.sun.com/products/ejb/>.

Running EJB components in EJB Server

You can run Enterprise JavaBeans as EJB Server components using any of these techniques:

- Define EJB components in PowerJ, using wizards to define the interfaces and deploy the Bean directly from PowerJ to EJB Server. See the PowerJ documentation for more information.
- Use the Adaptive Server plug-in to Sybase Central to import an EJB-JAR file that contains the classes and deployment descriptors for one or more EJB components. The Adaptive Server plug-in defines components with properties matching the deployment descriptor settings.
- Import compiled versions of a home interface, remote interface, implementation class, and (for entity Beans) the primary key class. The Adaptive Server plug-in defines IDL interfaces for the interfaces and the primary key, and defines an EJB component with default settings. You can configure additional settings such as transaction attributes and database resource references using the Adaptive Server plug-in Component Properties dialog box.
- Define an EJB component from scratch in the Adaptive Server plug-in, using the IDL generation tools to define the home interface, remote interface, and primary key type. The Adaptive Server plug-in generates Java classes for the home and remote interfaces and primary key class, as well as a template for the implementation class.

EJB clients connecting to EJB Server

EJB Server also supports the Enterprise JavaBean client model by generating EJB proxies and providing an EJB-compliant implementation of the JNDI NamingContext class. You can generate EJB-style proxies for any IDL interface, and use the proxies to call methods on components that implement that interface. The NamingContext class can also be used in EJB components to instantiate home interfaces for intercomponent calls.

Creating Component-Based Applications

About this chapter

This chapter describes the process of designing, building, and deploying applications with components executing in EJB Server.

Contents

Topic	Page
Application architecture	42
Designing the application	44
Implementing components and clients	46
Deploying the application	48

Application architecture

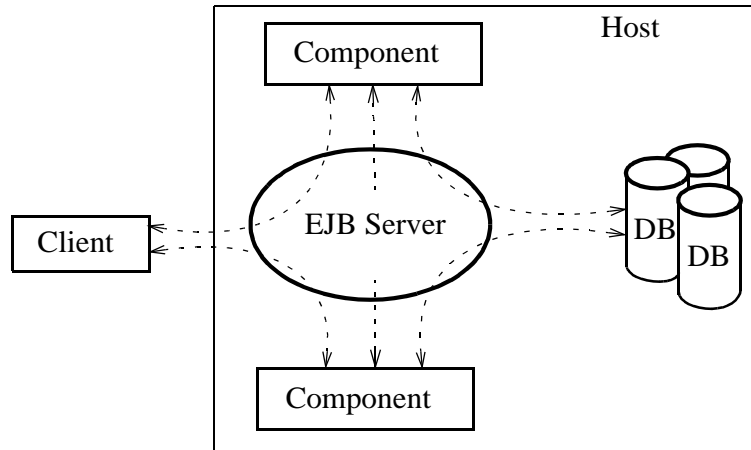
EJB Server applications are composed of clients and an EJB Server that hosts components. The clients can run on different machines; the components execute on the host server machine as part of the EJB Server process. Components, in turn, connect to databases on the host server machine.

Building EJB Server applications is different from building standard client/server applications in that the parts of the application communicate with each other in a variation of traditional three-tiered architecture.

EJB Server architecture

In traditional three-tiered architecture, the client resides on the first tier, the application server and components reside on the second tier, and remote databases reside on the third tier. In the EJB Server three-tiered architecture, see Figure 4-1, the databases reside on the same host as the EJB Server. They communicate using a Sybase high-speed JDBC driver through shared memory.

Executing methods on a component from the client or another component, retrieving data from databases, and other communications are managed by the EJB Server. EJB Server handles the details of transactions, threads, security, database connections, and network communication so that you can concentrate on writing the business logic and user interface for the components and clients.

Figure 4-1: EJB Server architecture

As in client/server applications, the client contains the user interface. Unlike client/server applications, however, business logic (such as stored procedures) is separate from both the clients and the database. Instead, business logic resides in the second tier as components that analyze data, perform computations, or retrieve information from data sources and process it. You design an EJB Server application by coding these tasks into an interface and into method prototypes.

A primary benefit of this model is that you can include pre-built components in the EJB Server application. If these components have been built outside EJB Server, you can import them using the Adaptive Server plug-in. Importing components adds their interfaces and method prototypes to EJB Server. The client and components are built from the same interface and method prototypes. You can build the client and components concurrently, as long as the client and component development teams notify each other if either of them changes the interface or method prototypes.

Designing the application

In the design stage, you plan the infrastructure for developing and deploying the application, define the EJB Server components, the component interfaces, and the EJB Server packages that contain the components. At the end of this phase, you will have packages and components defined in the Adaptive Server plug-in.

Follow these steps to design the application:

- 1 “Plan for server infrastructure needs” on page 44
- 2 “Define EJB Server packages” on page 44
- 3 “Define components” on page 45
- 4 “Define connection caches” on page 46

Plan for server infrastructure needs

For an enterprise application implemented by several developers, you may need to create several EJB Servers to increase developer productivity. For example, you might want dedicated servers for each of the following:

- **Component development** Servers to test components that are under development or revision. A typical configuration uses one server per developer, running on the developer’s personal workstation.
- **Client testing/Quality Assurance (QA)** Client developers require a server with a stable installation of the application components, to be used by client developers to test their programs. During the early development phase, you can deploy **stubbed** components to this server to allow testing of client connectivity and basic method execution. (A stubbed component has empty method implementations. For most component models, the EJB Server generates source for a stubbed implementation when you generate the component skeleton.)
- **Production** You will need to install EJB Server on the host machine for the live version of the application. For Internet applications, this machine must be available to clients that are outside your corporate firewall.

Define EJB Server packages

Components must be installed in a package before they are available for use in applications. You should install components that perform related tasks together in a single package. Chapter 6, “Working with EJB Packages and Components,” describes how to create packages in the Adaptive Server plug-in.

Packages are the units of deployment for your application; you can use the Adaptive Server plug-in to import and export archives of a package, its installed components, and related application files. For example, you can deploy a tested configuration by exporting packages from your test server and importing them into the production server. For more information, see “Deploying components” on page 48.

Packages are also one level in the EJB Server authorization hierarchy. You can edit the package’s required Role Memberships to restrict which users can access components in the package.

Define components

For each component, you must choose the component model, design the component interface, determine transactional semantics, and define the component in EJB Server.

Choose the component model Choose the component model based on your development team’s expertise. “Server-Side Component Support” on page 7 describes the available component models.

Design the transactional semantics You must decide what transactional semantics the component will follow and how the component lifecycle will be managed. Chapter 5, “Understanding Transactions and Component Lifecycles” explains the design concepts for transaction and lifecycle control in EJB Server components.

The following design decisions determine how EJB Server manages your component’s transactions:

- Which transaction attribute the component uses
- Whether transaction boundaries are managed explicitly in the component implementation or implicitly by EJB Server

If your component interacts with the database, you must specify a transactional attribute that determines how the component’s database work is grouped within EJB Server transactions. If another component invokes your component, the transaction attribute determines whether your component’s database work is done independently or as part of the existing EJB Server transaction.

You must also decide whether or not you will code your component to manage transaction boundaries explicitly. To manage transaction boundaries explicitly, each method must call one of EJB Server’s transaction state primitives to indicate the status of the component’s transactional work. “Using transaction state primitives” on page 61 describes this topic in detail.

Instead of writing code to manage transaction boundaries explicitly, you can set the component's Automatic demarcation/deactivation property in the Adaptive Server plug-in. This setting is appropriate if every method in your component executes a complete unit of transactional work (in other words, the transactional outcome is never pending when a method returns). When this option is enabled, EJB Server deactivates the component instance after every method invocation. Upon deactivation, the transaction is always committed unless the component aborts the transaction by calling the `rollbackWork` transaction primitive. In the Adaptive Server plug-in, the Automatic demarcation/deactivation property is set in the Component Properties window, beneath the Transactions tab. "Configuring component properties" on page 75 describes how to view and modify component properties in the Adaptive Server plug-in.

For any component, transactional or not, you must decide how your component's instance lifecycle will be managed. "Component lifecycles" on page 51 describes the general instance lifecycle model and your options for instance lifecycle management.

Define the component in the Adaptive Server plug-in Use the Adaptive Server plug-in to define the components. If you have already created Java components, you can import the component interfaces into the Adaptive Server plug-in—you do not need to define method prototypes again in the Adaptive Server plug-in.

Define connection caches

Connection caching increases the scalability of your application, since it eliminates repetitive login/logoff operations for connections to databases. Connection caching is also required for EJB Server transactions to function as intended.

You must define a connection cache for the database that your components interact with, and then implement your components to use cached connections. See "Managing connection caches" on page 153, which describes how to define connection caches in the Adaptive Server plug-in

Implementing components and clients

With the design in place, your component developers and client developers can begin implementing the clients and components that form the application.

Implementing components

To create an EJB component, use PowerJ or another JDK-1.2 compatible development tool to create the component. From PowerJ you can import the component definitions into EJB Server, and deploy the component on the EJB Server. If using another development tool, you must perform these tasks with the Adaptive Server plug-in.

To learn how to build EJB components, see Chapter 6, “Working with EJB Packages and Components.”

Design and implement the client

Client developers can work concurrently with component developers. To allow prototyping and testing of client programs, you may want to create a client test server that hosts stubbed versions of the application components (that is, components with minimal method implementations). All clients for EJB Server components must be Java clients.

The Java client Java applets do not require customer installation and simplify the task of providing upgrades. The customer always downloads the most recent applet. If you do not want the customer to wait for the Java classes to download from the EJB Server, you can install the Java classes on the client machine or use Marimba Castanet to speed up the download time.

If the client application is large and requires many Java classes, download time might be unacceptable. In this case, use a Java application that is installed locally on the client machine. This approach is ideal for intranet customers or even regular Internet customers. Although not as simple as providing upgrades with an applet, Java applications are no more difficult to upgrade than conventional software. In fact, Marimba Castanet can be used to automatically upgrade the Java application across the Internet.

Java IDEs such as PowerJ offer visual interface builders that greatly simplify the implementation of the user interface.

In some situations, you might want to implement different versions of a client for different users. For example, you may implement a Java applet version to allow new customers to connect over the Internet without installing a client program. For established customers who use the application heavily, you can implement a standalone client program that offers improved performance.

Client design issues In designing your client, plan to optimize network performance by keeping traffic between the client and components on the server to a minimum. To optimize network performance, plan to:

- Cache property changes in client data structures.
- Validate field values on the client.

- Update only the rows and columns that have changed. For example, do not implement a client to update an entire table when only a few rows have changed.
- Group data changes into larger sets with fewer method calls.

Deploying the application

After you have tested and debugged the application on your test server, it is time to deploy the component files to a production server and make the client application files available to the application users. Follow these steps to deploy the application:

- 1 “Deploying components” on page 48.
- 2 “Developing clients” on page 49.

Deploying components

To deploy components, you copy component definitions and implementation files to EJB Server. There are two ways to do this:

- Using PowerJ
- Importing an EJB JAR

Using PowerJ

If developing Java clients and components, you can deploy your application to the EJB Server directly from the PowerJ IDE. To deploy components, configure deployment options using the Run | Deploy options menu item, then deploy using the Run | Deploy menu item. See the PowerJ documentation or online help for more information.

Importing an EJB JAR

You can use a Java development tool such as Sybase PowerJ to define and develop Beans in the EJB 1.1 format and create an EJB-JAR file. The Adaptive Server plug-in can read the JAR file and create a package containing a component for each Bean in the JAR file. See Chapter 6, “Working with EJB Packages and Components,” for more information about importing EJB JAR files.

Developing clients

You can use PowerJ to develop Java clients for EJB Server components. Basic tasks for developing clients include:

- Generate EJB stubs.
- Add code to create the initial naming context and instantiate the home interface proxies.
- Add code to instantiate remote interface proxies.
- Add code to call remote interface methods.

See Chapter 7, “Creating Enterprise JavaBean Clients,” for detailed information about developing clients.

Understanding Transactions and Component Lifecycles

This chapter explains the EJB Server component lifecycle and transaction processing models. Transactions allow you to group database updates performed by multiple components into a single atomic unit of work, which greatly simplifies error recovery in component-based applications.

The component lifecycle determines how instances of a component are allocated, bound to a client, and destroyed. The EJB Server component lifecycle is designed to maximize reuse of resources and minimize the possibility that a client application can monopolize a server resource.

The component lifecycle and the transaction model are tightly integrated. You must understand both to use transactions effectively in your application.

Topic	Page
Component lifecycles	51
The EJB Server transaction processing model	55
OTS/XA transaction model	66

Component lifecycles

The EJB Server component lifecycle is designed to:

- Maximize sharing and reuse of server resources
- Minimize the possibility that a client application can monopolize server resources

To achieve these goals, EJB Server supports the concepts of component instance pooling and early deactivation.

Instance pooling allows a single component instance to service multiple clients. The component lifecycle contains activation and deactivation steps: Activation binds an instance to an individual client; deactivation indicates that the instance is unbound. Instance pooling eliminates resource drain from repeated allocation of component instances.

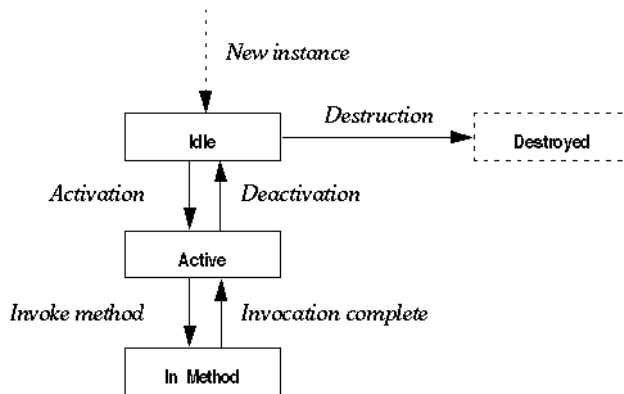
Early deactivation allows a component's methods to specify when deactivation occurs. Early deactivation prevents a client application from tying up the resources that are associated with a component instance and allows the instance to serve more clients in a given time frame.

A component that is deactivated after each method call and supports instance pooling is said to be a **stateless component** because the component's state is reset across the boundary of a transaction and activation. Early deactivation and instance pooling promotes greater scalability by enabling an increasing number of clients to use a static number of instances. An application design based on stateless components offers the greatest scalability.

States in the component lifecycle

Generic component lifecycle EJB Server components in any component model follow the state diagram illustrated in this figure:

Figure 5-1: States in the EJB Server component lifecycle



The state transitions are as follows:

- **New instance** The EJB Server runtime allocates a new instance of the component. The instance remains idle in the instance pool waiting for the first method invocation.
- **Activation** Activation prepares a component instance for use by a client. Once an instance is activated, it is bound to one client and can service no other client until it has been deactivated. If a component is transactional, activation also indicates the beginning of the instance's participation in a transaction.
- **In method** In response to a method invocation request from the client, the EJB Server runtime calls the corresponding method in the component. The next state depends on which of the transaction state primitives the method calls before returning. (The state transition also depends on whether the method returns with an uncaught exception.) See "Using transaction state primitives" on page 61 for more information.
- **Deactivation** Deactivation indicates that the component is no longer bound to the client. Methods can call either the `completeWork` or `rollbackWork` transaction state primitives to cause explicit deactivation of the instance. As discussed in "Using transaction state primitives" on page 61, these primitives also affect the transaction's outcome. Deactivation can also occur automatically, under any of the following circumstances:
 - If the instance is participating in a transaction, the instance is deactivated when the transaction commits, rolls back, or times out.
 - If you have configured the component's Instance Timeout property to a finite setting, an instance is deactivated if the time between consecutive method calls exceeds the timeout value. "Resources tab component properties" on page 80 describes how to configure this property.
- **Destruction** Destruction occurs if the component instance cannot be recycled. "Supporting instance pooling in your component" on page 54 describes how to ensure instance reuse. If the component cannot be reused, deactivation is followed by destruction of the instance.

The EJB Server component lifecycle allows component instances to be recycled; idle component instances can be cached when idle and bound to the service of individual clients only as needed. If your component has been coded to support early deactivation, a client holding a reference to the component's stub or proxy object may be serviced by several different instances of the component. After each deactivation, the next method invocation causes an instance to be activated and bound to the client. Overall server scalability is increased because a new instance does not have to be instantiated each time a client invokes a method.

Supporting instance pooling in your component

Instance pooling eliminates resource drain caused by repeated allocation of new component instances.

For Java components, you can implement a lifecycle-control interface to control whether the component instances are pooled. These interfaces also provide activate and deactivate methods that are called to indicate state transitions in a component instance's lifetime. For more information on these interfaces, see the following sections:

- *Java components* can implement the interface `jaguar.beans.enterprise.ServerBean`.

To support instance pooling, code that responds to activation events must restore the component to its initial state (that is, as if it were newly created). The Java interface has methods that allow an instance to selectively refuse pooling: `canReuse` in Java.

When the component Pooling option is set in EJB Server, the Java `canReuse` method is not called, even if the component implements the `ServerBean` Java interface.

Stateful versus stateless components

A component that can remain active between consecutive method invocations is called a **stateful component**. A component that is deactivated after each method call and that supports instance pooling is said to be a **stateless component**. Typically, an application built with stateless components offers the greatest scalability.

Stateful components A stateful component remains active across method calls.

Since deactivation happens at the mercy of client applications, you may wish to configure the Instance Timeout property for stateful components so that a client cannot monopolize a component instance indefinitely. See "Resources tab component properties" on page 80 for more information.

Stateless components In order for a component to be stateless, both of the following must be true:

- You have configured or implemented the component to be deactivated after every method invocation. In the Adaptive Server plug-in, you can enable the Automatic deactivation / demarcation property for the component (located on the Transactions tab in the Component Properties window). Alternatively, you can implement the component so that it calls either `completeWork` or `rollbackWork` in every method.
- You have enabled the Pooling option in the Component Properties window (this option is located on the Instances tab).

Stateless components cannot use instance-specific data to accumulate data between method invocations.

Some situations require that you accumulate data across method invocations. For example, a `PurchaseOrder` component might have an `addItem()` method that is called repeatedly to specify the contents of an order. In lieu of instance-specific data, you can use one of these alternatives to accumulate data:

- **Accumulate data in the database** Use connection caching and database commands to accumulate data in the database. This is the preferred technique.
- **Accumulate data in the client** Create a data structure that is passed to each method invocation and contains all accumulated data. This technique is only practical if the amount of data is small. Sending large amounts of data over the network will degrade performance.
- **Accumulate data in a file** If the accumulated data is small and represented by simple data structures, you can store the data in a local file.

The EJB Server transaction processing model

An *EJB Server transaction* is a transaction whose boundaries and outcome are determined by EJB Server. Components can be marked as transactional in the Adaptive Server plug-in. If a component is transactional, the EJB Server transaction manager ensures that the component's third-tier database queries execute as part of a transaction. Multiple components can participate in an EJB Server transaction; the EJB Server transaction manager ensures that all database changes performed by the participating transactions are all committed or rolled back.

Transactions

All transactions are defined by the ACID test:

- **Atomic** If a transaction is interrupted, all changes that the transaction has made are cancelled or rolled back.
- **Consistent** A transaction produces results that preserve invariant properties.
- **Isolated** A transaction's intermediate states cannot be monitored or changed by other transactions; transactions execute their results one after another.
- **Durable** The changes that a transaction completes are permanent.

How EJB Server transactions work

In the Adaptive Server plug-in, you can declare EJB Server components to be transactional. When a component is transactional and uses the EJB Server connection management feature, commands sent on a third-tier-database connection are automatically performed as part of a transaction. Component methods can call the EJB Server transaction state primitives to influence whether EJB Server commits or aborts the current transaction.

The component lifecycle is tightly integrated with the EJB Server transaction model. Component instances that participate in a transaction are not deactivated until the transaction ends or until the component indicates that its contribution to the transaction is over (that is, its work is done and ready for commit or that its work must be rolled back). An instance's time in the active state corresponds to the beginning and end of its participation in a transaction.

Benefits of using EJB Server transactions

The benefits of using transactions to group database updates are clear. You can easily code methods in a single component to implement transactions that run against a single data source. However, those methods may in turn be executed by another component, which itself is defining a transaction. In this situation, error recovery becomes difficult. For example, consider the following scenario in which an Enrollment component calls both Registrar and Billing components:

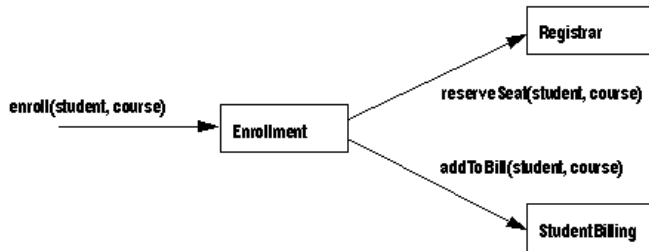
A transaction involving multiple components

In the following figure, the Enrollment.enroll() method calls methods in the Registrar and StudentBilling components:

- Registrar.reserveSeat() checks that a seat is available. If so, it decrements the count of available seats and adds the student to the course's enrollment list. If no seats are available, reserveSeat() fails.

- `StudentBilling.addToBill()` checks that the student has a billable credit record. If so, `addToBill()` adds the course cost to the student's bill for that semester. If the student has a credit problem (if, for example, she owes money for an overdue book), `addToBill()` fails.

Figure 5-2: An example EJB Server transaction



To be correct, both the database update made by the Registrar and the update made by the StudentBilling components must occur, or neither must occur. In other words, if the student cannot be billed, the course's available seats must not be changed. To handle this case, you could add logic to the `enroll()` method to undo changes (requiring an `unreserveSeat()` method in Registrar). However, as more components are added to the scenario, the logic needed to undo previous changes quickly becomes unmanageable. It is much easier to define all the participating components to use EJB Server transactions. Then an error in any component can induce a rollback of all changes made by the other participating components before the error occurred.

By defining the participating components to use EJB Server transactions, you can be sure that the work performed by the components that participate in a transaction occurs as intended.

Defining transactional semantics

❖ **To define how a component participates in transactions, you must:**

- 1 Choose a transaction coordinator. The transaction coordinator manages the flow of transactions that involve more than one connection. "Transaction coordinators" on page 58 describes the available options.

- 2 Specify the component's transaction attribute. Each component has a transaction attribute that determines whether instances of the component participate in transactions. "Transactional component attribute" on page 58 describes the attribute settings and their meanings.
- 3 Code methods to call the EJB Server transaction state primitives. Each method should call the appropriate transaction state primitive to reflect the state of the work that the component has contributed to the transaction. "Using transaction state primitives" on page 61 describes the state primitives in detail.
- 4 Specify a transaction timeout period if needed. By default, transactions are never timed out. You can configure a finite timeout period in the Adaptive Server plug-in. See "Transaction Timeout property" on page 62 for more information.

Transaction coordinators

All components installed in one EJB Server share the same transaction coordinator.

Choices for transaction coordinator include:

- **Shared connection** This "pseudo-coordinator" is built into EJB Server. In this model, all components participating in a transaction share a single connection. To use this model, all of your application data must reside on one data server, and all components that participate in a transaction must use a connection with the same user name and password.
- **OTS/XA transactions** For NT or UNIX users, this option complies with the Object Transaction Service (OTS) and X/Open Architecture (XA) standards. This option uses the Transarc Encina® transaction coordinator that is built into EJB Server. The Encina transaction coordinator uses two-phase commit to coordinate transactions among multiple databases.

The default coordinator is the "Shared Connection" coordinator. To view or change the coordinator, use the Server Properties dialog box in the Adaptive Server plug-in.

Transactional component attribute

Components in EJB Server have a transaction attribute that indicates how a component participates in transactions. You can view and change a component's transaction attribute using the Adaptive Server plug-in; the attribute is displayed on the Transactions tab in the Component Properties window. The attribute has the following values:

- **Not Supported** The Default. The component's methods never execute as part of a transaction. If the component is activated by another component that is executing within a transaction, the new instance's work is performed outside of the existing transaction.

- **Supports Transaction** The component can execute in the context of a EJB Server transaction, but a connection is not required in order to execute the component's methods. If the component is instantiated directly by a base client, EJB Server does not begin a transaction. If component A is instantiated by component B, and component B is executing within a transaction, component A executes in the same transaction.
- **Requires Transaction** The component always executes in a transaction. When the component is instantiated directly by a base client, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A executes within the same transaction; if B is not executing in a transaction, then A executes in a new transaction.
- **Requires New Transaction** Whenever the component is instantiated, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A begins a new transaction that is unaffected by the outcome of B's transaction; if B is not executing in a transaction, then A executes in a new transaction.
- **Mandatory** Methods may only be invoked by a client that has an outstanding transaction.
- **Bean Managed** Uses EJB 1.1 transactional behavior. The component cannot inherit a client or other component's transaction. The component can execute without a transaction or explicitly begin, commit, and roll back transactions by using the `javax.transaction.UserTransaction` interface for EJB components.

The following table lists design scenarios and the transaction attributes that apply to each.

Table 5-1: Deciding on a transaction attribute

Design scenario	Applicable transaction attributes
Your component interacts with the database, and its methods may be called by another component as part of a larger transaction. Multiple updates are issued before calling <code>completeWork</code> , or an update depends on the results of queries that were issued since the last call to <code>completeWork</code> .	Requires Transaction or Requires New Transaction
Updates from your component are performed by a single database update, the update logic is independent of any other query issued by the method, and you call <code>completeWork</code> in each method that issues an update. In other words, your component's updates are already atomic.	Supports Transaction
Your component's methods make intercomponent method calls, and the work done by called components must be included in one transaction.	Requires Transaction or Requires New Transaction
Methods in the component interact with more than one database, and updates to different databases must be grouped in the same transaction (this also requires a transaction coordinator that supports two-phase commit to those databases).	Requires Transaction or Requires New Transaction
Transactions begun by your component must not be affected by the outcome of transactions begun by other components that call your component.	Requires New Transaction
Work done by your component must never be done as part of a transaction.	Not Supported

For example, in the scenario illustrated in “A transaction involving multiple components” on page 56, the Enrollment component must be marked *Requires Transaction* or *Requires New Transaction*, since it calls methods in the Registrar and StudentBilling components, and the work performed by the called components must be grouped in a single transaction. Both Registrar and StudentBilling must be marked *Supports Transaction* or *Requires Transaction* so that their database updates can be grouped in the transaction begun by the Enrollment component.

Transaction Not Supported is useful when your component performs updates to a noncritical database. For example, consider a component whose sole function is to log usage statistics to the database. Since usage statistics are not mission-critical data, you can choose *Not Supported* as the component's transaction attribute to ensure that the logging updates do not incur the overhead of using two-phase commit.

Determining when transactions begin

After a base client instantiates a transactional component, the first method invocation begins an EJB Server transaction. This instance is said to be the **root instance** of the transaction. If the root instance invokes methods in other transactional components, those components join the existing transaction.

The outcome of the transaction is determined by how the participating components call the transaction state primitives discussed in “Using transaction state primitives” on page 61.

Use the home interface for the called component

For transactions to occur with the intended semantics, you must perform intercomponent calls using the home interface. Do not invoke another component’s methods directly.

Using transaction state primitives

EJB Server provides transaction state primitives that methods can call to direct the outcome of the current transaction. Each component model provides an interface containing methods for these primitives.

These methods end a component’s participation in a transaction (both cause the current instance to be deactivated):

- **completeWork** The component finished its work for the current transaction and should be deactivated when the method returns.
- **rollbackWork** The component cannot complete its work. Doom the current transaction and deactivate the instance when the method returns.

These methods are used to maintain state after the method returns (they delay deactivation of the component instance):

- **continueWork** Continue this component’s participation in the current transaction after the method returns, and allow the transaction to be committed if the component is deactivated. If a method calls no transaction primitive, this is the default behavior.
- **disallowCommit** Continue this component’s participation in the current transaction after the method returns, but roll back the transaction if the component is deactivated before calling another primitive besides `disallowCommit`.

These primitives can be used to query the state of the transaction (if any) in which the method is executing:

- **isInTransaction** Query whether the current method is executing in the context of a transaction.

- **isRollbackOnly** Query whether the current transaction is doomed to be rolled back or is still viable.

The following table describes how the transaction primitives are invoked in Java components.

Table 5-2: Java transaction primitives

Transaction primitive	Java InstanceContext method
completeWork	completeWork
rollbackWork	rollbackWork
continueWork	continueWork
disallowCommit	None. You can achieve the same effect by calling, and then raising an exception if deactivate is called before the next method invocation.
isInTransaction	inTransaction
isRollbackOnly	isRollbackOnly

Any participating component can roll back the transaction by calling the `rollbackWork` primitive; Java components can also cause a rollback by returning an unhandled exception. Only the action of the root component determines when EJB Server commits the transaction. The transaction is committed when the root component returns with a state of `completeWork` and no participating component has set a state of `disallowCommit`.

You can use the transaction state primitives in any component; the component does not have to be declared transactional. Calling `completeWork` or `rollbackWork` from methods causes early deactivation.

Transaction Timeout property

The root instance’s Transaction Timeout property specifies the maximum duration of an EJB Server transaction. The default timeout period is infinite. You can configure finite timeouts in the Adaptive Server plug-in, as described in “Resources tab component properties” on page 80.

A transaction begins when a base client activates a transactional component; this component is the root component of the transaction. The root component’s Transaction Timeout property determines the maximum duration of the transaction.

If the transaction is not committed or rolled back within the allotted time, it is automatically rolled back. In this case, the client receives the CORBA TRANSACTION_ROLLEDBACK exception when it tries another method invocation. The client’s object reference remains valid, and the transaction can be retried.

Transactions are never rolled back in the middle of a method invocation. If the timeout occurs during a method invocation, and the method does not commit the transaction, the transaction is rolled back when the invocation completes.

Example

As discussed in “Benefits of using EJB Server transactions” on page 56, EJB Server transactions are most useful when your application uses intercomponent calls.

As an example, consider the scenario illustrated in “A transaction involving multiple components” on page 56. The pseudocode below shows the logic used to ensure that the work performed by the Registrar.reserveSeat() and StudentBilling.addToBill() occurs within the same transaction.

In the Registrar component, the reserveSeat() method must check the number of seats. If there is space for the new student, then the method adds the student, decrements the count of available seats, and sets a state of completeWork. If a seat is not an available, the method calls rollbackWork to roll back the current transaction.

Here is the pseudocode for Registrar.reserveSeat():

```
check number of seats
if enough seats
    decrement number of seats
    add student to enrollment list
    completeWork
else
    rollbackWork
end if
```

The transaction attribute for Registrar must be *Requires Transaction* so that the query for available seats and the update of available seats always occur in the same transaction.

In the StudentBilling component, the addToBill() method must verify the student’s credit. If the student does not already owe money, the method adds the cost to the semester bill and sets a state of completeWork. If the student owes money, the method calls rollbackWork to roll back the current transaction. Here is the pseudocode for StudentBilling.addToBill():

```
check student’s balance
if balance > 0
    add cost to bill
    debit balance
```

```
        completeWork
    else
        rollbackWork
    end if
```

The transaction attribute for StudentBilling must be *Requires Transaction* so that the balance query, the billing calculation, and the debit of the student's balance always occur in the same transaction.

In the Enrollment component, the enroll() method first calls Registrar.reserveSeat(). After Registrar.reserveSeat() returns, the method checks whether the transaction is still viable using the isRollbackOnly primitive. If the transaction is viable, the method calls StudentBilling.addToBill(). Here is the pseudocode for Enrollment.enroll():

```
    invoke Registrar.reserveSeat()
    if isRollbackOnly returns true
        return
    else
        invoke StudentBilling
        completeWork
    endif
```

The transaction attribute for Enrollment must be *Requires Transaction* so that the work done by StudentBilling and Registrar occurs as a single transaction.

Dynamic enlistment in Bean-managed transactions

EJB Server supports dynamic enlistment for Bean-managed transactions, which allows you to create a connection in one method of a stateful Bean, use the connection in another method, and close the connection in a third method.

For a JDBC 2.0 shared connection (PooledConnection), the container manages the single connection's enlistment and disenlistment in transactions.

For XA connections, the Object Transaction Service libraries need to know all the resources that will participate in a transaction when it starts. If you get an XAConnection before you start a transaction, EJB Server enlists the XAConnection in the transaction. If you start a transaction before you create an XAConnection, EJB Server creates the connection and enlists it in the transaction.

Dynamic enlistment allows you to do this:

```
conn1 = dsl.getConnection();
// A
```

```
user_transaction.begin();  
//  
conn2 = ds2.getConnection();  
conn3 = ds3.getConnection();  
// B  
conn2.close();  
//  
user_transaction.commit();  
// C  
conn3.close();  
conn1.close();
```

Where at these points, the following are true:

- A – conn1 is not part of any transaction.
- B – conn1, conn2, and conn3 are part of the user_transaction.
- C – conn1 and conn3 are not part of any transaction.

You can get only one connection per resource. Each getConnection call for the same database returns the same connection.

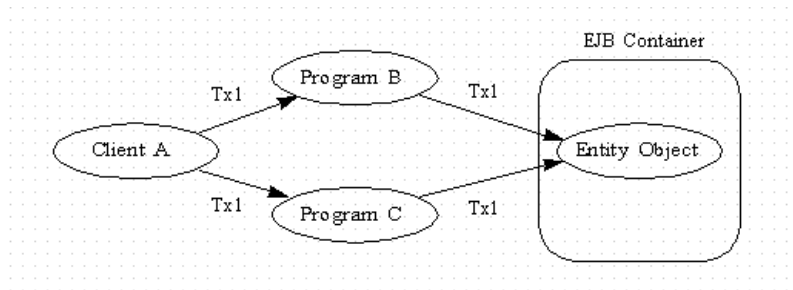
Warning! XA performance diminishes when connections span methods.

Entity Bean local diamonds

An entity object accessed from more than one path in the same transaction, as shown in Figure 5-3, is called a *diamond*. A *local diamond* exists when the access paths originate from, and the entity object resides on, the same server.

Typically, EJB Server uploads data from the database at the beginning of a transaction and downloads data to the database at the end of a transaction. When more than one program accesses an Entity Bean within the same transaction, this can lead to inconsistent views of the data. For instance, if Program B updates the entity's data and then Program C reads the data, Program C does not see the changes made by Program B. To solve this problem, when EJB Server detects a diamond, it uploads data at method invocation and downloads data when the method completes.

Figure 5-3: Entity object diamond



OTS/XA transaction model

EJB Server includes CORBA Object Transaction Service (OTS) and X/Open architecture (XA) as one of its distributed transaction models. The Transarc Encina® transaction coordinator manages OTS/XA transactions for EJB Server. You can define components, and component methods so that the transaction coordinator automatically handles transactions (called implicit control). You can also write code in the component or client to manage transactions (called explicit control).

EJB Server provides for distributed transactions using the two-phase commit protocol. Two-phase commit ensures that all changes to recoverable resources (for example, multiple database servers) occur atomically, and the failure of any resource to complete causes all other resources to undo changes. Two-phase commit consists of a prepare phase and an execution phase. In the prepare phase, the transaction coordinator validates that all resources are available. In the execution phase, the transaction coordinator executes all updates to the resources.

Note EJB Server does not currently support nested OTS/XA transactions (also called subtransactions). If a method attempts to create a subtransaction, the `SubTransactionUnavailable` exception is raised.

Component lifecycle
and transactional
behavior

An OTS/XA transaction coordinator uses XA resources to manage transactions. An XA resource manages information using an XA-compliant interface, for example, a database server or IBM's MQSeries® (a message queueing system). The XA interface standard is an element of the X/Open Distributed Transaction Processing (DTP) model. Currently, Sybase provides an XA-compliant interface through CT-Lib. In addition, EJB Server supports jConnect, which is a JTA (Java Transaction API)-compliant JDBC driver. See the "Managing XA resources" on page 159 for detailed information about enabling and managing XA resources.

A component with the OTS-Style transaction attribute enabled follows the standard component lifecycle as described in "Component lifecycles" on page 51.

Generally, OTS transactions behave in the same way as described in "The EJB Server transaction processing model" on page 55. For more information about component transaction attributes, see "Transactional component attribute" on page 58.

EJB Server does not support transactions that spawn over multiple EJB Servers.

Working with EJB Packages and Components

This chapter describes how to install, modify, and deploy Enterprise JavaBeans in EJB Server packages using the Adaptive Server plug-in.

If your site uses PowerJ, see the PowerJ documentation for information on deploying EJB classes directly to EJB Server from PowerJ.

Topic	Page
Packages and Enterprise JavaBean components	69
Importing Enterprise JavaBeans	71
Installing components	74
Modifying components	75
Configuring component properties	75
Generating stubs and skeletons	86
Creating Enterprise JavaBeans	87
Modifying packages	93
Configuring package properties	94
Exporting packages to EJB-JAR files	95

Packages and Enterprise JavaBean components

In the Adaptive Server plug-in, EJB Server packages allow you to group related EJB components as a logical unit. Typically, components in a package work together to provide a coherent service or function.

You can create JavaBeans from scratch or, more likely, import the JavaBeans to EJB Server using the Adaptive Server plug-in. When you import JavaBeans, they must be contained in a JAR file or Java class file. The Adaptive Server plug-in reads the JAR file or class file and creates an EJB Server package containing a component for each Bean in the file. See “Importing Enterprise JavaBeans” on page 71.

EJB Server packages serve the following purposes:

- **They are a unit of deployment** Using the Adaptive Server plug-in, you can import and export archived copies of the components in a package and related application files.
- **They allow you to control which users can access the components** Packages form one level in the EJB Server authorization hierarchy. A package is not available to the user unless it is deployed to the EJB Server's Installed Packages folder. The Adaptive Server plug-in allows you to map roles through each package for methods within the package. See "Configuring package properties" on page 94 for information about mapping roles.

Default packages

When EJB Server is installed, the AseAuth default package is deployed automatically to EJB Server. You will see this package in the Installed Packages folder. AseAuth contains information you need to log in to EJB Server from the Adaptive Server plug-in. *Do not alter or delete this package.*

Importing Enterprise JavaBeans

This section describes how you import Enterprise JavaBeans. These are the usual methods you will use with EJB Server. See “Creating Enterprise JavaBeans” on page 87 for directions for creating JavaBeans from scratch.

The Adaptive Server plug-in to Sybase Central supports two methods of importing Enterprise JavaBeans:

- **From an EJB-JAR file** An EJB-JAR file contains the implementation classes, interface classes, and deployment descriptor for one or more Beans. You can use a Java development tool such as Sybase PowerJ to define and develop Beans and create an EJB-JAR file. You can import JAR files in the EJB 1.1 format. The Adaptive Server plug-in reads the JAR file and creates a package containing a component for each Bean in the JAR file.
- **From an EJB class file** The Adaptive Server plug-in can import component and method information from Java class files. Use this method if you have created a Bean’s interfaces and implementation class, but have not created a deployment descriptor. You will need to manually configure properties that would otherwise be read from the deployment descriptor. You cannot import Java package files.

PowerJ deploys Enterprise JavaBeans directly to EJB Server

If you are developing in PowerJ, use the Enterprise JavaBean Deployment Wizard to install EJB components in EJB Server. If using another IDE, use the Adaptive Server plug-in to import the Bean as described below.

Note Finder methods in an entity Bean’s home interface can return `java.util.Collection` or `java.util.Enumeration`. All EJB components defined in a package or an EJB-JAR file must use the same type for finder method return values.

Importing EJBs from an EJB-JAR file

Importing an EJB 1.1 JAR file is a two-step process:

- 1 Deploy the JAR file to the repository.
- 2 Install the package in the Installed Packages folder in the Adaptive Server plug-in.

❖ **Deploying an EJB 1.1 JAR file to the repository**

- 1 Start the Adaptive Server plug-in if it is not already running, and connect to the EJB Server where you want to install the component.
- 2 Double-click the Installed EJB Packages folder.
- 3 Select File | Deploy | EJB 1.1 JAR
The Deploy wizard displays.
- 4 If a package exists in your repository with the same name as the EJB JAR display name, indicate whether the Adaptive Server plug-in should prompt you before overwriting existing packages with the new definition.
- 5 Enter the path to the JAR file and click Next.
- 6 The Adaptive Server plug-in creates a new package in the repository that contains a component for each Bean defined in the JAR file, printing status messages and warnings to the Deploy Wizard. The new package has the same name as the EJB JAR display name. If there is no display name, the new package has the same name as the JAR file. For each Bean in the EJB-JAR, EJB Server creates an EJB component with the same name as the *ejb-name* element in the EJB-JAR deployment descriptor.

Home names for imported EJB components

EJB Server sets an imported Bean's home name to the EJB Server default, *package/component*, where *package* is the Adaptive Server plug-in package name, and *component* is the Adaptive Server plug-in component name.

❖ **Installing a package in the Adaptive Server plug-in**

- 1 Double-click the Installed EJB Packages folder. Choose File | Install Existing Packages.
The Select dialog box displays.
- 2 Choose a package from the drop-down list of deployed packages in the repository.
- 3 Click OK.
- 4 Select View | Refresh All.
The package appears in the Installed EJB Packages folder.
- 5 Optionally generate stubs and skeletons for the component Beans. See "Generating stubs and skeletons" on page 86.

Use the status dialog as a to-do list

In the deployment status dialog box, the Adaptive Server plug-in displays warnings for each setting that requires further attention before running the application. You can copy and paste this text to a text editor to use as a to-do list.

You may need to configure the following settings in the Component Properties dialog box before running EJBs:

- For Beans that use container-managed persistence, the settings described in Persistence for entity components
- Resource references, described in Configuring resource references
- EJB references (to components that are not installed with the JAR file), described in Configuring EJB references
- Environment properties, described in Configuring environment properties

You may need to configure the Role mapping settings in the Package Properties dialog box, described in “Role Mapping properties” on page 94.

Importing EJBs from
EJB class files

The Adaptive Server plug-in can import component and method information from Java class files. Use this technique if you have created a Bean’s interfaces and implementation class, but have not created a deployment descriptor. You will need to manually configure properties that would otherwise be read from the deployment descriptor afterwards.

❖ Importing EJB class files

Before importing class files

Verify that the code base under which the class file is deployed is specified in the CLASSPATH environment variable, as inherited by the Adaptive Server plug-in process.

Create a package to contain the component if necessary.

Import EJB class files as follows:

- 1 Specify the package in which to install the component as follows:
 - a Open the Installed EJB Packages folder.
 - b Double-click the package to which the component will be added.
- 2 Double-click the Add new component icon in the right window.

The Component wizard displays.

- 3 In the Type of Component window, select Import an EJB Class File, and click Next.
- 4 Enter the component name and EJB class and interface names as follows:
 - **Component name** The name of the component to be created in the Adaptive Server plug-in, for example, FinanceBean.
 - **Component type** Choose one of the following to match your implementation:

Type	Description
StatelessSessionBean	A stateless session Bean
StatefulSessionBean	A stateful session Bean
EntityBean	An entity Bean with Bean-managed persistence.
 - **Remote interface** The full path to the Java class file that contains the Bean's remote interface.
 - **Home interface** The full path to the Java class file that contains the Bean's home interface.
 - **Bean class name** The full path to the Java class file that contains the Bean's implementation class.
 - **Primary key class** If defining an entity Bean, enter the full path to the Java class file that contains the Bean's remote interface. If defining a session Bean, leave blank.
- 5 Click Next.
- 6 The Adaptive Server plug-in displays the Component Properties dialog box. The Component's type and Java classes have been filled in by the importer. Specify values for the remaining properties before running the Bean.

Installing components

Your component must be installed in a package before it can be run by applications. Components that have the same name but are installed in different packages are different components; modifying or deleting one does not effect the other.

❖ **You can create a new component and install it to a package as follows:**

- 1 Double-click the Installed Packages folder to expand it.
- 2 Double-click the package to which the component will be added.
- 3 Double-click the Add new component icon.
- 4 In the Component Wizard dialog box, select Define New Component, and click Next.
- 5 Enter the component name in the Enter New Component Name dialog box, and click Next.

The Summary Page window displays.

- 6 Verify the component name.

The new component appears in the package's list of installed components. Configure the settings as described in "Configuring component properties" on page 75

Modifying components

❖ **To modify a component:**

- 1 Highlight the component you want to modify.
- 2 From the File menu, select one of the following options:
 - **Properties** Displays the Component Properties window described in "Configuring component properties" on page 75. Make any modifications required, and click OK.
 - **Delete** Removes the component from the package.

Configuring component properties

The Component Properties window allows you to configure the settings that EJB Server uses to load the component and invoke its methods. Component properties are organized on the following tabs:

Tab	Description
General	Defines basic information about the component, including the supported IDL interfaces, the component type, and implementation details.
Transactions	Defines the components transactional properties, such as how the component participates in transactions and whether the component explicitly commits its work.
Instances	Defines how instances of the component are managed, including instance creation, thread binding, and client/component bindings.
Resources	Configures properties that govern the component's use of server and database resources.
Persistence	Specifies the primary key type for EJB entity Beans.
All Properties	Allows you to manually edit component property settings in the EJB Server configuration repository. For advanced users.

General component properties

The General tab defines basic information about the component, including the supported IDL interfaces, the component type, and implementation details. These properties may have already been configured correctly by the import process. The following table describes the window controls.

Table 6-1: General component properties

Property	Description	Notes
Component Type	Specifies the type of the component, which can be: EJB - Stateless Session Bean A stateless session bean EJB component. EJB - Stateful Session Bean A stateful session Bean EJB component. EJB - Entity Bean An entity Bean EJB component.	EJB components must be implemented in accord with version 1.1 of the Enterprise JavaBeans specification.
Comment	Specifies description of the component. The description can be up to 255 characters.	Enter a comment that describes the purpose of the component.
Class	The name of the class that implements the Bean, in Java dot notation.	

Transactions tab component properties

The Transactions tab configures the component's transactional properties. Chapter 5, "Understanding Transactions and Component Lifecycles," provides useful background for the transactional properties.

Transaction attribute values

The transaction attribute determines how methods in your component participate in transactions; the setting affects all methods.

The transaction attribute can have the following values:

- **Not Supported** (The component-level default) The component's methods never execute as part of a transaction. If the component is activated by another component that is executing within a transaction, the new instance's work is performed outside of the existing transaction.
- **Supports** The component can execute in the context of an EJB Server transaction, but a connection is not required in order to execute the component's methods. If the component is instantiated directly by a base client, EJB Server does not begin a transaction. If component A is instantiated by component B, and component B is executing within a transaction, component A executes in the same transaction.

- **Required** The component always executes in a transaction. When the component is instantiated directly by a base client, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A executes within the same transaction; if B is not executing in a transaction, then A executes in a new transaction.
- **Requires New** Whenever the component is instantiated, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A begins a new transaction that is unaffected by the outcome of B's transaction; if B is not executing in a transaction, then A executes in a new transaction.
- **Mandatory** Methods may only be invoked by a client that has an outstanding transaction.
- **Bean Managed** For EJB session Bean components only. The component can explicitly begin, commit, and rollback 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 EJB Server transaction.

Stateless session Beans can use this attribute, but transactions begun in a method must be committed or rolled back before that method returns. Otherwise, EJB Server logs an error and returns an exception to the client. Stateful session Beans can create transactions that remain open across several method calls.

- **Never** The component's methods never execute as part of a transaction, and the component may not be called in the context of a transaction. If a client or another component calls the component with an outstanding transaction, EJB Server throws an exception.

Instances tab component properties

Properties on the Instances tab configure how instances of the component are created and bound to server-side threads and client-side object references. The properties are as follows:

Property	Description
Concurrency	<p>Enabling this option allows multiple method invocations to occur simultaneously. Concurrent access can decrease the response time of client method invocations. Enable this option for any component that is thread safe.</p> <p>If this option is disabled, EJB Server serializes all method calls to the component.</p> <p>Concurrency applies to execution of all instances</p> <hr/> <p>Concurrency option disabled If the Sharing and Bind Thread options are selected, the Concurrency option is implicitly disabled.</p>
Bind Object	<p>Applies to stateful session Beans only. When this property is enabled, an instance is bound to a client's proxy reference until the client destroys or releases the reference.</p> <p>If you enable this option, your component must be thread-safe; that is, one instance must be able to execute on multiple threads concurrently. A client may call the proxy from multiple threads, or pass the proxy to another process or component; consequently, there is no guarantee that calls are serialized with Bind Object enabled.</p> <p>Component instances are destroyed when the client instance reference times out (the time out period is configured on the Instances tab—see “Instances tab component properties” on page 78). Instances are not pooled.</p> <p>Bind Object is most commonly used for storage components, which are used to store a component's state information in a database. See “Persistence tab component properties” on page 82 for more information on storage components.</p>
Bind Thread	<p>When this option is enabled, component instances are bound to the creating thread. Enable this option if the component uses thread-local storage.</p> <p>If the Bind Thread option is selected, multiple instances may still run concurrently on separate threads. To ensure that only one instance is active at a time, make sure that the Concurrency option is not selected.</p> <p>When Bind Thread is enabled, instances are pooled if the Pooling option is enabled. The thread is pooled with the instance in this case.</p>
Pooling	<p>When this option is enabled, component instances are always pooled after deactivation. If you enable the Pooling option in the Adaptive Server plug-in, your component is always pooled, and these methods are not called.</p>

Property	Description
Sharing	<p>When this option is enabled, a single, shared instance of the component services all client requests.</p> <p>A shared component can store data in instance variables. However, if the component’s Concurrency option is also selected, you must add code to synchronize access to instance variables.</p> <hr/> <p>Sharing setting overrides Pooling setting If you select both Sharing and Pooling, Sharing takes precedence.</p> <hr/>
Stateless	For EJB session Beans, the Stateless option is set correctly when the component type is set, and must not be changed.
Transient	For EJB stateful session Beans, this property must be enabled for the standard EJB passivation and activation to occur. It must be disabled if you want to configure a stateful session Bean to support failover using the Persistence tab properties (see “Persistence tab component properties” on page 82).
Reentrant	When this option is enabled, an instance is allowed to participate in loopback call sequences, which are call sequences where one of the Bean’s methods calls another component which in turn calls a method in the calling Bean instance. Most Beans are not implemented to support reentrancy, and you must not enable this option unless the Bean developer has verified that the implementation allows it.

Resources tab component properties

Properties on this tab govern the allocation and deallocation of resources required by the component.

- **Transaction Timeout** A component’s Transaction Timeout property specifies the maximum duration of an EJB Server transaction. See Chapter 5, “Understanding Transactions and Component Lifecycles,” for more information on EJB Server transactions.

The timeout period is configured in seconds, with 0 indicating infinity (that is, no timeout). If the component's Transaction Timeout property is not set, the default is inherited from the server properties. The default for a new server is 0. When specifying timeouts, a resolution of 5 seconds is recommended. EJB Server checks for timeouts after each method returns. Your component will not be deactivated in the middle of an invocation because of a timeout. When a transaction times out, the next method invocation in the client-side ORB throws the `CORBA::TRANSACTION_ROLLEDBACK` system exception.

To set Transaction Timeout for a server, display the All Properties tab in the Server Properties window. Then set the `com.sybase.jaguar.server.tx_timeout` property.

Network transport time is included in the measured timeout period. You may need to configure a larger timeout period if clients connect over slow networks.

- **Instance Timeout** Specifies how long, in seconds, an active component instance can remain idle between method calls before the client's proxy becomes invalid. If the timeout expires, the instance is automatically deactivated. Instance Timeout is useful for ensuring timely deactivation of stateful components. ("Stateful versus stateless components" on page 54 explains this term.) The setting has no effect for stateless components.

When the timeout period is exceeded, EJB Server deactivates the component and invalidates the client's object reference. If the client attempts another method invocation, the client-side ORB throws the `CORBA::OBJECT_NOT_EXIST` exception. At this point, the client must create a new proxy instance for the component.

This property is not set for new components; the component inherits a default value from the server properties. At the server level, configure the instance timeout by displaying the All Properties tab in the Server Properties window. Then set the `com.sybase.jaguar.server.timeout` property.

The timeout period is configured in seconds, with 0 indicating infinity (that is, no timeout). If the component's Instance Timeout property is not set, the default is inherited from the server properties. The default for a new server is 0. When specifying timeouts, a resolution of 5 seconds is recommended.

Network transport time is not included in the measured timeout period. You may need to configure a larger timeout period if clients connect over slow networks.

Persistence tab component properties

The Persistence tab allows you to specify an EJB entity Bean's primary key and configure settings that allow EJB Server to save component state to a database server.

Table 6-2 summarizes the Persistence settings. See Chapter 8, "Managing Persistent Component State," for detailed information on these fields.

Table 6-2: Persistence tab component properties

Field	Description
Persistence	<p>Specifies whether component state is saved, and if so, how. The available options are:</p> <ul style="list-style-type: none"> • None The default. The component's state is not stored in a database. • Java Serialization For EJB stateful Session Beans only. The component implementation class is serialized and deserialized to save and restore component state. • Component Class Your component implementation manages persistence. Used for EJB entity Beans. • Automatic Persistent State EJB Server manages the persistent state of your component.
Primary Key	<p>The primary key for EJB entity Beans. Specify the IDL type of the components primary key. For example: <code>foo:bar:MyPK</code>. Components with a primary key must have a <code>findByPrimaryKey</code> method in their home interface, and can have additional finder methods that allow clients to look up instances that match a desired primary key.</p> <p>Unless you have defined an entity Bean by importing class or EJB-JAR files, you must define the primary key type yourself. For an EJB entity Bean, choose from the types listed in "Allowable primary key types" on page 89.</p>
Storage Component	<p>Specifies the name of a component that reads and writes component state information from a remote database server. Required when using automatic persistence, or when using component-managed persistence with an implementation that delegates to EJB Server's built-in storage component.</p>
Connection Cache	<p>Specifies the connection cache used by the storage component. The cache must be installed on all servers where your component runs and allow by-name access.</p>
Table	<p>Specifies the name of the database table to store component state information. Create the table in the default database rather than in the <code>dbname..table</code>.</p>
Time Out	<p>This setting is reserved for future use.</p>

Field	Description
Time Stamp	<p>When using a mapped database table, specifies the timestamp used for optimistic concurrency control. Specify one of the following:</p> <ul style="list-style-type: none"> • A column name The name of a column in the mapped table that is incremented in each update. By default, EJB Server uses 4-byte integer timestamp. You can also use a 16-byte binary value, but to do so, you must set the <code>com.sybase.jaguar.component.ts.length</code> property to <code>binary(16)</code>. • “None” Enter “none” to disable optimistic concurrency control. This setting is not recommended. • No value If you leave the Time Stamp field blank, EJB Server uses all column values to perform optimistic concurrency control. <p>For best performance, use a 4-byte integer timestamp column. The timestamp column need not be mapped to the component’s persistent state fields. 16-byte binary timestamp values are not usable when other processes (besides EJB Server) update a table.</p>

All Properties tab

The All Properties tab allows you to edit component property settings as they are stored in the EJB Server configuration repository. You can only modify or delete properties that you have added—you cannot modify or delete default properties, such as the Instance Timeout property.

❖ **To add a property:**

- 1 Fill in the Add fields as follows:
 - a Enter the property name in the Name field
 - b Enter the value in the Value field.
- 2 Click Add and then OK.

❖ **To delete a property:**

- 1 Highlight the property you want to delete:
- 2 Click Delete and then OK.

The following component properties can be configured only from the All Properties tab:

com.sybase.jaguar.component.keys For an EJB entity Bean, specifies the name of an IDL typedef for a sequence of the Bean's primary key structures. This type is used when generating the skeleton and implementation classes for the component.

When you manually specify a value for the Primary Key field on the Persistence tab, EJB Server sets this property to `module::componentKeys` where *module* is the module containing the primary key type, and *component* is the component name. The Adaptive Server plug-in defines the type if it does not exist, using the following structure:

```
typedef <sequence pk> componentKeys
```

where *pk* is the primary key type, and *component* is the component name.

Set the `com.sybase.jaguar.component.keys` property only when you have manually defined a sequence that uses another naming convention or that is located in another module.

If you have used PowerJ or the Adaptive Server plug-in import feature to import an entity Bean, the `com.sybase.jaguar.component.keys` typedef may use a different naming convention.

com.sybase.jaguar.component.tx_outcome Determines whether an exception is thrown to the client when a transaction is rolled back. Sybase recommends that you do not alter this setting.

com.sybase.jaguar.component.refresh This property specifies whether the component can be refreshed. If the value is `false`, the File | Refresh option has no effect for the component. Allowable values are `true` and `false`. The default is `true`.

com.sybase.jaguar.component.java.classes For Java components, this property lists additional java classes that must be reloaded when the component is refreshed. The property takes as values a list of fully qualified class names separated by commas. You can specify all classes in a package using wildcards, as in this example:

```
com.xyz.MyPackage.*
```

You can specify all classes in a JAR file by specifying the JAR file name, as in this example:

```
MyEntityBean.jar
```

The JAR file must be deployed in the `/$SYBASE/$SYBASE_EJB/java/classes` subdirectory.

Copies of the specified classes must be deployed under one of the following locations. When loading classes required by Java components, EJB Server searches for classes in this order:

- 1 Any JAR file that is listed in the `com.sybase.jaguar.component.java.classes` property and deployed in the EJB Server `SYBASE/$SYBASE_EJB/java/classes` subdirectory.
- 2 The class tree based at the `SYBASE/$SYBASE_EJB/java/classes` subdirectory.
- 3 The class tree based at the `SYBASE/$SYBASE_EJB/html/classes` subdirectory.

com.sybase.jaguar.component.control Specifies the name of the IDL control interface. The control interface defines methods called by the EJB Server in response to changes in the instance lifecycle. The choices are summarized in this table:

Control Interface	Description
JaguarEJB::EntityBean	For EJB entity Beans.
JaguarEJB::StatefulSessionBean	For EJB stateful session Beans.
JaguarEJB::StatelessSessionBean	For EJB stateless session Beans.

Generating stubs and skeletons

You must generate stubs and skeletons for a Bean before it can run.

❖ **To generate stubs and skeletons for a component Bean:**

- 1 Highlight the component icon.
- 2 Choose File | Generate Stubs/Skeletons
The stubs and skeletons dialog box opens.
- 3 Select Generate Skeleton.
- 4 Enter a code base for the generated files. Sybase recommends the `SYBASE/$SYBASE_EJB/java/classes` subdirectory.
- 5 Click OK.

Creating Enterprise JavaBeans

In most cases it is easiest to define a Bean using one of the import methods described in “Importing Enterprise JavaBeans” on page 71. However, if you prefer editing IDL to Java, you may follow the technique described here.

❖ **Creating a new EJB component from scratch:**

Follow this procedure to create a new EJB component and define the home and remote interface.

1 Double-click the Installed Packages folder that will contain the Enterprise JavaBean.

2 Select the Add new component icon in the right window.

The Component wizard displays.

3 In the Add Component dialog box, select the Define New Component check box and click Next.

4 In the Name of Component dialog box, enter a name for the component and click Next.

5 In the Summary Page, click Finish to create the object.

6 Display the Component Properties dialog box. Make the following changes on the General tab:

a Set the Component Type to correspond to one of the following values:

Component Type	To indicate
EJB - Entity Bean	An entity Bean
EJB - Stateful Session Bean	A stateful session Bean
EJB - Stateless Session Bean	A stateless session Bean

b In the Class Type field, enter the name of the Java class that will implement your Bean, for example, `foo.bar.MyBeanImpl`.

Note The Home Interface Class, Remote Interface Class, and Primary Key Class fields cannot be edited. These fields are set automatically after the Bean’s IDL interfaces and datatypes have been defined. You can change them by changing the component’s IDL interfaces and types in subsequent steps.

- 7 *If you are defining a stateful session Bean*, optionally switch to the Persistence tab and enter a time limit in the Time Out field. This value specifies how long, in seconds, that a client can hold an instance reference without making any calls. If you do not enter a value, or you specify 0, client references do not expire.
- 8 *If you are creating an entity Bean*, specify the primary key as follows:
 - a Define the primary key type as one of the “Allowable primary key types” on page 89.
 - b Click on the Persistence tab, and type the name of the IDL primary key type into the Primary Key field. The Persistence field must be set to Component Class (the default). Leave all other fields besides Persistence and Primary Key blank.
- 9 Click OK to close the Component Properties dialog box.
- 10 The Adaptive Server plug-in has created default home and remote interfaces named *package::componentHome* and *package::component*, respectively, where *package* is the Adaptive Server plug-in package name, and *component* is the component name.
- 11 Edit the home interface methods, following the design patterns described in “Defining home interface methods” on page 89.
- 12 Edit the remote interface methods. See “Defining remote interface methods” on page 91.

Note If portability to other EJB Servers is required, use only in parameters in remote interface methods.

- 13 If creating an entity Bean with container-managed persistence, configure the persistence settings as described in Chapter 8, “Managing Persistent Component State.”
- 14 Optionally configure the transaction properties for each method in the home and remote interfaces, or if all are the same, configure the component’s transaction properties.
- 15 Generate stubs and skeletons for the component as follows:
 - a Highlight the component icon.
 - b Choose File | Generate Stubs/Skeletons.
The stubs and skeletons dialog box displays.
 - c Select Generate Skeletons and click OK.

- d Specify a code base for the generated files.
 - e Click OK.
- 16 The Adaptive Server plug-in generates a template for the Bean implementation class suffixed with *.new*, for example *MyBeanImpl.java.new*. Use this template as the basis for your Java implementation. The Adaptive Server plug-in also generates Java equivalents for the home and remote interfaces, and for an entity Bean, the primary key type.
 - 17 Compile the component source files, and make sure they are correctly deployed to EJB Server. See “Deploying component classes” on page 92.
 - 18 If you are testing the component with a Java applet, generate and compile stubs using the *\$/SYBASE/\$SYBASE_EJB/html/classes* subdirectory as the Java code base.

Allowable primary key types

Define an entity Bean’s primary key as one of the following:

An IDL structure The structure should reflect the primary key for the database relation that the entity Bean represents. In other words, add a field for each column in the primary key. Define the structure to match the intended Java package and class name. For example, if the Java class is to be *foo.bar.PK1*, define a new structure *PK1* in module *foo::bar*.

The name of a serializable Java class Enter the name of a serializable Java class, for example: *foo.bar.MyPK*.

The IDL string type Use string if the key relation has only a string column. In Java, the mapped primary key is *java.lang.String*.

Defining home interface methods

You can add methods to a home interface using a text editor. However, the method signatures in a home interface must follow the design patterns described here to ensure that the generated code works as intended.

Patterns for create methods All Beans can have create methods, which clients call to instantiate proxies for session Beans and insert new data for entity Beans.

Create methods must return the Bean’s IDL remote interface type and raise *CtsComponents::CreateException*. Create methods can take any number of in parameters. To distinguish multiple overloaded create methods, append two underscores and a unique suffix. (This is the standard Java to IDL mapping for overloaded method names. When generating stubs for Java, EJB Server removes the underscores and suffix from the stub method name). The pattern is as shown below:

```
remote-interface create
```

```
(
  in-parameters
) raises (CtsComponents::CreateException);

remote-interface create__suffix1
(
  in-parameters
) raises (CtsComponents::CreateException);
```

Patterns for finder methods Only entity Beans can have finder methods. Clients call finder methods to look up entity instances for existing database rows. Names of finder methods typically have names beginning with *find*.

Every entity Bean must have a `findByPrimaryKey` method that matches the following pattern:

```
remote-interface findByPrimaryKey
(
  in pk-type primaryKey
) raises (CtsComponents::FinderException)
```

where *remote-interface* is the IDL remote interface, and *pk-type* is the IDL type of the primary key.

Entity Beans can have additional finder methods of two types:

- Those that return a single remote interface instance and raise `CtsComponents::FinderException`, as shown in the pattern below:

```
remote-interface findSuffix
(
  in-parameters
) raises (CtsComponents::FinderException)
```

where *remote-interface* is the IDL remote interface, *Suffix* is a name suffix other than *ByPrimaryKey*, and *in-parameters* is a valid parameter list composed solely of in parameters.

- Those that return a sequence of instances whose primary keys match a specified search criteria. The pattern is:

```
componentList findSuffix
(
  in-parameters
) raises (CtsComponents::FinderException)
```

where *component* is the component name, *Suffix* is a name suffix other than *ByPrimaryKey*, and *in-parameters* is a valid parameter list composed solely of in parameters.

Sequence types are automatically generated

The Adaptive Server plug-in creates IDL typedefs defining a sequence of remote interface methods and a sequence of primary keys when you set the Primary Key field on the Persistence tab of the Component Properties dialog box. The type for a sequence of remote interface instances is *componentList* and a sequence of primary keys is *componentKeys*, where *component* is the component name.

Defining remote interface methods

The IDL for your Enterprise Bean's remote interface must define a remove method and the business methods implemented by the Bean.

remove methods are called by clients to delete the database row associated with an entity Bean, and to release a reference to a session Bean instance. remove methods have the following signature:

```
void remove
(
)
raises (::CtsComponents::RemoveException);
```

You can define business methods using a text editor. The procedure is the same as for any other IDL interface.

Note If portability to other EJB Servers is required, use only in parameters in remote interface methods.

❖ To configure EJB 1.1 role references:

- 1 If necessary, define new Adaptive Server roles to be used by callers of the component. You can create roles in the Adaptive Server plug-in in the Roles folder under the Adaptive Server icon. Adaptive Server and EJB Server share roles.
- 2 Verify that J2EE roles are mapped to Adaptive Server roles in the properties of the package where the component is installed; check the Role Mappings tab in the Package Properties window—see “Configuring package properties” on page 94. You must map a J2EE role name for each role to be used in role references.

Deploying component classes

If you are creating components from scratch in the Adaptive Server plug-in to Sybase Central, you must follow the steps in this section to deploy the component class and other classes that it depends on. If you deploy from PowerJ, PowerJ performs these steps for you. If you are using another EJB development tool that can export EJB-JAR files, import the EJB-JAR file as described in “Importing Enterprise JavaBeans” on page 71. If you import an EJB-JAR file that calls EJB Server components that are not implemented in the same JAR file, you must list the stub classes for the called components in the custom class list as described below.

EJB Server supports hot refresh of components by using a Java class loader. This feature speeds the development process by allowing you to deploy new class versions without restarting EJB Server. Repeat the steps below to deploy new versions of your implementation.

❖ **To deploy EJB component classes:**

- 1 Deploy the component class files, stub and skeleton files, and other classes required by the implementation to EJB Server. For example, you may need to copy stubs for user defined types and utility classes that are in your component’s package.

If deploying class files, place each class in their respective `$$SYBASE/$$SYBASE_EJB/java/classes` package subdirectories. If deploying a JAR file, place it in the `$$SYBASE/$$SYBASE_EJB/java/classes` subdirectory.

The preferred code base is `$$SYBASE/$$SYBASE_EJB/java/classes`
For security reasons, it is preferable to deploy Java components to the `$$SYBASE/$$SYBASE_EJB/java/classes` subdirectory or some other directory that is not accessible to HTTP downloads. Deploying to this directory also allows your component to be refreshed, and allows you to deploy classes in JAR files without reconfiguring the server’s CLASSPATH environment variable. If you deploy to another location, make sure it is listed in the server’s CLASSPATH environment variable.

- 2 Use the Adaptive Server plug-in to configure the component’s custom class list, specifying the classes that must be loaded when your component is loaded or reloaded, as described in “The custom class list” on page 93.
- 3 Use the Adaptive Server plug-in to refresh the component by highlighting its icon and choosing View | Refresh All. You can also refresh the component by refreshing the package, application, or server where it is installed.

The custom class list To support component refresh, you must specify the custom class list to be loaded when a component is refreshed in the “com.sybase.jaguar.component.java.classes” on page 85 component property. This property must be set on the Properties tab in the Component Properties dialog box. “com.sybase.jaguar.component.java.classes” on page 85 describes the syntax of this property.

The custom class list for an EJB component must contain these classes:

- These packages:

```
com.sybase.ejb.*; javax.naming.*; javax.naming.spi.*
```

- Stubs for all components that your component calls. If the called component’s classes are loaded in a JAR file, list the JAR file name in the custom class list.
- Other classes that your component loads and passes as parameters or return values for intercomponent calls, or passes to clients as method return values and output parameter values.
- Classes that extend `javax.naming.InitialContext` or other `javax.naming` classes and that are called by your component.

Troubleshooting ClassCastException errors

When calling `javax.naming.InitialContext.lookup`, if you see `NamingContext` exceptions with root-cause exception `ClassCastException`, check for the following errors:

- You are casting to an incorrect type (check the class name of the object returned by lookup).
 - Your component has refresh enabled, and the custom class list does not contain some required classes.
 - Your component has refresh enabled, and calls a component that has refresh disabled or vice-versa.
-

Modifying packages

❖ To modify an existing package:

- 1 Highlight the package you want to modify.

- 2 From the File menu, select one of the following options:
 - **Properties** Displays the Package Properties window described in “Configuring package properties” on page 94. Make any modifications required, and click Ok.
 - **Delete** Removes the package from EJB Server and from the repository.

Configuring package properties

The Package Properties window has three tabs:

- General
- Role Mapping
- All Properties

General tab properties

The following table describes the properties on the General tab.

Table 6-3: Package properties: General tab

Property	Description	Comments/Example
Description	A description of the package. The description can be up to 255 characters.	View or change the description of an existing component or set the description of a new one.

Role Mapping properties

You can map permissions for component methods to roles defined in Adaptive Server. See “Permissions and roles” on page 15 for more information about roles in EJB Server.

If you want to restrict access to a Bean, you must, for each method:

- Include a J2EE role at the method level when you create the Bean.
- Map the J2EE role to an Adaptive Server role in the Role Mapping dialog box.

❖ **To map a J2EE role to an Adaptive Server role:**

- 1 If necessary, define a new Adaptive Server role. See the *Adaptive Server Administration Guide* for instructions.
- 2 Select the Role Mapping tab from the Package Properties window.
- 3 Click Add. Double-click the J2EE role and enter a J2EE role name. You can also enter a description for the role in the provided field.

- 4 Select an Adaptive Server role from the drop-down list. This is the role from which the J2EE role inherits its permissions and members.
- 5 Repeat steps 2 through 4 for each method in the package with an encoded J2EE role.

All Properties settings

The All Properties tab allows you to edit package property settings as they are stored in the EJB Server configuration repository. You can only delete properties that you have added—you cannot delete default properties, such as the `com.sybase.jaguar.package.components` property.

❖ To edit package properties:

- 1 Look for the property name in the list of properties. If it is displayed, highlight the property and click Modify. Otherwise, click Add.
- 2 If adding the property, fill in the Add Property fields as follows:
 - Enter the property name in the Name field
 - Enter the value in the Value field.
- 3 If modifying a property, edit the displayed value in the Modify Property window.

Exporting packages to EJB-JAR files

You can create an EJB-JAR file that contains the Java classes and deployment descriptors for the EJB components installed in an EJB Server package. The JAR file can be deployed to another EJB Server or any EJB-compatible server.

Exporting JAR files requires the Java Development Kit (JDK) version 1.2.2. The EJB Server installation enables the JDK for exporting packages. See the installation guide for your platform for information.

❖ Exporting an EJB-JAR file

- 1 In the Installed Packages folder, highlight the package to export and choose File | Export | EJB 1.1 JAR.
- 2 Enter the path and file name for the new JAR file and click Next.
- 3 The Adaptive Server plug-in creates the JAR file, displaying status messages in the Export wizard.

Creating Enterprise JavaBean Clients

This chapter describes how to implement EJB clients. For general information on implementing Enterprise JavaBeans and EJB clients, please see the EJB Specification, available for download from Sun Microsystems Web site at <http://java.sun.com/products/ejb/docs.html>.

If your site uses PowerJ, please see the PowerJ documentation for information on code generation wizards for EJB clients.

Contents

Topic	Page
Developing an EJB client	97
Generating EJB stubs	98
Instantiating home interface proxies	100
Instantiating remote interface proxies	104
Calling remote interface methods	106
Managing transactions	106
Serializing and deserializing Bean proxies	107

Developing an EJB client

Follow the steps in the table below to create an EJB client:

Step	Action	For more information
1	Generate EJB stubs.	See “Generating EJB stubs” on page 98.
2	Add code to create the initial naming context and instantiate the home interface proxies.	See “Instantiating home interface proxies” on page 100.
3	Add code to instantiate remote interface proxies.	See “Instantiating remote interface proxies” on page 104.
4	Add code to call remote interface methods.	See “Calling remote interface methods” on page 106.

Step	Action	For more information
5	Optionally add code to control transactions and serialize and deserialize instances.	See: <ul style="list-style-type: none"> • “Managing transactions” on page 106 • “Serializing and deserializing Bean proxies” on page 107

Generating EJB stubs

Stub classes act as proxies for an instance of the EJB component. You can generate EJB stubs for components that are implemented in any of the EJB Server supported component models.

❖ **To generate Java source files for stub classes:**

- 1 Highlight a component to generate stubs for all interfaces and types required by a component.
- 2 Select File | Generate Stubs/Skeletons. The Generate Stubs & Skeletons dialog box displays.
- 3 Select the Generate Stubs option and the Generate Java Stubs option. Enter values in the Stubs fields as follows:
 - **Java Version** Choose Java 2.0 if any home interface has finder methods that return `java.util.Collection`.
 - **Java Code Base** Enter the top-level directory path where generated files should be created.

The path must be valid. It can include a drive and as many directories as you want. You can use `%SYBASE%\%SYBASE_EJB%` (Windows NT) or `$SYBASE/$SYBASE_EJB` (UNIX) to specify subdirectories within the EJB Server installation directory, for example:

```
%SYBASE%\%SYBASE_EJB%\html\classes
```

Other variable substitutions or shell aliases such as “~” to indicate your home directory are not allowed.

If you specify a relative path, such as `myclasses`, the path is interpreted relative to the EJB Server `$SYBASE/$SYBASE_EJB/html/classes` directory.

- 4 Click OK.

Java packages

For each IDL module, Java equivalents for all interfaces, types, and exceptions that are defined in the module are generated to a single Java package. The default Java package name is specified by the module's name or its Javadoc package comment.

If the module has a line of this form in the doc comment, stubs are written in the specified Java package:

```
** <!-- javaPackage dotty-package -->
```

where *dotty-package* is the dot-format Java package name.

If the doc comment does not specify a Java package, stubs are generated to a package that matches the IDL module name. For example, stubs for module `foo::bar` are generated in Java package `foo.bar`.

Compiling stubs

For each IDL interface that is assigned to a component, the Adaptive Server plug-in generates a Java interface with the same name as the IDL interface, a stub class that implements that interface, a helper class, and a holder class. For example, for an IDL interface named `Calculator::Calc`, the Adaptive Server plug-in creates the source files listed in the following table:

Table 7-1: Java stub source files for example interface `calc`

File Name	Purpose
<i>Calc.java</i>	Defines an interface with methods equivalent to the component's methods.
<i>Calc_Stub.java</i>	Class that implements the interface.
<i>CalcHolder.java</i>	Used when interface references are passed as an input or output parameter.

The Adaptive Server plug-in creates stubs for each interface and datatype defined in a module. If your component references a module that contains multiple interfaces, you will find that additional stub files are generated besides the stubs for the interfaces that are directly implemented by your component.

Compile the stub classes with a JDK 1.2 compiler. Make sure that the `CLASSPATH` setting contains the code base directory and the EJB Server `html/classes` subdirectory. For example:

```
set CLASSPATH=%SYBASE\SYBASE_EJB%\html\classes;  
%SYBASE\SYBASE_EJB%\java\classes;%SYBASE\SYBASE_EJB%  
javac *.java
```

Instantiating home interface proxies

EJB clients use the Java Naming and Directory Interface (JNDI) to resolve logical Bean home 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 7-2. 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:9000");  
  
    // 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.

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.

Table 7-2: Sybase EJB 1.1 InitialContext Properties

Property name	Description
java.naming.factory.initial	Specifies the fully qualified Java class name of the class that returns <code>javax.naming.InitialContext</code> instances that interact with the naming provider. Use <code>com.sybase.ejb.InitialContextFactory</code> for EJB Server clients.
java.naming.provider.url	Specifies the URL to connect to the EJB name server. Set the value to a URL with the following format: <code>iiop://hostname:iiop-port/initial-context</code> where: <ul style="list-style-type: none"> <code>hostname</code> is the host machine name for the EJB Server that serves as the name server for your application. If omitted, the default is <code>localhost</code>. <code>iiop-port</code> is the IIOP port number for the server. <code>initial-context</code> is the initial naming context. This can be used to set a default prefix for name resolution. For example, if you specify <code>USA/Sybase/</code>, 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 <code>iiop://localhost:9000/</code> .
java.naming.security.principal	Specifies the user name for the EJB Server session. Required if user name/password authentication is enabled for your EJB Server.
java.naming.security.credentials	Specifies the password for the EJB Server session. Required if user name/password authentication is enabled for your EJB Server.
com.sybase.ejb.RetryCount	Specify the number of times to retry when the initial attempt to connect to the server fails. The default is 5.
com.sybase.ejb.RetryDelay	Specify the delay, in milliseconds, between retry attempts when the initial attempt to connect to the server fails. The default is 2000.

Property name	Description
com.sybase.ejb. socketReuseLimit	<p>Specify the number of times that a network connection may be reused to call methods from one server. The default is 0, which indicates no limit. The default is ideal for short-lived clients.</p> <p>In Sybase testing, settings between 10 and 30 proved to be a good starting point. If the reuse limit is too low, client performance degrades.</p>
com.sybase.ejb. GCInterval	<p>Specifies how often the ORB forces deallocation (Java garbage collection) of unused class references. Though this property is set on an individual ORB instance, it affects all ORB instances. The default is 30 seconds. The default is appropriate unless you have set an idle connection timeout of less than 30 seconds. In that case, you should specify a lower value for the garbage collection interval, since connections are only closed while performing garbage collection. In other words, the effective idle connection timeout ranges from the idle connection timeout setting to the smallest integral multiple of the garbage collection interval.</p>
com.sybase.ejb. IdleConnectionTimeout	<p>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 ORB 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.</p>

Resolving Bean home names

Call the `Context.lookup` method to resolve a Bean's home name to a proxy for the Bean's home interface. If the server where the Bean is installed has a name context configured, pass the server's name context as part of the Bean home name, in the format:

Server-name-context / Bean-home

Call `javax.rmi.PortableRemoteObject.narrow` to narrow the returned object to the Bean's home interface class. `narrow` requires as parameters the object to be narrowed and a `java.lang.Class` reference that specifies the interface type to be 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 home 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 home 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 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.

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 "Transactions tab component properties" on page 77 for more information on transaction attributes.
- *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
Statefull 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 `getEJLObject` to restore the proxy, as shown in the example below:

```
String _serializeFrom; // Name of file to read from
Statefull 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 = (statefull) handle.getEJBObject();
    istream.close();
} catch (Exception e)
{
    System.out.println(
        "Deserialization failed. Exception "
        + e.toString());
    e.printStackTrace();
    return;
}
```

Managing Persistent Component State

You can code components to store state information in the database rather than in memory. Doing so offers failover for stateful components, and allows you to map relational data to a component interface using the EJB entity Bean model.

Topic	Page
Persistence for entity Java Beans	109
Persistence for stateful components	114
Storage components	116
Supported Java, IDL, and JDBC/SQL types	116
Table schema for binary storage	117

Persistence for entity Java Beans

Entity components present an object view of relational data to clients; each instance of an entity component maps to a row in a database relation.

Entity components must be EJB entity Beans implemented according to the EJB 1.1 standard (see Chapter 6, “Working with EJB Packages and Components”). You can implement entity components by following these requirements:

- Define a primary key type for the component. See “Allowable primary key types” on page 89 for more information.
- Create a home interface for the component with a `findByPrimaryKey` method and, optionally, additional finder and create methods. See Patterns for finder methods for more information.

For an entity component, you can manage persistence using these techniques:

- **Component managed** In this technique, you implement the code that reads and writes persistent data and maps the relational column values to fields in the implementation class. This model corresponds to the Bean Managed Persistence model required by the EJB 1.1 specification.
- **Automatic persistence** In this technique, EJB Server manages the storage and retrieval of persistent data.
- **Generated class** In this technique, a generated Java class saves and restores component state from a remote database. The Adaptive Server plug-in does not generate such classes, but third-party tool vendors can use this option. The generated class can inherit from or delegate to the component's implementation class to save and restore component state.

Using component-managed persistence

To use component-managed persistence, you must configure the component's persistence properties and implement the required methods from the `EntityBean` interface.

Display the Component Properties window in the Adaptive Server plug-in and configure the following fields on the Persistence tab:

- **Persistence** Choose Component Class.
- **Primary Key** Enter the name of the primary key type (see "Allowable primary key types" on page 89).

In most cases, no other persistence settings are required. You can delegate to EJB Server's built-in storage components rather than implementing your own database access code. If you do so, configure the Storage Component, Connection Cache, and Table fields (see "Storage components" on page 116).

Using automatic persistence

When using automatic persistence, EJB Server manages all interaction with the database. There are two options for database storage when using automatic persistence:

- **Using mapped fields** In the mapped table model, you define a mapping from a database table to fields in your component implementation class. When a write to the database is required, EJB Server reads the field values; after reading new data from the database, EJB Server assigns new field values for each mapped database column. This model corresponds to the Container Managed Persistence model required by the EJB 1.1 specification.
- **Using binary storage** In this model, you define state-accessor methods and an IDL state type. EJB Server calls your state-accessor methods before writing data to the database and after reading from the database. The state data is stored in an encoded binary form. Because the relational data is encoded, this model does not support finder methods other than `findByPrimaryKey`.

Identifying the storage technique

The component uses mapped field storage if the value of the `Table` field on the `Persistence` tab begins with `map:`, for example, `map:MyTable`.

❖ To configure automatic persistence:

- 1 Configure `Persistence` tab properties.
- 2 Specify field to column mapping properties.
- 3 Specify finder-method queries.

Configure Persistence tab properties

Display the Component Properties window in the Adaptive Server plug-in and configure the following fields on the `Persistence` tab:

- **Persistence** Choose Automatic Persistent State.
- **Primary Key** Enter the name of the primary key type (see “Allowable primary key types” on page 89). If you have imported an EJB entity Bean, the primary key has been defined already.
- **Storage Component** Enter or choose the name of the component that manages interaction with the database. See “Storage components” on page 116 for more information.
- **Connection Cache** Enter the name of a JDBC connection cache that connects to the database. The cache must have by-name access enabled and be installed on all servers where your component is installed.

- **Table** If using mapped table fields, enter:

`map: table`

Where *table* is the database table name. If you are using binary storage, simply enter the table name.

- **Time Stamp** When you are using mapped table fields, the Time Stamp setting determines how the server uses optimistic concurrency control to prevent overlapping updates to the same column. Specify:

- **A column name** The name of a column in the mapped table that serves as a timestamp. By default, EJB Server uses a 4-byte integer timestamp and explicitly increments the value with each update.

The timestamp column need not be mapped to the component’s persistent state fields. Other processes that update the mapped table must increment the timestamp with each update.

Specify field to column mapping properties

If the table’s primary key maps to a single field in the implementation class (which must be the same type as the component’s primary key), display the All Properties tab and configure the properties in the following table:

Table 8-1: Mapping database columns to a single field

Property name	Value
<code>mapField:[key]</code> (For single-column keys only.)	The database column name.
<code>com.sybase.jaguar. component.key.field</code>	The name of the component field that the key maps to.

To map database fields to columns, display the All Properties tab and define properties to map database columns to fields in the implementation class, using the name/value patterns listed in the table below:

Table 8-2: Mapping database columns to class fields

For columns of this type	Property name	Value
Key fields (enter one property for each key field)	<code>mapField:field[key]</code> Where <i>field</i> is the name of the field in the implementation class that this key column maps to.	The database column name.

For columns of this type	Property name	Value
Non-key fields (enter one property for each field)	<code>mapField:field</code> Where <i>field</i> is the name of the field that this key column maps to.	The database column name.

You can optionally append a SQL type name to column names, in square brackets. For example, to specify the column type must be `varbinary(1024)`, enter:

```
icon[varbinary(1024)]
```

The type name is used during automatic table creation. This feature is useful when a Java type can map to multiple SQL types; in that case, EJB Server defaults to the type with minimal storage requirements when creating the table. `Varchar` columns default to 100 bytes length, and `varbinary` columns default to 255 bytes. Specify a type name to override the default.

Specify finder-method queries

Each finder method in the component's home interface requires a database query to select a set of primary keys. For example, the `findByPrimaryKey` method selects the key that matches the input parameter. A `findAll` method might return all keys in the table.

EJB Server can correctly infer the query required to execute the `findByPrimaryKey` method. For other finder methods, you must enter properties to specify the query. Display the All Properties tab and define new properties for each finder method. Name each property `mapQuery:method`, where *method* is the finder method name. For the value, enter a query to select primary key values with a filter appropriate for the semantics of the finder method. You can use the following placeholders to represent column and table names and parameter values:

Placeholder	To indicate
[key]	The table's primary key (which can consist of multiple columns).
[table]	The name of the table.

Placeholder	To indicate
<code>@param</code>	Reference the value of parameter <i>param</i> in the finder method's IDL signature. Note If the component was imported from an EJB JAR file, the parameter names will not match those in the original Java implementation. Instead, they are p0, p1, and so forth.
<code>@param.fieldName</code>	If method parameter <i>param</i> is not a simple type, reference the value of field <i>fieldName</i> .

The following are examples of queries using placeholders. This query returns all rows in a table:

```
select [key] from [table]
```

This query uses the value of the *expiryDate* parameter to filter a range of *closingDate* column values:

```
select [key] from [table] where closingDate <
@expiryDate
```

Persistence for stateful components

Stateful components collect client session data over successive client method invocations. Normally, state data is stored in memory using fields in the implementation class.

You can manage persistence using these techniques:

- **Java serialization** This model can be used only in EJB stateful session Beans. To save persistent state, EJB Server serializes the component class instance and saves the binary data to the database.
- **Automatic persistence** In this model, you define a state datatype in IDL or Java and implement component methods to receive state data read from the database and return state data to be written to the database. EJB Server calls your state access methods, and manages interaction with the database.

Using Java serialization

To use Java serialization, configure the following fields on the Persistence Tab in the Component Properties window:

- **Persistence** Choose Java Serialization.
- **Storage Component** Enter or choose the name of the component that manages interaction with the database. See “Storage components” on page 116 for more information.
- **Connection Cache** Enter the name of a JDBC connection cache that connects to the database. The cache must have by-name access enabled and be installed on all servers where your component is installed.
- **Table** Enter the name of a database table where the serialized data is to be stored. EJB Server creates the table if it does not exist.

Using automatic persistence

To use automatic persistence, configure the following properties fields on the Persistence Tab in the Component Properties window:

- **Persistence** Choose Automatic Persistent State.
- **Storage Component** Enter or choose the name of the component that manages interaction with the remote database. See “Storage components” on page 116 for more information.
- **Connection Cache** Enter the name of a JDBC connection cache that connects to the database. The cache must have by-name access enabled and be installed on all servers where your component is installed.
- **Table** Enter the name of a database table where the serialized data is to be stored. EJB Server creates the table if it does not exist.

Storage components

A storage component read and writes component state information from the database server. If your component uses automatic persistence or Java serialization, you must specify the storage component used to interact with the persistent data store. The storage component uses the connection cache and database table identified on the Persistence tab in the Component Properties dialog box.

The storage component options are:

- **CtsComponents/JdbcStorage** Uses a JDBC connection cache to provide persistent storage of component state. This component has the Requires transaction attribute. The component's state is saved in the context of any existing transaction associated with the component.
- **CtsComponents/JdbcStorageReqNew** A copy of the *CtsComponents/JDBCStorage* component that has the RequiresNew transaction attribute. The component's state is saved using a separate transaction from that used to manage any database work performed by the component.

Supported Java, IDL, and JDBC/SQL types

Table 8-3 lists the Java, IDL, and JDBC/SQL types that EJB Server supports for persistent storage using mapped fields. Types on one row are equivalent. The JDBC/SQL column lists the `java.sql.Types` constants that the storage component uses to bind to the database column. When creating tables, ensure that each column's type is compatible with the JDBC/SQL type that corresponds to the mapped Java field.

Table 8-3: Supported Java, IDL, and JDBC datatypes

Java field type	CORBA IDL field type	JDBC/SQL column type
boolean	boolean	TINYINT
char	char	CHAR
byte	octet	TINYINT
short	short	SMALLINT
int	long	INTEGER
long	long long	BIGINT
float	float	REAL

Java field type	CORBA IDL field type	JDBC/SQL column type
double	double	FLOAT
java.lang.String	CtsComponents::StringValue	VARCHAR
byte[]	CtsComponents::BinaryValue	VARBINARY
java.lang.Boolean	CtsComponents::BooleanValue	TINYINT
java.lang.Character	CtsComponents::CharValue	CHAR
java.lang.Byte	CtsComponents::OctetValue	TINYINT
java.lang.Short	CtsComponents::ShortValue	SMALLINT
java.lang.Integer	CtsComponents::LongValue	INTEGER
java.lang.Long	CtsComponents::LongLongValue	BIGINT
java.lang.Float	CtsComponents::FloatValue	REAL
java.lang.Double	CtsComponents::DoubleValue	FLOAT
java.lang.BigDecimal	CtsComponents::DecimalValue	DECIMAL
java.lang.Date	CtsComponents::DateValue	DATE
java.lang.Time	CtsComponents::TimeValue	TIME
java.lang.Timestamp	CtsComponents::TimestampValue	TIMESTAMP
<i>Serializable object</i>	(N/A)	VARBINARY

Values that can be null

If a field can contain nulls, do not use a primitive type. Instead, use the `CtsComponents::TypeValue` IDL type and the equivalent Java object type. For example, rather than `float`, use `CtsComponents::FloatValue` and `java.lang.Float`.

Table schema for binary storage

When using the binary storage technique, the table used by the `JdbcStorage` and `JdbcStorageRegNew` components has this schema:

Column	Data format
ps_key (primary key)	<p>The table's primary key. The column datatype is different for different component primary key types (that is, the IDL or Java type specified in the Primary Key field on the Persistence tab):</p> <ul style="list-style-type: none"> • If the component has no primary key, ps_key must be variable-length binary, 16-byte maximum length. • If the component's key is the IDL string type, ps_key must be variable length character, 255-character maximum length. • If the component uses <i>any other</i> primary key type, including java.lang.String, ps_key must be variable length binary, 255-byte maximum length. <p>This column cannot be null.</p>
ps_size	Integer, cannot be null.
ps_bin1	Variable length binary, 255 bytes maximum length, can be null.
ps_bin2	Variable length binary, 255 bytes maximum length, can be null.
ps_bin3	Variable length binary, 255 bytes maximum length, can be null.
ps_bin4	Variable length binary, 255 bytes maximum length, can be null.
ps_data	Binary large object. This type must be functionally equivalent to a Sybase image type. The JDBC driver used by the specified connection cache must allow access to the ps_data column using the JDBC setBytes and getBytes methods.

Developing Applications with PowerJ and EJB Server

This chapter gives an overview of how to develop distributed, Web, and client/server applications with PowerJ and EJB Server. You'll find information about the objects and code that make up the pieces of an application and the ways PowerJ and EJB Server together can provide business solutions.

Topic	Page
About the development process	119
Building distributed and Web applications that use EJB Server	128
Building client/server applications using JDBC	137
Building Enterprise JavaBeans 1.1 components	141

About the development process

What you can build

This chapter describes scenarios for developing the following kinds of applications:

- Distributed or Web applications using EJB Server in the middle tier
- Client/server applications

Applets, applications, components, and Web server extensions

There are several types of Java programs: applets, applications, components, and Web server extensions. PowerJ generates behind-the-scenes code for each of these types so that you don't have to write as much code for the mechanics of the program. You can concentrate on program-specific code.

Applets An applet is a Java program that requires a host program, such as a browser, to run. An applet is usually part of an HTML page and is downloaded when it is needed. You don't have to deploy applets to individual computers. Because they are downloaded as needed, applets should be small so users don't get impatient waiting for the applet to start.

Because applets are downloaded, they are subject to restrictions so they can't harm the user's system. For example, applets cannot access the local file system and can make connections only back to the server they came from.

Your applet class will extend the standard class `java.applet.Applet`, which provides default implementations for the `init`, `start`, `stop`, and `paint` methods. In your applet source file, PowerJ generates code to override the `init` method.

Applications A **Java application** behaves like any other program. In Java terminology, an application is a Java program that does not require a host server or browser to run. The user sees windows and menus and interacts with controls. The application can connect with a middle-tier or database server.

Since applications must be locally installed rather than downloaded each time they are run, they can be larger than applets. The user's class path environment variable must include the directories containing the application's code files.

A standalone application has a `main` method that runs when the application starts. It includes a main form and can include other forms, including dialog boxes and frames with menus.

Components Components are standardized, reusable pieces of software that are hosted in another program. You can install them in servers that are designed to host components like EJB Server, or include them in Java applications.

Business-logic components consist of methods that implement business rules and other application logic. These components can be included in a client application, but are more typically hosted in a component server like EJB Server in a distributed application.

User-interface components are used in client applications to enhance the user interface. Typically, user-interface components extend standard user interface classes. For example, a custom list box component might provide custom sorting methods, or a text box or check box might have data awareness.

In PowerJ, you can add components to the component palette and include them in your applications. You can put components on your forms, customize them, and use their methods through the Reference Card and drag-and-drop programming. For business-logic components on an EJB Server, when you add a component to PowerJ, a proxy for it is added to the palette. You can then use distributed components just as you would local components.

In PowerJ, it is easy to create JavaBeans components. A Bean might be an encapsulated user-interface component, such as an enhanced button with custom functionality, or it might be a business-logic component that includes methods and events. You can use the class, method, property, and event wizards to create the Java classes for the component. You can also write code that allows the JavaBeans user to modify properties at design time. There are specific standards and conventions for creating Beans.

Web server extensions You can create Web server extensions that follow the Java Servlet API. You can deploy servlets in EJB Server or other Web servers that support Java Servlets. For Web servers that do not directly support the Servlet API, PowerJ provides a DLL that translates from CGI, NSAPI, or ISAPI to the Servlet API.

How you build it

You use PowerJ to build most of the pieces of the applications described here. Some of the activities for building a Java application are listed below. Depending on which application architecture you choose, some of these activities may apply to creating the client and others to creating the server.

- 1 Create a workspace, then create a target, which is the type of program you want to build. A workspace can include several targets, where each target is a part of a larger application.
- 2 Create the user interface by creating one or more forms and adding controls and nonvisual components.
- 3 Optionally, add menus by creating a frame (a type of form), adding a MenuBar object to the frame, and using the Menu editor to design menus.
- 4 Optionally, access database data by adding transaction and query objects to the form. Make the visual components of the form bound to the query object.
- 5 Code application-processing logic. You can place this code in different locations:
 - In events and methods for a form, control, or menu
 - In events and methods for nonvisual components on a form (PowerJ calls these framework classes—they are visible at design time as icons)
 - In classes that you add to your application

The PowerJ Reference Card and drag-and-drop programming make it easy to look up classes, properties, and methods and insert appropriate code in your program.

Creating workspaces, targets, and classes

What you do

First, create a workspace. A workspace contains targets, and targets contain forms and other classes.

Targets

A **target** is an application, applet, class, or collection that you create with PowerJ. Types of targets include:

- Applet
- Standalone Java application
- A set of Java class files
- Servlet
- Enterprise JavaBeans component
- JavaBeans component
- WebApplication
- ZIP, JAR, and CAB archive files

You can run a target program anytime during a PowerJ session. PowerJ builds the target by compiling PowerJ files into Java class files and displaying the program's user interface. Depending on the target type and your current run options, your application may be displayed in the applet viewer, in a Web browser, or as a standalone program.

Not all targets can be run. For example, you cannot run a set of Java class files that make up a class library, because it is not a complete application.

A WebApplication target can be used to tie other targets and files together into a single manageable package. It lets you organize, maintain, and publish all the files of an application for an intranet or the Internet.

Similarly, archive file targets (ZIP, JAR, and CAB) can be used to collect other targets and files into a single archive file.

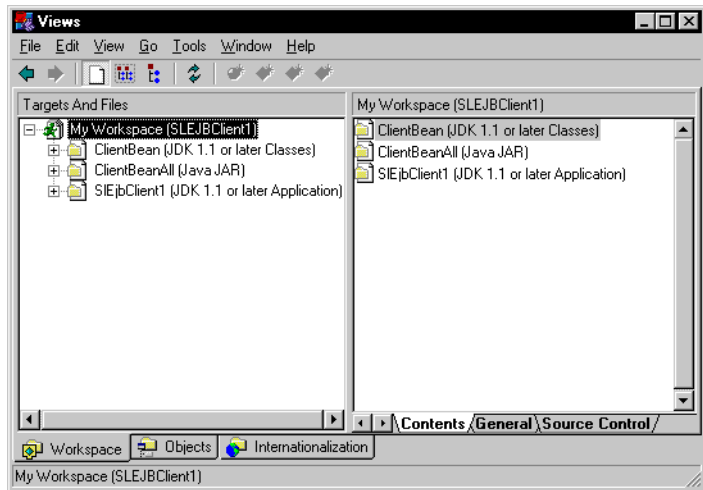
Each target has its own folder; this avoids naming conflicts when source files have the same name.

Workspaces

All work in PowerJ is done within a **workspace**, which is a collection of one or more targets. Your workspace can include all the targets involved in the complete application.

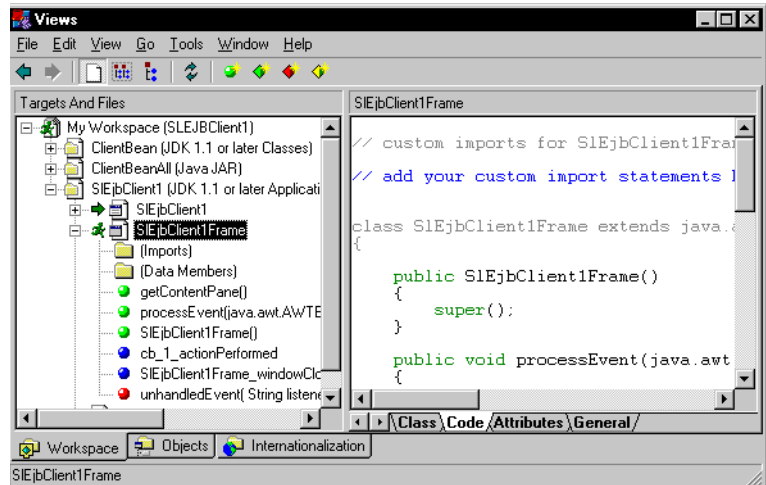
PowerJ creates a workspace definition file for a workspace. The file is a summary of the workspace and lists all the targets that belong to the workspace.

The Workspace view displays the targets in the current workspace. A target's Build Options property sheet lets you specify options for building the target.



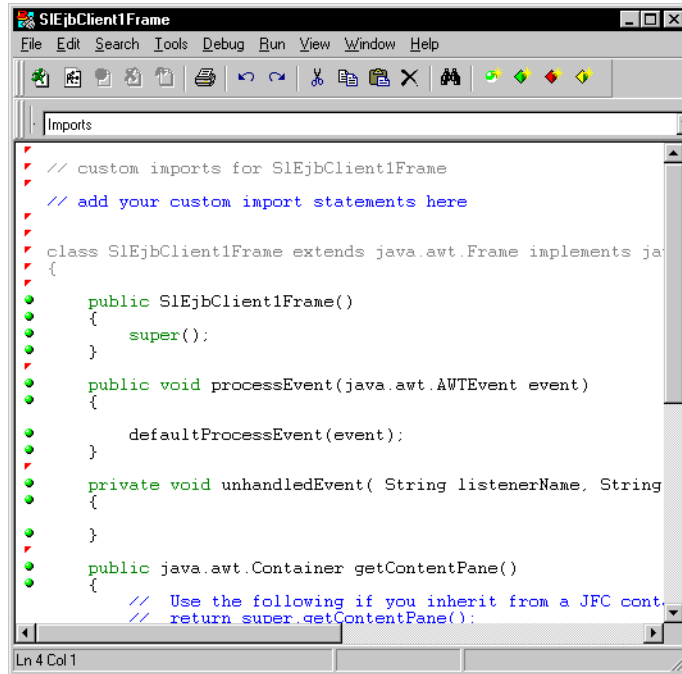
Classes

Each Java file in the target is a Java class. You can expand the targets in the left pane of the Workspace view to show classes as well as the properties, methods, and data members of individual classes. A pop-up menu lets you add new properties, methods, and events. In the right pane, you can further define your classes and attributes, and view and edit Java code.



Writing code

From the Workspace view, you can use the code page or open the code editor for a class or an individual method or event. In the editor options, you can choose to view all the code for the class or one method at a time. You can hide or view the code that PowerJ generates.



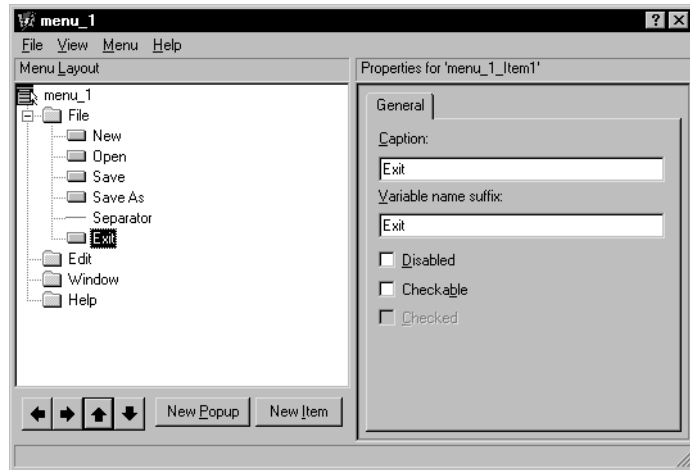
```

SIEjbClient1Frame
File Edit Search Tools Debug Run View Window Help
Imports
// custom imports for SIEjbClient1Frame
// add your custom import statements here
class SIEjbClient1Frame extends java.awt.Frame implements java
{
    public SIEjbClient1Frame()
    {
        super();
    }
    public void processEvent(java.awt.AWTEvent event)
    {
        defaultProcessEvent(event);
    }
    private void unhandledEvent( String listenerName, String
    {
    }
    public java.awt.Container getContentPane()
    {
        // Use the following if you inherit from a JFC cont.
        // return super.getContentPane();
    }
}
Ln 4 Col 1
```


Designing menus

You add menus to your application with a MenuBar object. A **MenuBar object** represents all the menus displayed by the form. MenuBar objects can be added only to forms that are based on the Frame class.

To add a menu to your application, you create a form whose type is Frame and drop a MenuBar object onto the form. You can add individual menus and menu items in the Menu Editor, accessed by selecting Edit Menu from the form's pop-up menu.



Accessing data

In PowerJ, you can use the transaction object to simplify connecting to many types of data sources. You can use either the **DataWindow, Java Edition** (also known as the DataWindow JavaBeans component) or the **query object** for SQL statements and data retrieval. You use the query object if you want to use data-bound controls instead of the DataWindow. Both the DataWindow JavaBeans component and the query object can use the transaction object, or they can process result sets without a database connection.

The Database component palette includes icons for adding transaction and query objects to a form. You create instances of the Transaction and Query classes by selecting them in the palette and dropping their icons onto a form.

To display retrieved data, you can use either data-bound controls with the query object, or the DataWindow JavaBeans component. On the Standard component palette, many of the visual controls (such as check boxes, text boxes, and labels) can be connected to a data source and column so that they display the data in the current row of a query object's result set. You make the connection to the query on the Database page of the control's property sheet.

Coding application logic

What you do

You can add your application logic in event handlers, methods, or external classes. In choosing where to add this code, keep in mind that effective encapsulation will make your classes more reliable and reusable. Don't try to accomplish everything in a single method.

Events

At design time, PowerJ makes it easy for you to write code that gets executed when events are triggered. PowerJ takes care of the infrastructure for events, such as event sources and listeners. You just write event-handler code that you want to be run when the event occurs.

If you want to set up a new event handler in your code, then your code must set up the event infrastructure.

Methods

When you add a new method, PowerJ inserts the method declaration and opens the code window so you can write the method's code.

When writing code, you can use the PowerJ Reference Card to look up methods and properties of classes and components and then insert the method calls into your code.

Classes

In Java, a **class** provides the definition of an object; an object is an instantiation of a class. An application with a user interface includes forms, which are classes with specific support for the design environment. You can also have classes that are not forms. For these classes, you work in the Workspace view and code editor. You add methods to the class to contain the application logic.

You can add a class to a target and use the Workspace view to add properties, methods, and events to the class. This is called a managed class because PowerJ has records of the functions it contains.

You can also add a Java source file to a target. You can edit the source in the code editor, but the Workspace view does not display the methods and properties of the class. The extension for a source file is *.java*.

PowerJ uses an enhanced file format for saving forms and managed classes. Files for forms have the extension *.wxf*; files for managed classes have the extension *.wxc*. When you build your application, PowerJ generates Java source files (with the extension *.java*) that are then compiled into binary files (with the extension *.class*). The binary files contain bytecodes that can be interpreted by a Java VM. These files are usually called class files.

Building distributed and Web applications that use EJB Server

This section describes how to use PowerJ and EJB Server together to create distributed and Web applications.

About EJB Server

What it is

EJB Server is a **component transaction server** that hosts Java components. Using PowerJ, you can develop EJB Server Java components and Java client programs that connect to EJB Server and execute component methods.

When a client invokes a method on an EJB Server component, EJB Server intercepts the call, locates an instance of that component that can carry out the request, passes the parameters and invokes the method, then returns the result to the client.

EJB Server components

An EJB Server component is a reusable module of code that combines related tasks into a well-defined interface. EJB Server components are installed on an EJB Server and contain methods that execute business logic and access data sources. Components are nonvisual—they do not display graphics or user interfaces. You can import an EJB Server component into PowerJ and use the Reference Card to browse its methods and properties.

Data access

To optimize database processing, EJB Server provides support for **connection caching**. Connection caching allows EJB Server components to share pools of preallocated connections to the database server, avoiding the overhead imposed when each instance of a component creates a separate connection. By establishing a connection cache, an EJB Server can reuse connections-made to the same data source.

Architecture of distributed and Web applications

How it works

In a distributed or Web application that uses Java and EJB Server together, a Java client accesses the EJB Server, which in turn accesses a database.

About the distributed architecture

In the distributed architecture, the client is a Java application rather than an applet. The application and Java VM need to be installed on the user's machine and the application files made accessible to the Java VM.

About the Web architecture

A Web application is a variation on the distributed architecture where the client is a Java applet hosted in a Web browser.

About the Web client The browser handles communication with a Web server via the HTTP protocol. The Web page that contains the applet and the applet itself are downloaded via HTTP. The applet then runs in the Web browser but bypasses the Web connection and communicates directly with the EJB Server. Connecting directly to EJB Server enables persistent connections with the client and avoids the problems with stateless HTTP.

A major advantage of the Web architecture is that the client applet is downloaded when the Web page is requested. You don't have to worry about deployment to individual users.

The main disadvantage of the Web architecture is that an applet must be downloaded each time it is run, and unless it is marked as trusted, cannot provide full application services, such as accessing other files, running other programs, or making native calls.

EJB Server as Web server EJB Server can fill the role of Web server using the HTTP protocol, as well as provide support for IIOP connections that invoke the services of the EJB Server transaction server.

Creating a distributed or Web application

The general procedure for using PowerJ to create a distributed or Web Java application that uses EJB Server is:

- 1 Decide what functionality will be encapsulated in an EJB Server component. Typically, the component implements business logic that analyzes data, performs computations, or retrieves and processes data from the database.
- 2 In PowerJ, write the code for the component. In addition to implementing the component's business logic, you may also call EJB Server methods to take advantage of transactions and other EJB Server performance features (information about these features follows).
- 3 In PowerJ or the Adaptive Server plug-in to Sybase Central, define connection caches to manage pools of connections to the remote databases that your component connects to.

- 4 In PowerJ, set the deployment options for your component, including transaction management, instance pooling, and timeout settings.
- 5 In PowerJ, deploy the component to EJB Server. This automatically creates CORBA skeletons for your component, and optionally adds a proxy to the PowerJ component palette.
- 6 In PowerJ, create the client, which can be a Java application or applet. Add the proxy object and PowerJ InitialContext object to your client, and you can access your component's methods as easily as if the component were available locally. The InitialContext object takes care of connecting to your EJB Server component and initializing the proxy.

Information about building client applications and applets is in "Building a Java client for a distributed or Web application" on page 136.

Building EJB Server components with PowerJ

A Java component for EJB Server can be an interface, a class, or a JavaBeans component. You can implement any of these in PowerJ.

An EJB Server Java component follows the Enterprise JavaBeans implementation model. Enterprise JavaBeans (EJB) components are portable to any server that follows Sun's EJB 1.1 specification. An EJB **session Bean** models the interaction between an end user and the EJB Server. For example, in an online purchasing application, a session Bean might keep track of a user's uncommitted purchases. An EJB **entity Bean** represents a row of data stored in a database. For example, an entity Bean might represent a customer's purchase order. From the client's view, entity Beans persist as long as the associated database row has not been deleted. EJB components use standard `javax.ejb` APIs for component lifecycle and transaction control.

The Enterprise JavaBeans model allow you to use the EJB Server connection caching, transaction management, and lifecycle control features.

There are some restrictions to keep in mind for components. They include:

- Parameters for methods in an EJB Server component must have datatypes that can be defined with CORBA IDL.
- Classes and JavaBeans components must have a default constructor (a constructor with zero parameters). When you deploy the component to the EJB Server, PowerJ warns you about classes and methods that don't conform.

Implementing the component

General procedure

Creating and deploying an EJB Server component involves these main tasks:

- 1 Define and implement the EJB component in PowerJ.

Create an EJB 1.1 target that defines the type of Bean, the Java package and class names, and the component's transactional attribute. PowerJ generates skeleton implementations for the `SessionBean` or `EntityBean`'s home interface, remote interface, and implementation class. Provide signatures for business methods in the remote interface and for any required methods in the home interface, and add code to implement these methods in the implementation class.

- 2 Set the deployment options for your component, including the EJB Server package name and EJB Server component name.
- 3 Deploy to EJB Server so you can test the component. This is an iterative process (deploy, test, debug, and redeploy). PowerJ supports in-process debugging of EJB Server components.

For more information about debugging Java components running in EJB Server, see the PowerJ documentation.

EJB Server services

When designing the component, you can take advantage of these EJB Server services to enhance your application's performance:

- Transaction management
- Database access and result set management
- Connection caching
- Instance pooling

These EJB Server features enable you to write high-performance applications with effective error management. You need to set deployment options in PowerJ to enable some of these features in your component, and your component needs to call methods that allow it to cooperate with other components.

Transaction management

How it works

When a component is transactional and uses the EJB Server connection management feature, commands sent to a data source are automatically performed as part of a transaction. Component methods can call EJB Server's transaction state primitives to influence whether EJB Server commits or aborts the current transaction.

The EJB Server coordinates the database activity of all transactional components participating in the transaction. The application can roll back everything that took place in the transaction if any component could not complete its part of the work.

❖ **To define how a component participates in transactions:**

- 1 Choose a **transaction coordinator** for the EJB Server. The transaction coordinator manages the flow of transactions that involve more than one connection and sometimes more than one data source.
- 2 Set the component's **transaction attribute** to determine how the component participates in transactions.
- 3 Code methods to call EJB Server's **transaction state primitives** to influence the transaction outcome.

Each task is described in detail below.

Choosing a transaction coordinator

The transaction coordinator manages the flow of transactions that involve more than one connection and sometimes more than one data source. To enable transactions involving multiple data sources, you must configure your EJB Server to use a transaction coordinator that supports two-phase commit, such as OTS/XA.

Setting the transaction attribute

Each EJB Server component has a transaction attribute that determines how instances of the component participate in transactions. Values are:

Attribute	Description
Requires Transaction	The component always executes in a transaction. Use this option when your component's database activity needs to be coordinated with other components, so that all components participate in the same transaction.

Attribute	Description
Requires New Transaction	Whenever the component is instantiated, a new transaction begins.
Supports Transaction	The component can execute in the context of an EJB Server 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 component's database work is done outside of any transaction.
Not Supported	The component's methods never execute as part of a transaction. If the component is activated by a client that has a pending transaction, the component's work is performed outside the existing transaction.
Mandatory	Component methods must be called in the context of a pending transaction. If a client calls a method without an open transaction, the EJB Server ORB throws an exception.
Bean Managed	For EJB session Beans only. The component can explicitly begin, commit, and rollback new, independent transactions by using the <code>javax.transaction.UserTransaction</code> 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 EJB Server transaction.

Influencing transaction outcome

If your component participates in EJB Server transactions, you can call transaction state primitives to explicitly commit or roll back database updates performed in a method.

Components that use the Bean Managed or OTS Style transaction attribute must explicitly begin and end transactions using the APIs described below. For components that use any other attribute, EJB Server implicitly commits each method's work when the method returns unless the method has requested rollback.

Different component types use different transaction APIs:

- **EJB components using attribute Bean Managed** Only EJB session Beans can use the Bean Managed transaction attribute. Components using this attribute must call the methods in the interface `javax.transaction.UserTransaction` to begin, commit, and roll back transactions. If the component is not a stateful session Bean, then transactions begun in a method call must be committed or rolled back before the method returns. Otherwise, EJB Server logs a runtime error and returns an exception to the client.
- **EJB components using any other attribute** When an EJB component does not use the Bean Managed transaction attribute, EJB Server implicitly commits the component's work after each method returns. To override the default outcome, call the `EJBContext.setRollBackOnly` method.

Database access and result set management

How it works

Java components send result sets to the client using the interfaces in the `com.sybase.jaguar.sql` package:

- Methods in the `JServerResultSetMetaData` interface define the format of rows in a result set.
- Methods in the `JServerResultSet` interface define column values for rows in a result set and send the rows to the client.

You don't have to implement these interfaces. The `jaguar.server.JContext` class contains static methods for obtaining objects that implement these interfaces.

Sending a `ResultSet` object to the client

To send database data to the client:

- Get the data by sending a query to the remote server. Use **`java.sql.Statement`** or one of its extensions. The appropriate method depends on the query you are making.
- Convert the results of the query to **`TabularResults.ResultSet`**.

Sending results row by row

You can also send the result set row by row by building another `ResultSet` object that contains a subset of the original query. Methods in the `metadata` and `resultset` interfaces let you specify the columns and data in the result set.

Connection caching

How it works

A connection cache contains a pool of preallocated connections that components can use repeatedly as needed to communicate with a database server using a common user name and password. Connection caching provides:

- **Improved performance through reuse of connections** The EJB Server connection manager allows client sessions to share previously opened connections so that server CPU time and memory are not consumed by opening more connections than necessary.
- **Improved scalability** Since connection caching allows the same number of clients to be serviced using fewer connections, less memory and other resources are required.
- **Support for transaction semantics** EJB Server's declarative transaction model requires that you call the connection caching APIs to obtain and release all database connections

To realize these benefits, a component must be coded to use a cached connection only when necessary and to release the connection back to the cache at other times. A component should not hold connections while waiting for more input from the client application. As a general rule, each method call that requires a third-tier connection should take a connection handle when invoked and release it before returning.

JDBC 2.0 drivers provide implicit support for connection pooling. When using JDBC drivers that do not conform to the JDBC 2.0 specification, you can define a connection cache in the Adaptive Server plug-in or PowerJ for your components' use.

Transaction support requires cached connections

If your component participates in EJB Server transactions, you must use an JDBC 2.0 driver or define an EJB Server connection cache and obtain connections using the EJB Server connection manager classes.

Java Connection Manager classes

Java components can use the Java Connection Manager (JCM) classes for connection caching. The JCM classes manage JDBC connections.

The JCM classes are:

- **com.sybase.jaguar.jcm.JCMCache** Represents a configured connection cache and provides methods to manage connections in the cache.
- **com.sybase.jaguar.jcm.JCM** Provides access to JDBC connection caches defined in the Adaptive Server plug-in. JCM methods return JCMCache instances.

To access a connection cache, configure a PowerJ transaction object to connect to the cache, and use the transaction object. Typically, you will also use a query or DataStore object.

Instance pooling

How it works

Instance pooling allows EJB Server to maintain a cache of component instances and bind them to client sessions on an as-needed basis. When components support instance pooling, the scalability of your application increases. Instance pooling eliminates execution time and memory consumption that would otherwise be spent allocating unnecessary component instances.

Implicit pooling of EJB components

Entity Beans and stateless session Beans can be implicitly pooled after any method invocation. EJB Server calls the `activate` and `passivate` methods to indicate when an instance has been bound to a client session.

Stateful session Beans remain bound to a client session as long the server has not crashed, the client has not called `remove` to unbind from the instance, or the Beans session timeout value has not expired. EJB Server may serialize the instance and save it to disk to conserve memory. EJB Server calls the `passivate` method before serializing the Bean, and the `activate` method when the Bean has been deserialized. In the code for these methods, you must release and reacquire any object references that do not support serialization.

Building a Java client for a distributed or Web application

Types of clients

The Java client in a EJB Server distributed application can be an applet or an application, depending on whether you want your client to run in a Web browser.

Enterprise JavaBeans model for clients

EJB Server supports the EJB 1.1 client model using stubs that call the EJB Server CORBA ORB.

General procedure

Creating a Java client for EJB Server involves these general steps:

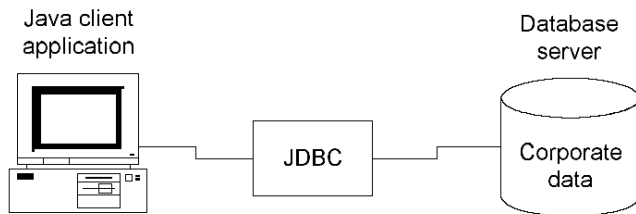
- 1 In PowerJ, add the EJB Server component. This creates a proxy on the PowerJ component palette.
- 2 Add a proxy object and PowerJ InitialContext object to your client.
- 3 Write code that:
 - Calls component methods by calling the corresponding method in the stub class.
 - Cleans up client-side resources by setting proxy references to null. This expedites Java garbage collection.
- 4 Execute the Build command to compile your Java client and then deploy it.

Compiling and deploying the Java client

What you do	In PowerJ, you compile your Java classes with the Build command. Testing the distributed or Web application requires access to the EJB Server, on either your own machine or another machine accessible to you.
Setting up and publishing a client applet	<p>When PowerJ builds an applet, it generates an HTML file with an APPLET tag that you can use for testing. You can copy the APPLET tag from the generated file into the HTML file you are developing for your applet.</p> <p>PowerJ provides a WebApplication target type that you can use to collect all the files that you want to deploy.</p>
Deploying a client application	<p>After you build the client application, you can test it within PowerJ. You can run in Debug or Release (nondebug) mode.</p> <p>When you are ready to deploy the application to a user's machine, you can check the CLASSPATH directories that PowerJ is using so that you can set the CLASSPATH correctly for the users.</p>

Building client/server applications using JDBC

How it works	<p>Client/server architecture means an application that connects directly to a database. Business logic and the user interface are implemented together on the client. Your application may also have business logic as stored procedures in the database.</p> <p>In Java applications, the database connection is made using JDBC.</p>
--------------	---



The Java client application uses a **transaction object** to make a database connection. A **query object** or **DataWindow object** makes the SQL query and manages the result set. To present the data in the user interface, you can use the **DataWindow JavaBeans component** or a **data-bound control**.

The PowerJ Database wizard makes it easy to create database forms and generates much of the required code for instantiating the transaction and query objects and managing the data binding.

About JDBC

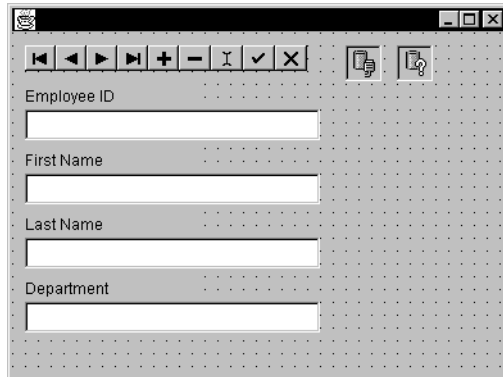
JDBC is a standard that describes how to connect to and talk to a database from within a Java application or applet. JDBC is a set of Java interfaces, not actual Java classes. The JDBC interface (also called the JDBC API) provides Java programmers with a uniform interface to a wide range of relational databases.

To use JDBC, you need a JDBC driver that implements the methods specified in the interface. EJB Server provides the Sybase high-speed, shared-memory driver for this purpose.

Building the application

What you do

PowerJ makes it easy to create a database form. Typically, you use the Form wizard to create the form. The wizard gathers your specification and sets up the transaction object, query object, and bound controls that will display the retrieved data. PowerJ takes care of most of the code for instantiating the objects, making the database connection, executing the query, and populating the controls.



Transaction object

In a PowerJ program, the transaction object handles the database connection. Its properties store information about the database you want to connect to, and it manages SQL transactions via commits and rollbacks.

The transaction object is in the Database page of the component palette. To add a transaction object to your form, you select the transaction icon in the palette and click on the form. If you used the database option in the Form wizard, the transaction object is added automatically.

Transaction properties The transaction object has properties for connection and transaction management. You can set them in the wizard or on the object's property sheet. You need to specify the JDBC driver and the URL for the database, among other properties. You can also specify that you want to connect to the database automatically when the form is created, whether each database operation gets committed automatically when it is completed, and whether updates are allowed.

You can also set properties at runtime. It is typical to let the user specify a user ID and password and set the `UserId` and `Password` properties in code. The following code sets these properties with values saved in string variables:

```
transaction_1.setUserID(userid );
transaction_1.setPassword( password );
```

Connecting If you have not set up `AutoConnect` behavior, you can connect to the database using the `connect` method of the transaction object:

```
transaction_1.connect( );
```

Transaction management If you have not set up `AutoCommit` behavior, you can commit or roll back changes explicitly with methods of the transaction object:

```
transaction_1.commit();
transaction_1.rollback();
```

Query object

A **query object** represents a query on a specific database and can be used to execute any SQL statement. After the transaction object connects to the database, interactions with the database are done through query objects.

How the query object manages data A query object has several data buffers. The buffers store the data as it was retrieved from the database, the current state of the data, rows that have been deleted, and rows that are temporarily filtered out of view.

To update the database, PowerJ generates SQL statements to modify the contents of the database so that they match the contents of the primary buffer.

Query object properties The query object has properties that associate it with a transaction object, describe the SQL statement for the query, and provide information necessary for updating the database, such as primary key columns. You can set the properties in the query object property sheet at design time or you can use query object methods at runtime.

For the SQL statement, you can type the text in the property sheet or use the PowerJ **Query Editor** to construct the query. If you want to change the SQL statement at runtime, you can use the `setSQL` method:

```
String userStatement = "select * from dba.employee";  
query_1.setSQL( userStatement );
```

Executing the query If you set the query's `AutoOpen` property, your program will automatically execute the query when the query object is created. To execute the query at runtime, you call the `open` method:

```
query_1.open();
```

If the query's SQL statement returns a result set, you can call methods that work with the data, often using bound controls.

Setting up data-bound controls

A **data-bound control** is an object whose value is automatically updated by query results. When you use bound controls, you can display database data with very little coding effort. For example, you can bind a text box to a query object so that the text box always shows the value of a specified column in the current row. If you move the cursor to a different row of data, the text box automatically changes to show the value in the same column of the new row.

If the user changes the value of a bound control, it typically changes the corresponding value in the query object's primary buffer. For example, if a text box displays the value of Column 1, changing the value of the text box typically changes the value of Column 1 in the current row (as stored in the primary buffer). Changes made in the primary buffer can be incorporated in the database itself using the `update` method.

A bound control, such as a text box or check box, displays values from a single database column. Controls such as grids can display values from several columns.

To use an object as a bound control, you check the `Bound Control` check box on the object's property sheet at design time. You can't convert an ordinary control to a bound control at runtime.

Data for bound controls To associate data with the bound control, you set the control's `DataSource` and `DataColumns` properties. You can set them on the property sheet at design time or with the `set` methods at runtime.

The `DataSource` property specifies the query object to which the control will be bound. To set its value at runtime, use the control's `setDataSource` method:

```
textf_1.setDataSource( query_1 );
```

The `DataColumns` property determines which column's value is displayed by the bound control. To set its value at runtime, use the `setDataColumns` method. The string identifying the column can be the column's name or number:

```
textf_1.setDataColumns( "emp_id" );
```

To specify multiple columns for controls such as grids, you can list more than one column, with entries separated by semicolons:

```
grid_1.setDataColumns( "emp_id;dept_id" );
```

Data navigator PowerJ's data navigator control provides a simple way for the user to move through a database. It displays forward and back buttons and buttons for adding, editing, and deleting rows.

Data navigators are bound controls, just like the controls that display data. They control the position of the cursor in the result set by calling methods of the query object, such as `moveFirst` or `movePrevious`.

Building Enterprise JavaBeans 1.1 components

PowerJ 3.6 supports Version 1.1 of the Enterprise JavaBeans specification.

See Sun EJB component specs

See the EJB 1.1 specification from Sun Microsystems at <http://java.sun.com/products/ejb/>.

Using Power J, you can automatically deploy EJB 1.1 components to EJB Server. But EJB 1.1 components created in PowerJ can also be imported into other application servers by using that application server's importing methodology.

If you are currently using an application server that does not support EJB 1.1 component technology, either upgrade your current application server to a version in which EJB 1.1 is supported or use EJB Server.

Creating an EJB 1.1 component is done in nearly the same way as described in the PowerJ 3.5 documentation. You use the EJB 1.1 Component wizard to specify the EJB component and the wizard creates an EJB 1.1 target and, optionally, an associated JAR file. The EJB 1.1 target contains the EJB component's implementation class, the Home interface, and the Remote interface.

PART 3

Information for Administrators

This part describes how to set up and manage the EJB Server.

Configuring EJB Server

This chapter describes basic configuration tasks that you can perform to customize your installation, such as replacing an EJB Server, changing server properties, and defining new connection caches.

The EJB Server runtime environment is preconfigured. With minimum setup, you can have a fully functioning EJB Server. Although the default settings are usually sufficient, EJB Server provides you with the flexibility to customize your server environment when necessary.

You can perform all configuration tasks using the Adaptive Server plug-in. For instructions for starting Sybase Central and the Adaptive Server plug-in and for starting and enabling EJB Server, refer to task descriptions in Chapter 2, “Getting Started”.

Topic	Page
Configuring an EJB Server	145
Configuring server stack size	151
Character sets	152
Shared-memory connections	152
Managing connection caches	153
Managing XA resources	159
Configuring listeners	163
Replacing an EJB Server	165

Configuring an EJB Server

To configure or modify the properties of an individual EJB Server:

- 1 From within the Adaptive Server plug-in, display the EJB Server you want to configure by double-clicking the host Adaptive Server icon and then double-clicking the Enterprise JavaBeans folder.
- 2 Highlight the EJB Server.

- 3 Select File | Properties. You see the Server Properties window, which contains these tabs:
 - General – define general individual server properties.
 - Naming Service – set the EJB Server naming service options.
 - All Properties – edit server property settings in their raw format, that is, as they are stored in the configuration repository.

Saving property changes and refreshing the server:

If you modify any property, click OK in the Server Properties window to save your changes, or click Cancel to disregard the changes.

When you modify server properties you must refresh the server for the changes to take effect. To refresh the server, select View | Refresh All.

General

Table 10-1 describes the general properties that you can configure for individual servers.

Table 10-1: Server general properties

Property	Description	Comments
Charset	Specify the character set used by the server. Make sure that the character set in the EJB Server is the same as that used in Adaptive Server.	By default, the server uses <i>iso_1</i> .
Description	Enter a description of the server, up to 255 characters in length.	
Classpath	Displays the contents of the server's <i>CLASSPATH</i> environment variable. This setting specifies the directories from which Java class files can be loaded. It is defined by the start-up script when you start the server.	The field is read-only and helpful for debugging various errors.

Log/Trace

Tracing provides information about activities carried out by your application. Trace output is sent to the EJB Server log file. To establish the level of detail for logging and tracing, select the Log/Trace tab. Table 10-2 describes the logging and trace properties.

Table 10-2: Debug/Trace properties

Property	Description
Log File Name	The name of the EJB Server log file. This file defaults to <i>srv.log</i> in the Adaptive Server startup directory. <i>srv.log</i> logs a wide range of information and is helpful in isolating problems. You can create the log file in an alternate directory by prefixing a full path to the file name you enter.
Log File Size (Bytes)	The size, in bytes, to which the log file grows before it is truncated.
Truncate Log on Startup	When this flag is set, the log truncates every time the server is restarted. Keep in mind that if the server crashes and this flag is set, you will lose the log file and the information it contains.
Trace Attentions	If set, traces attentions received or acknowledged by EJB Server.
Trace Network Driver APIs	If set, traces Net-Lib driver requests.
Trace Network Driver Requests	If set, traces network layer protocol requests.

Naming Service

Select the Naming Service tab on the Server Properties window to set the EJB Server’s naming service options. You can use this property sheet to configure an EJB Server to be a name server.

Initial Context

Enter the EJB Server default name context. The name server binds any object implementations on the server to the server’s initial name context.

If you use a EJB Server as a name server, the name context can be a compound name with each organization level separated with a forward slash (“/”); for example, */us/sybase/finance*.

Naming Server

Use these options to specify whether the EJB Server is also a name server and whether to enable heartbeat detection.

If a server is not accepting connections, the name server does not return a profile (host:port) information to the client. The name server also detects when a failed server is ready to accept connections again and starts routing client requests to that server.

- Click Enable as a Name Server to configure the EJB Server as a name server. If you select this option, you can then set the other Naming Service options described below.

Naming Server Strategy

If you enabled the EJB Server as a name server, indicate whether the server provides transient or persistent object name storage. By itself, an EJB Server name server provides transient storage. However, you can add persistent storage capabilities to EJB Server by using an external naming service, such as an LDAP name server.

If you enable persistent storage, enter the following information:

- The URL of the LDAP name server
- A manager DN (distinguished name) for the LDAP server
- The manager DN password

The manager DN provides exclusive access to all objects in the LDAP server database to bind and update the objects on the name server. The manager DN and its password are part of the LDAP server configuration properties set by the server administrator. Refer to your LDAP server documentation for complete information.

All Properties

For advanced users only. Select this tab to edit server property settings in the EJB Server configuration repository. You can use this tab to edit any property prefixed with “com.sybase.jaguar.server.”

Most server properties can be configured on other tabs in the Server Properties dialog box, except the following:

- **com.sybase.jaguar.server.authservice** The name of a custom component that authenticates IIOP user connections. The default is AseAuth/DbAuth.

- **com.sybase.jaguar.server.authorization.service** The name of a custom component that authorizes user access to components and HTTP URLs.
- **com.sybase.jaguar.server.authorization.permcachetimeout** The length of time, in seconds, that the server can cache authorization data for a user's access to a resource. The default is 7200 seconds, which is equivalent to 2 hours.
- **com.sybase.jaguar.server.jvm.debugging** Whether in-process Java debugging is enabled for servlets and Java components. Set to true to enable debugging (you must also start the debug version of the EJB Server).
- **com.sybase.jaguar.server.jvm.nojit** Specifies whether the Java Virtual Machine just-in-time (JIT) compilation feature is disabled. Set the value to true (the default) to disable the JIT feature.
- **com.sybase.jaguar.server.jvm.options** Specifies initialization options for the Java Virtual Machine. You can specify any option that is valid for the java command line. Separate options with commas, for example:

`-Dmy.system.property.1=foo,-Dmy.system.property.2=bar`

- **com.sybase.jaguar.server.jvm.verbose** Specifies whether the Java class loader should write information about each class loaded to the server log. The default is false, which indicates that class loader logging is disabled.
- **com.sybase.jaguar.server.jvm.verboseGC** Specifies whether the Java garbage collector should write information about each that is destroyed to the server log. The default is false, which indicates that garbage collector logging is disabled
- **com.sybase.jaguar.server.roleservice** The name of a custom component that evaluates user's role membership to control access to components and HTTP URLs. The default value is AseAuth/DbAuth.
- **com.sybase.jaguar.server.services** A list of components that run as service components in the server.
- **com.sybase.jaguar.server.timeout** Specifies the default instance timeout for stateful components running in the server.
- **com.sybase.jaguar.server.tx_timeout** Specifies the default transaction timeout for components running in the server.

See "Configuring server stack size" on page 151 for information about setting server stack size using the `com.sybase.jaguar.server.stacksize` parameter.

Configuring server stack size

Your EJB Server has a stack size property that determines the amount of memory reserved for the call stack associated with each thread created by the server. EJB Server runs each client request on a different thread, so the stack size is the dominant factor in determining how many client requests can be served simultaneously.

The default stack size is 256 K. This is appropriate for almost all situations, and provides adequate reserve memory for the worst case loads that have been tested by Sybase engineering and customers.

For production servers that see heavy use from large numbers of clients, you may wish to decrease the stack size from the default value. However, you must ensure that the stack size is adequate for the components running on the server. If the stack size is too small, your server may experience thread stack overflow errors (these are recorded in the server log).

Warning! Under no circumstances should you reduce the stack size below 64K. If you reduce the stack size, test your server thoroughly under worst-case client loads and check the log for stack overflow errors.

❖ Configuring stack size for servers

- 1 Highlight the icon for the EJB Server and select File | Server Properties.
- 2 Display the All Properties tab. Scroll down if necessary to the `com.sybase.jaguar.server.stacksize` property. Server properties are listed in alphabetical order.
- 3 Enter a stack size in the Value field, specified in bytes as a decimal number. (The field will display with no value if you have not specified a value before. This means the default setting is in effect.)
- 4 Stop and restart the EJB Server.

Character sets

EJB Server and Adaptive Server must have identical character sets. If you change the value of the Adaptive Server character set, you must also change the value of the EJB Server character set. You can set the EJB Server character set in the All Properties tab of the EJB Server properties sheet. See “All Properties” on page 149.

Shared-memory connections

Adaptive Server uses the value of the number of user connections configuration parameter to establish the number of shared-memory connections for EJB Server. Thus, if number of user connections is 30, Adaptive Server establishes 10 shared-memory connections for EJB Server. Shared-memory connections are not a subset of user connections, and are not subtracted from the number of user connections.

To increase the number of user connections for shared memory, you must:

- 1 Increase number of user connections to a number one-third of which is the number of desired shared-memory connections.
- 2 Reboot Adaptive Server.

Although number of user connections is a dynamic configuration parameter, you must restart the server to change the number of user connections for shared memory. See the *System Administration Guide* for more information.

Managing connection caches

A connection cache maintains a pool of available connections that EJB Server components use to interact with the data server. You must configure connection caches for the specific user/database combinations used by your components. A connection cache entry improves performance by eliminating the overhead associated with setting up a connection when one is required.

Note You must install caches in an EJB Server before components in that server can access the cache. You must refresh the cache or refresh the server using View | Refresh All, or restart the server before any changes to the list of installed caches or to cache properties take effect.

Creating and installing a new connection cache

To create a new connection cache and add it to an EJB Server:

- 1 Double-click the EJB Server icon.
- 2 Double-click the Installed Connection Caches folder.
- 3 Double-click the Add new connection cache icon in the right side of the window.
- 4 Follow directions in the Add Connection Cache wizard. You will enter:
 - The connection cache name
 - An optional description of the cache
 - The JDBC driver name
 - The server name, which is the URL appropriate for JDBC calls
 - The user name for the cache
 - The user password for the cache
- 5 Configure the connection cache properties as described in “General tab connection cache properties” on page 155.

Configured connection cache entries appear on the right side of the window of the Adaptive Server plug-in whenever you highlight the Installed Connection Cache folder on the left side of the window.

Modifying connection caches

To view or modify a connection cache entry:

- 1 Expand the Installed Connection Cache folder.
- 2 Highlight the connection cache you want to modify.
- 3 From the File menu, select one of the following options:
 - Properties – view or modify this connection cache’s properties. See “General tab connection cache properties” on page 155.
 - Delete – removes the connection cache from the server.

Modifying connection cache properties

To modify the properties of a connection cache:

- 1 Double-click the EJB Server for which you want to modify connection cache properties.
- 2 Click on the Installed Connection Cache folder.
- 3 Highlight the connection cache you want to modify.
- 4 Select File | Properties. You see the Connection Cache Properties window, which contains these tabs:
 - General – define general server properties.
 - Advanced – edit server property settings in their raw format, that is, as they are stored in the configuration repository.

You must use the cache properties file to manually configure the additional properties described in “Other cache settings” on page 156.

After you have configured a connection cache, click OK to save your changes, or click Cancel to disregard them. You must refresh a newly installed cache for any changes to take effect, and you should test the connection with Ping before trying to access it from components. These operations are described in detail below.

You cannot define two distinct caches that use identical values for server, user, password, and JDBC driver. If two caches are defined with matching values for these settings, and your application requests one, EJB Server returns the first match that is found.

Saving property changes and refreshing the server:

If you modify any property, click OK or Apply in the Connection Cache Properties sheet to save your changes, or click Cancel to disregard the changes.

When you modify server properties you must refresh the server for the changes to take effect. To refresh the server, highlight the server icon and select View | Refresh All.

General

Select the General tab on the Connection Cache Properties window to set the basic connection cache options described in Table 10-3.

From the General tab window, you can test the cache configuration to verify that connections can be made using the options you supply. See “Connection cache ping” on page 158 for more information.

Table 10-3: General tab connection cache properties

Property	Description	Comments/Example
Connection Cache Name	The name for this cache configuration.	Connection cache names are limited to one word, which can contain letters, numbers, and underscores. Names are case-sensitive. You cannot modify the name of an existing connection cache.
Description for the cache	The description of the connection cache section.	The description is a string of up to 255 characters.
Server Name	The URL appropriate for use in JDBC calls.	For the Sybase shared-memory JDBC driver, use: <pre>jdbc:sybase:shm:null:0</pre> where host name = null and port = 0 (zero), because the connection is made through shared memory and not through the network.
Driver name	Set the driver name and properties using the Driver tab on the General window. Your choice for library type is: <ul style="list-style-type: none"> JDBC – for connections using the Sybase shared-memory JDBC driver. 	The names for each of the cache types are: <ul style="list-style-type: none"> For NT platforms: JDBC – the Java class name for the driver class. For example, the Sybase jConnect 5.2 driver requires <i>com.sybase.jdbc2.jdbc.SybConnectionPoolDataSource</i>. For UNIX platforms: JDBC – the Java class name for the driver class; for example, <i>com.sybase.jdbc2.jdbc.SybDriver</i>.
User Name	The user name for this cache.	The name used (along with a password) to connect to the database identified by the server entry.

Property	Description	Comments/Example
Password	The password for this cache.	The password used in connection with a user name to connect to the database identified by the server entry. Passwords are encrypted in the EJB Server configuration file. The Adaptive Server plug-in does not display passwords for existing caches. If you need to change a password, enter the new password and click OK.

Advanced

Select the Advanced tab on the Connection Cache Properties window to set the cache options described in Table 10-4

Table 10-4: Advanced tab connection cache properties

Property	Description	Comments/Example
Enable cache-by-name access	Select this option to allow retrieval of a database connection using the connection cache name instead of requiring a user name and password.	By default, a cache cannot be retrieved by its name. You must be logged in as sa to update the cache's properties to allow the cache to be retrieved by name. Cache-by-name is less secure than requiring a user name and password.
Enable connection sanity check	Whether connections should be verified before releasing them into the cache.	Components may release a connection that is not ready for use by another component. For example, there may be unretrieved results on the connection. Enabling this option causes EJB Server to test whether the connection is usable before replacing it in the cache. Disabling the option increases performance, but may complicate debugging.
Number of Connections in Cache	The number of connections in the pool.	After a connection is released, it is returned to the pool. The default value is 10. You can increase this number if performance suffers due to an insufficient number of available connections.
Service Name	The name of the Adaptive Server to which the Sybase shared-memory JDBC driver connects.	Service name is ignored for caches that use JDBC drivers other than jConnect.™

Other cache settings

The cache settings described in this section can not be set in the Adaptive Server plug-in. You must edit the underlying configuration file to change them. Use a text editor to edit the cache's property file located in the `$$SYBASE/$SYBASE_EJB/Repository/ConnCache` subdirectory. The file is `CacheName.props`, where `CacheName` represents the cache name as displayed in the Adaptive Server plug-in.

JDBC connection properties

For a JDBC connection cache, these connection properties allow you to specify settings beyond those shown in the Connection Cache Properties dialog box. Different JDBC drivers recognize different sets of properties.

For the Sybase high-speed, shared-memory JDBC driver, define cache properties in this form:

```
jdbc:sybase:shm:null:0
```

Any property whose name does not begin with `com.sybase.jaguar` is passed to the JDBC driver as a connection property. For example:

```
PACKETSIZE=2048
```

If a property setting conflicts with a setting in the Connection Cache Properties dialog box, the dialog box setting takes precedence.

Enabling set-proxy support

Adaptive Server Enterprise allows a user to assume the identity and privileges of another user. This feature can be used with any database that recognizes the command:

```
set session authorization "login-name"
```

When proxy support is enabled, connections retrieved from the cache are set to act as a proxy for the username associated with the EJB Server client. To set-proxy to another user name, use the Java `JCMCache.getProxyConnection()` method in your component.

Set-proxy support must be enabled in the cache properties file before components can take advantage of it. To enable set-proxy support, add the following line to the cache properties file:

```
com.sybase.jaguar.conncache.ssa=true
```

To disable support, delete this line or change `true` to `false`.

Connection cache refresh

If you have just installed the cache in a server or modified an installed cache, refresh the server or the connection cache before you attempt to test the cache. You can refresh as follows:

- To refresh the cache:

- a Highlight the Installed Connection Caches folder under the server icon where the cache is installed.
 - b Select View | Refresh Folder.
- To refresh the server, highlight the server icon where the cache is installed, then choose View | Refresh All. All caches installed in the server will be refreshed.

Refreshing a cache may affect running components that are using the cache, specifically:

- If you change the connectivity library setting, cache references held by components become invalid. Attempts to retrieve connections or query cache properties will cause errors. In this case, the component must retrieve a new cache handle.
- If you change other properties, such as user name, password, server name, or the number of connections in a cache, cache references remain valid, but components may be affected by the changed settings. For example, if you change the server name, connections retrieved after the cache has been refreshed will go to the server indicated by the new name.

Connection cache ping

This feature allows you to test the cache configuration to verify that connections can be made using the supplied parameters. To ping, the connection must be installed in the server that the Adaptive Server plug-in is connected to. If you have just installed the cache or changed any settings, refresh the cache before testing it.

To test the cache with Ping:

- 1 Open the Installed Connection Cache folder under the EJB Server icon where the cache is installed.
- 2 Right-click on the icon of the cache you want to ping.
- 3 Choose File | Properties.
- 4 In the General tab of the Connection Cache Properties dialog, click Ping.
- 5 The Adaptive Server plug-in reports whether the connection attempt succeeded.

If Ping fails, check the message text for a description of the problem. The server log file may contain additional information about the cause of the error.

If you change the cache properties to correct the problem, you must refresh the cache before testing again.

Managing XA resources

EJB Server uses the two-phase commit protocol for distributed transactions. To use this feature, your Adaptive Server installation must have a valid ASE_DTM license.

You can use the Adaptive Server plug-in to:

- Enable the OTS/XA feature for EJB Server
- Create XA connection resources for accessing Adaptive Server

Setting up XA resources

This section describes procedures for configuring and enabling XA resources on Adaptive Server and EJB Server.

❖ To configure Adaptive Server and EJB Server for XA resources:

You can also perform steps 2 and 4 from the Adaptive Server plug-in.

- 1 Make a copy of the `$$SYBASE/$SYBASE_EJB/config/afconfig.dat` file and move the copy to a secure location. If the XA configuration process fails, you will need to copy an uncorrupted version of this file back into the release area before reconfiguring XA resources.
- 2 Enable Distributed Transaction Management (DTM) on Adaptive Server:


```
sp_configure 'enable dtm', 1
```
- 3 Run the script `sqlserver12.sql` located in `$$SYBASE/$SYBASE_EJB/html/classes/sp`.
- 4 Grant the `dtm_tm_role` system role to the user in Adaptive Server:


```
sp_role 'grant', dtm_tm_role, user_name
```
- 5 Create the OTS/XA transaction log device in `$$SYBASE/$SYBASE_EJB`:
 - From the UNIX command shell, execute:

```
echo x | dd seek=8k of=server_nameOTSLog.dev
```

- From the DOS prompt on Windows NT, execute:

```
Filevol server_nameOTSLog.dev 400K
```

where *server_name* is the physical name of your EJB Server. If you do not create a log file, EJB Server will not start up.

- 6 Shutdown and restart Adaptive Server.

❖ **To enable EJB Server for OTS/XA transactions:**

Enable the EJB Server for OTS/XA transactions from the Adaptive Server plug-in:

- 1 Highlight the EJB Server folder.
- 2 Select File | Properties.
- 3 Select the Transactions tab.
- 4 Select OTS/XA Transactions.
- 5 Press OK.

Creating XA resources

You must configure XA resources to access a specific database. XA resources differ from connection caches in that XA resources are XA-Library interfaces that maintain their own connection pool separate from the connection cache connection pool.

If a get connection call (such as the Java `getConnection` method call) is in a transaction, the XA resource is automatically used to return a connection. If a get connection call is not in a transaction, the connection cache is automatically used. If the transactional behavior for a component uses the Supported option, then EJB Server determines at runtime whether the component executes its get connection calls in a transaction; if it does, you must configure both a connection cache and a corresponding XA resource for a database.

If you execute a transaction without an XA resource configured for a database, the EJB Server connection manager returns `CS_FAIL`.

By default, EJB Server uses the XA resource library for the JDBC connection: `com.sybase.jdbc2.jdbc.SybaseXADataSource`. You can also use these shared libraries or DLLs to obtain an XA resource that is exported from the database connection libraries:

Connection library	Shared library	DLL
Sybase Client Library 11.0	libjxa.so	libjxa.dll
Oracle OCI 7.x	libclntsh.so	xa73.dll
Oracle OCI 8.x	libclntsh.so	xa80.dll

To change the shared library or DLL, edit the connection cache properties file `%SYBASE%\%SYBASE_EJB%Repository\ConnCache\<cache_name>.props`. For example, to instruct EJB Server to use `oraclient8.dll` instead of `xa80.dll` for Oracle OCI 8.1.x, add this line to the connection cache properties file:

```
com.sybase.jaguar.conncache.xadllname = oraclient8.dll
```

Note You must install XA resources in an EJB Server before components in that server can access the XA resources. You must refresh XA resources or refresh the server using `View | Refresh All` in the Adaptive Server plug-in, or restart the server before any changes to the list of installed XA resources or to XA resource properties take effect.

If a configured XA resource is not running or cannot be connected to, the EJB Server cannot initialize. Copy an uncorrupted version of the `SYBASE/SYBASE_EJB/config/afconfig.dat` file back into the release area and reconfiguring XA resources. See “To enable EJB Server for OTS/XA transactions:” on page 160.

❖ **To create OTS/XA transactions for XA resources:**

See Table 10-5 for a description of the XA properties you enter when you create an OTS/XA transaction resource.

- 1 Double-click the XA Resources folder.
- 2 Double-click the Add an XA resource icon in the right side of the window. The Add an XA Resource wizard displays.
- 3 Enter a name and description of the XA resource. Press Next.
- 4 Enter the server name, a user name, and a password in the Database Connection Information window. Press Next.
- 5 Enter a dll or class name in the Connectivity Information window. Press Next.

- 6 Enter a database name, a default string, an open string, and a close string in the XA Driver Information window.

Note If the Open String is set incorrectly, the EJB Server does not initialize.

Press Finish.

Table 10-5: XA resource properties

Property	Description
Name	A name for the XA resource.
Description	A brief phrase describing the purpose of the XA resource.
Server Name	Name of the XA resource server for shared memory. Enter: <code>NetworkProtocol=shm:Server=null:Port=0</code>
User Name	A name you can use to access the server.
Password	The password for the user.
DLL or Class Name	The file name of the XA resource library. Enter: <code>com.sybase.jdbc2.jdbc.SybXADataSource</code>
Database Name	If you selected CT-LIB, OCI 7.x, or 8.x, specify the database name.
Default String	The string used to connect to the XA resource. You cannot modify this string, which is automatically built from the information that you entered in the previous tabs.
Open String	In this optional field, you can specify any valid open string options. For example, for a Sybase Client-Library 11.0 XA resource, you can enter: -L <i>logfile</i> where <i>logfile</i> is where you want to store log information.
Close String	In this optional field, you can specify a value used by the resource to close a connection.

See your XA resource documentation for more information about the Open Suffix and Close String syntax

Configuring Listeners

A listener is an EJB Server port that communicates to clients using various protocols. Supported protocols are IIOP, TDS, and HTTP.

This section describes the tasks required to configure listeners. You can:

- Create a new listener.
- Modify listener settings.

Preconfigured listeners

EJB Server comes with preconfigured listeners for all protocols. The default host for these listeners is specified at installation. You can also modify port number settings for the preconfigured listeners. For more information, see “Modifying an existing listener” on page 164.

Listener failover

If a server cannot retrieve listener information from the repository for an IIOP listener or if an IIOP listener has not been configured, the server attempts to open a listener at this address:

```
IIOP: localhost, 9000
```

Listener start-up can fail if a port is already in use. You can verify the listener addresses in use by viewing the initial log entries in the *srv.log* file.

Configuring listeners

This section describes how to create, modify, and delete a listener. All of the configuration tasks require you to first access the Listeners folder from the Adaptive Server plug-in:

- 1 Double-click the Adaptive Server icon.
- 2 Double-click the Enterprise Java Beans folder.
- 3 Double-click the EJB Server folder.
- 4 Click the Listeners folder on the right side of the window.

❖ **Creating a new listener**

- 1 Double-click the Add new listener icon.
The Add new listener wizard displays.
- 2 Add the name, host name, and port number for the listener in the Name and Description window. Press Next. See Table 10-6
- 3 Select the iiop, http, or tds protocol from the drop-down menu on the Select Type of Protocol window. Press Next.
- 4 Verify the new listener on the Summary Page Window. Press Finish.
The new listener appears on the right side of the window.

❖ **Modifying an existing listener**

- 1 Highlight the listener you want to modify.
- 2 Select File | Properties.
- 3 Make your modifications and click OK. Listener properties are described in Table 10-6.

❖ **Deleting a listener**

- 1 Highlight the listener you want to delete.
- 2 Select File | Delete.

Table 10-6: Listener profile properties

Property	Description	Comments/example
Protocol	Select the protocol from the drop-down list: <ul style="list-style-type: none"> • IIOP • HTTP • TDS 	TDS, IIOP, and HTTP do not provide encryption. TDS and IIOP provide user name and password-based authentication.
Host name	The name or IP address of the EJB Server host to which the listener is being assigned.	Use the actual machine name or IP address. This allows clients from other machines access to EJB Server.
Port	The port number on the host to which the listener is assigned.	Make sure that the port is not in use by any other service.

Replacing an EJB Server

Use the Sybase Installer to add an EJB Server to a new Adaptive Server host. You can replace an existing EJB Server with a new EJB Server using the Adaptive Server plug-in.

Warning! Replacing an EJB Server removes the existing EJB Server from Sybase Central and from the `syservers` table in the master database. Connection caches and packages associated with the old EJB Server are lost.

Use caution when replacing an EJB Server.

Before replacing an EJB Server, make a list of the packages and connection caches in the existing server that you want to redeploy to the new server.

❖ To replace the EJB Server:

- 1 Highlight the Enterprise JavaBeans folder.
- 2 Double-click the Add an EJB Server icon in the right window.
The Add an EJB Server wizard displays.
- 3 Enter a name for the new server. Server names must be one word, and can be up to 30 characters long.
- 4 Enter a port number for the new server. The default is 9000.
- 5 Select Finish.
- 6 Install AseAuth from the Repository.
See “Installing a package in the Adaptive Server plug-in” on page 72 for how to install a package.
- 7 Set these property values on the All Properties tab of the EJB Server properties window for AseAuth:
 - `com.sybase.jaguar.server.authservice=AseAuth/DbAuth`
 - `com.sybase.jaguar.server.dbsecurityurl=jdbc:sybase:shm:null:0`
 - `com.sybase.jaguar.server.dbsecuritydriver=com.sybase.jdbc2.jdbc.SybDriver`
 - `com.sybase.jaguar.server.dbauthlogfile=dbauth.log` or the name of the EJB Server log file (optional)
 - `com.sybase.jaguar.service.roleservice=AseAuth/DbAuth`

See “All Properties” on page 149 for information on how to set these options.

- 8 Start the EJB Server. See “Starting EJB Server independently” on page 25.
- 9 Reimport packages from the Repository.
- 10 Reconfigure the connection caches.

EJB Server Naming Services

A *naming service* lets you associate a logical name with an object, such as a package and component. Naming helps EJB Server applications easily locate an object anywhere on a network, then implement the referenced object.

The naming service “binds” a name to an object. The combination of bound name and its referenced object is the *name context*. The referenced object in a name context can be a component within a package or even an existing name context, the same way a named directory can contain a file or other named directory.

The collection of name context information—each object and its bound name—comprises the *namespace*. When client applications reference an object, they look to the namespace to cross-reference or *resolve* the name with the referenced object.

Topic	Page
How does the EJB Server naming service work?	167
JNDI support	171
Configuring the EJB Server naming service	176
Using an LDAP server with EJB Server	177

How does the EJB Server naming service work?

The process of binding objects is performed by a name server. Each EJB Server can be its own name server, or you can configure an EJB Server to use another server as its name server. You can also use an external naming service, such as an LDAP server, in conjunction with the EJB Server naming service.

You set the naming service options for each EJB Server using the Naming Service tab on the Server Properties window.

EJB Server initial context

The EJB Server naming service relies on an “initial” or default name context for each EJB Server. You set the initial context when you set up the EJB Server Naming Service properties.

The server name context syntax follows a specific organization or schema. You can use this schema to represent the hierarchy of objects in the namespace, for example by geographic region, organizational unit, and so on.

If you use an EJB Server as the name EJB Server server uses this format:

<Level 1>/<Level 2>/<Level 3>/...

The number of levels depends on the hierarchy you want to represent. For example:

US/sybase/finance
US/sybase/marketing
US/sybase/sales

If you use an LDAP server as an external naming service, the initial context must follow the syntax and schema of the LDAP server. LDAP servers have predefined schema for common objects such as country, organization, and organizational unit. EJB Server uses the following format for an LDAP-compatible initial context:

ou=<organizational unit>, o=<organization>, c=<country>

Using the previous examples, the initial contexts would be:

ou=finance,o=sybase,c=US
ou=marketing,o=sybase,c=US
ou=sales,o=sybase,c=US

On start-up, the name server binds all object implementations on an EJB Server to the initial context of the server on which the object is installed. Once the server binds an object, the structure of the resulting name context is:

<initial context>/<package>/<component>

where

<initial context> is the initial context property for the server where the component is installed.

<package> is the name of the package being bound, as displayed in the Adaptive Server plug-in.

<component> is the name of the component being bound, as displayed in the Adaptive Server plug-in.

Note You can set the server properties to enable password protection for name binding on a EJB Server name server. See “Name binding password security” on page 177.

Name binding example

To illustrate how an EJB Server name server uses the initial context to create name contexts for objects on multiple servers, assume two EJB Servers:

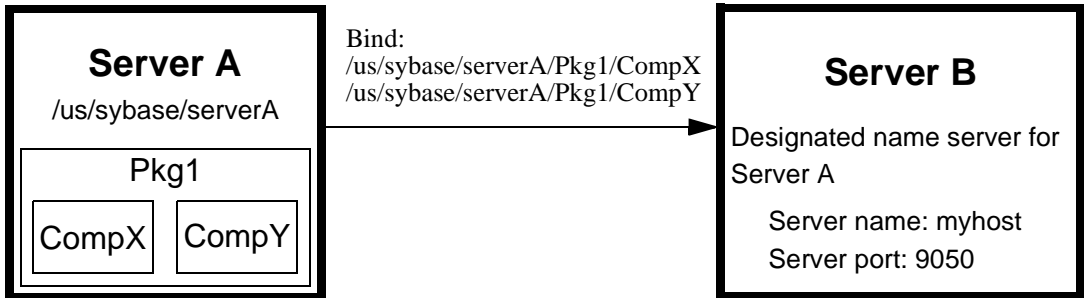
- Server A contains package Pkg1 and components CompX and CompY. You assign the server an initial context of */us/sybase/serverA*.
- Designate server B to be the name server for server A by specifying the URL for server B (*iiop://myhost:9050*) in its Naming Services properties.

When you start server A, it connects to server B, using the name server URL you entered in server A’s Naming Service properties. The name server gets the initial context for server A and binds each object installed on server A. The resulting name contexts are based on server A’s initial context, the package name, and the components in the package. For this example, the name server creates the following bindings:

```
/us/sybase/serverA/Pkg1/CompX  
/us/sybase/serverA/Pkg1/CompY
```

Figure 11-1 illustrates the name binding process.

Figure 11-1: Name binding process



An application referencing object `CompY` uses the URL of the name server, followed by the object's name context. For example:

`iiop://myhost:9050/us/sybase/serverA/Pkg1/CompY`

The name server finds the name context in the namespace, resolves the name context with the object it references, then implements the object.

If you had not assigned an initial context to Server A, the name server, server B, would create name contexts for objects `Pkg1/CompX` and `Pkg1/CompY` using the initial context of the name server. In this case, the client application can simply retrieve `CompY` using this URL:

`iiop://myhost:9050/Pkg1/CompY`

Transient vs. persistent storage

The EJB Server naming service inherently provides *transient* object name storage. The name server is instantiated when you start an EJB Server, and binds names to all the known object references. The name server provides the bound name and object references to the EJB Server's session manager object. Because this information is stored in memory, the name context information is retained only as long as the EJB Server is running.

You can add *persistent* object name storage capabilities to EJB Server by using an external directory naming service, such as an LDAP server. The external server retains object name information, and the EJB Server name server updates this information whenever it creates new bindings or unbinds existing ones.

To use an external naming service, specify the URL of the external server in the Naming Service properties of the designated EJB Server name server. You must also provide a manager DN (distinguished name) and password that has exclusive access to all objects in the LDAP server database for EJB Server to be able to update the stored name context information.

JNDI support

Java Naming and Directory Interface (JNDI) is a standard Java interface for accessing distributed objects and services by name. It provides a portable, unified interface for naming and directory services. The JNDI specification is independent of any specific directory or naming service such as LDAP, NDS, DCE/CDS, or NIS.

The EJB Server JNDI implementation includes the JNDI service provider interface (SPI), which enables you to use a variety of custom directory and naming services. EJB Server uses the SPI in conjunction with the CosNaming interface to provide component lookup capability. Given a bound name, the SPI locates the referenced package and component. Once it locates the component, the SPI works with the client stub interface to instantiate the component and return the requested object.

JNDI version level

In EJB Servers, the JNDI InitialContext object follows the JNDI 1.2 interface specification. When you start the EJB Server, the JNDI classes required for the server's JDK version are configured automatically.

JNDI J2EE features

EJB Server supports the JNDI features required by the Java 2 Enterprise Edition (J2EE) platform specification.

In J2EE, you can use the application component's naming environment to customize an application's business logic without accessing the source code. The application component's container implements the environment as a JNDI naming context and provides the JNDI interfaces to access the environment properties that you define in the deployment descriptor.

Environment properties

When you deploy a J2EE application, use the deployment descriptor to define all the environment properties that the application component needs to access. This sample code defines the environment property (env-entry) *maxExemptions* as an Integer and sets its value to 10:

```
<env-entry>
  <description>
    The maximum number of tax exemptions
  </description>
  <env-entry-name>maxExemptions</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>10</env-entry-value>
</env-entry>
```

The information between the opening and closing env-entry tags defines an environment entry element, which consists of:

- **description** This is optional.
- **env-entry-name** The environment property name, relative to the *java:comp/env* context.
- **env-entry-type** The environment property's Java datatype must be one of: Boolean, Byte, Double, Float, Integer, Long, Short, or String.
- **env-entry-value** The environment property value, which is optional.

Within the same container, all instances of an application component share the same environment properties. The component instances cannot modify the environment at runtime.

An application component instance uses the JNDI interfaces to locate the environment naming context and access the environment properties. To locate the naming context, an application creates a *javax.naming.InitialContext* object and gets the *InitialContext* 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 InitialContext();
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();
```

Default name service

When you call the empty constructor to create a new InitialContext, EJB Server sets the Context.INITIAL_CONTEXT_FACTORY system property and sets the EJB Server EJB name service as the default.

EJB references

An EJB reference identifies the home of an enterprise Bean. You can use the deployment descriptor to create a link between an EJB reference and an enterprise Bean, contained within an EJB JAR file. Deployment descriptor interfaces allow an application component to access an enterprise Bean's home interface using EJB references.

To locate an enterprise Bean's home interface, declare an EJB reference in the deployment descriptor and use JNDI to look up the interface. The referenced enterprise Bean must be in the *ejb* subcontext of the application component's environment.

Declaring an EJB reference

You can declare an EJB reference in the deployment descriptor using the `ejb-ref` element. The data between the opening and closing `ejb-ref` tags defines an `ejb-ref` element. This code sample defines an EJB reference to the *Employee* entity Bean:

```
<ejb-ref>
  <description>
    Reference to the Employee entity Bean
  </description>
  <ejb-ref-name>ejb/Employee</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.wooster.empl.EmployeeHome</home>
  <remote>com.wooster.empl.Employee</remote>
</ejb-ref>
```

An `ejb-ref` element contains:

- **description** This is optional.
- **ejb-ref-name** The name of the Bean used in the application component.

- **ejb-ref-type** The Bean type, Entity or Session.
- **home** The expected Java type of the home interface.
- **remote** The expected Java type of the remote interface.
- **ejb-link** This is optional.

This code sample illustrates how to use JNDI to look up the home interface of the *Employee* enterprise Bean:

```
// Get the default initial JNDI context
Context initContext = new InitialContext();

// Look up the home interface of the Employee enterprise
// Bean
Object result =
    initContext.lookup("java:comp/env/ejb/Employee");

// Convert the result to the correct type
EmployeeHome empHome = (EmployeeHome)
    javax.rmi.PortableRemoteObject.narrow(result,
        EmployeeHome.class);
```

Declaring an EJB link

You can define a link from an EJB reference to an enterprise Bean by declaring an `ejb-link` element in the deployment descriptor. The application component and the target enterprise Bean must be in the same J2EE application. This sample code creates a link to the *Employee* enterprise Bean, by adding an `ejb-link` element to the Bean's EJB reference definition:

```
<ejb-ref>
  <description>
    Reference to the Employee entity Bean
  </description>
  <ejb-ref-name>ejb/Employee</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.wooster.empl.EmployeeHome</home>
  <remote>com.wooster.empl.Employee</remote>
  <ejb-link>Employee</ejb-link>
</ejb-ref>
```

For information about using the Adaptive Server plug-in to add and configure EJB references in EJB components, see Chapter 6, "Working with EJB Packages and Components."

Resource factory references

A resource factory is an object that you use to create resources. You can assign a logical name to a resource factory in the deployment descriptor.

A `resource-ref` element defines a single resource factory reference. This code sample defines a reference to the resource factory that implements the `DataSource` interface:

```
<resource-ref>
  <description>
    Data source for the database in which the Employee
    enterprise Bean records transactions
  </description>
  <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

A `resource-ref` element contains:

- **description** This is optional.
- **res-ref-name** Resource reference name used in the application's code.
- **res-type** Resource Java datatype that the application expects.
- **res-auth** Resource sign-on authorization, Bean or Container.

This code sample obtains a reference to the resource factory that implements the `DataSource` interface, and uses that reference to get a database connection (resource):

```
// Obtain the initial JNDI context
Context initContext = new InitialContext();

// Look up the resource factory using JNDI
javax.sql.DataSource ds = (javax.sql.DataSource)
    initContext.lookup
        ("java:comp/env/jdbc/EmployeeAppDB");

// Get a database connection
java.sql.Connection connection = ds.getConnection();
```

For information about using the Adaptive Server plug-in to add and configure resource references in EJB components, see Chapter 6, "Working with EJB Packages and Components."

UserTransaction references

J2EE application components can use the Java Transaction API (JTA) UserTransaction interface to manage transactions. A component instance can look up an object that implements the interface using the JNDI name *java:comp/UserTransaction*.

In this code sample, an application component uses the interface to manage a transaction:

```
// Get the initial JNDI context
Context initContext = new InitialContext();

// Look up the UserTransaction object
UserTransaction tran = (UserTransaction)
    initContext.lookup("java:comp/UserTransaction");

// Start a transaction
tran.begin();

// data updates

// Commit the transaction
tran.commit();
```

Configuring the EJB Server naming service

Use the Naming Service tab on the Server Properties window to set the EJB Server's naming service options. You can use the Naming Service properties to configure an EJB Server to be a name server, or point to another EJB Server as its name server.

The Naming Service property sheet includes:

- The EJB Server's initial context.
- Whether or not the EJB Server is enabled as a name server.
- If the server is not enabled as a name server, the URL for the EJB Server acting as the name server.
- Heartbeat detection – periodically verifies that name servers are either accepting client connections or have failed.

- If you are using an LDAP server to provide persistent name storage, the URL of the LDAP name server, as well as the manager DN (distinguished name) for the LDAP server.

For complete information about setting the Naming Service properties for an EJB Server, see “Naming Service” on page 148.

Name binding password security

You can establish password protection on the EJB Server naming service to allow name binding only from designated EJB Servers. This prevents unauthorized applications from creating name bindings using an EJB Server name server.

To use the name binding password feature, you must set the:

```
com.sybase.jaguar.server.CosNaming.bindpassword
```

property for the name server and each server participating in the naming service. You set this property using the All Properties tab in the Server Properties window. The default value is “jaguar.”

All servers participating in the password-protected name service must have the same password as the name server. If the `bindpassword` property is empty, or does not exist in the property file for a name server, the name server accepts binds from any source.

Using an LDAP server with EJB Server

To add persistent object name storage capabilities to EJB Server, you can use an external directory naming service, such as an LDAP server. The EJB Server properties include an optional URL for specifying the port for the external name server.

When you use an external name server, EJB Server uses JNDI to communicate with the name server through the specified URL.

LDAP object schema and EJB Server objects

LDAP servers have predefined schema for common objects such as country, organization, and organizational unit. EJB Server uses the following format for an LDAP-compatible initial context:

ou=<organizational unit>, o=<organization>, c=<country>

Storing EJB Server object bindings on an LDAP server

When you use an LDAP server with an EJB Server name server, the *CosNaming* component binds all implemented objects on the servers that use the designated EJB Server name server, and stores the name context information on the LDAP server. If EJB Server detects previously-bound objects on the external name server, it updates the existing bindings with current name context information. When you shut down the EJB Server, it unbinds the stored information.

❖ To connect an EJB Server name server to an LDAP server:

- 1 On start-up, the EJB Server name server connects to the LDAP server using the URL specified in the EJB Server name server's Naming Service properties.
- 2 The EJB Server name server authenticates the connection to the LDAP server using the manager DN specified in the EJB Server name server's Naming Service properties.
- 3 The EJB Server name server attempts to retrieve any existing matching name contexts from the LDAP server. If successful, the EJB Server name server uses the existing name context information.
- 4 The EJB Server name server prepares the server object with the required attributes.
- 5 The EJB Server name server attempts to add the server object to the LDAP server. If the object already exists, the LDAP server updates the existing object with the current attributes.
- 6 The EJB Server server adds any new package/component name context information, or modifies the existing information if necessary.

Index

Symbols

- , (comma)
 - in SQL statements xv
- { } (curly braces)
 - in SQL statements xv
- () (parentheses)
 - in SQL statements xv
- [] (square brackets)
 - in SQL statements xv

A

- activation, component
 - definition of 53
- Adaptive Server plug-in 46, 131
 - application objects managed in 10
 - capabilities 22
 - generating EJB stubs with 98
 - overview of 9
 - setup 22
 - starting 22
- addresses, network
 - configuring 8
 - specifying in EJB clients 102
- afconfig.dat file 159
- applets
 - about 119
 - distributed applications 129
- applications, EJB Server
 - architecture of 42
 - creating 41
 - defining components for 46
 - deployment of 48
 - design of 44
 - introduction to 41
- architecture
 - EJB component 32
 - EJB Server 42

- of EJB Server applications 42
- AseAuth package 70, 165

B

- basic tasks 21–27
- bindpassword 177
- building
 - components 47
 - EJB Server applications 41
- business logic and EJB Server components 128

C

- caches, connection
 - support for 13
- character sets 12
 - conversions 12
- classes, Java
 - for EJB components 92
- classpath
 - environment variable 155
- CLASSPATH environment variable 129, 137, 155
- client session management 11
- client/server applications
 - Java 119, 137
- clients
 - deployment of 49
 - design considerations for 47
 - development process for 46
 - EJB 97
 - session management and 10
 - types of 8
- code set
 - See Also* character sets
- comma (.)
 - in SQL statements xv
- compiling

Index

- Java stubs 99
 - component
 - definition 18
 - executing methods on 19
 - instantiating 19
 - restarting after modifying 146, 155
 - component lifecycle
 - management of 11
 - component models
 - supported 18
 - component transaction server *see* EJB Server
 - components
 - building 47
 - client stubs and proxies for 8
 - configuring properties for 75
 - creation and destruction of 11, 52
 - defining 45
 - definition of 10
 - deploying 92
 - design of 45
 - development process for 46
 - EJB 31, 32, 69
 - installing to a package 75
 - introduction to 7
 - lifecycle management 11
 - lifecycle of 51, 52
 - overview 7
 - persistent 109
 - PowerJ 16
 - properties to control instance allocation 78
 - recycling of instances 54
 - refreshing 85
 - refreshing after modifying 10
 - stateful 54
 - stateful vs. stateless 54
 - stateless 52, 54
 - storage 116
 - supported types 7
 - transactional properties 58, 77
 - types of 7
 - concepts 19
 - concurrency
 - component property 79
 - configuring
 - listeners 163
 - connecting to the Adaptive Server 23
 - connection cache properties 155, 156
 - Advanced 156
 - cache-by-name 156
 - connection cache sanity 156
 - description 155
 - General 155
 - JDBC 157
 - name 155
 - number of connections 156
 - server name 155
 - service name 156
 - connection caches
 - creating 153
 - defining 46
 - installing 153
 - managing 153
 - modifying 154
 - removing 154
 - support for 13
 - connection caching 128
 - connection timeout
 - configuring for EJB clients 103
 - container 33
 - conventions
 - Java-SQL syntax xiii
 - Transact-SQL syntax xiv
 - create methods
 - IDL design pattern for 89
 - creating
 - listeners 164
 - curly braces ({})
 - in SQL statements xv
 - custom class list
 - configuring 85, 93
- ## D
- databases
 - data access and EJB Server 134
 - PowerJ JDBC access 126, 138
 - transaction management 132
 - transactions in Java 138
 - deactivation
 - definition of 53
 - default packages 70

- deleting
 - EJB Server packages 75, 94
 - listeners 164
 - deploying components
 - EJB JAR 48
 - PowerJ 48
 - deploying packages 10
 - description
 - component property 77
 - EJB Server package property 94
 - design, application 44
 - developing
 - EJB Server applications 41
 - developing clients 49
 - development process
 - PowerJ 119
 - disabling EJB Server 24
 - disconnecting from the Adaptive Server plug-in 23
 - distributed applications
 - applet clients 129
 - connection caching 134
 - Java and EJB Server 129
 - Java client, deploying 137
 - transaction management 132
 - Distributed Transaction Management (DTM) 159
 - dtm_tm_role system role 159
- E**
- early deactivation
 - definition of 52
 - EJB 8
 - See also* EJB clients, EJB components
 - client model 97
 - EJB Server support for 38
 - generating stubs for 98
 - home interfaces 89
 - JAR file 71, 95
 - overview of 31
 - remote interfaces 91
 - EJB architecture 32
 - EJB clients 32, 39
 - creating 97
 - EJB components
 - creating 69, 87
 - creating home interfaces for 89
 - defining remote interfaces for 91
 - deploying classes for 92
 - exporting 95
 - importing 71
 - introduction to 32
 - primary keys for 89
 - running 38
 - types of 33
 - using transactions in 35
 - EJB container 33
 - EJB Server 32
 - architecture 42
 - component lifecycle model 51
 - configuring 145
 - connection caching 128, 134
 - creating applications for 41
 - description 5
 - developing distributed applications with Java 129
 - disabling 24
 - early deactivation 132
 - EJB component support in 38
 - enabling 23
 - execution engine 6
 - instance pooling 136
 - Java components 128, 131
 - overview 1
 - prerequisite knowledge 17
 - result set management 134
 - roles 94
 - server runtime 6
 - ServerBean interface 136
 - services for components 131
 - shutting down 26
 - starting 25
 - transaction management 132
 - transaction processing model 55
 - verifying status 27
 - EJB Server properties 154
 - EJB Server roles 15
 - EJB Server runtime environment 20
 - EJB Server transactions
 - benefits of 56
 - explanation of 56
 - EJB transaction attributes 35
 - enabling EJB Server 23

Index

Enterprise JavaBeans

See EJB

Enterprise JavaBeans (EJB) components

building with PowerJ 141

entity Bean 34

EJB component type 33

entity components

definition of 109

environment variables

CLASSPATH 155

classpath 155

events, Java 127

configuring 79

definition of 52

instance timeout

component property 80

instances, component

properties to configure allocation of 78

instantiating

components 19

intercomponent calls

and EJB Server transactions 56

interfaces

EJB home 89

EJB remote 91

F

finder methods

IDL design pattern for 90

G

garbage collection, Java

configuring for EJB clients 103

general server properties

description 147

generating

EJB stubs 86, 98

H

HTTP

protocol and EJB Server 129

support for 8

I

IIOP

support for 8

initial context 168

installing

components 74

instance pooling

adding support for 54

J

J2EE roles 15, 91, 94

JAR file 71

EJB 1.1 71, 95

Java

components 120

packages for generated stubs 99

version for generated stubs 98

Java applications

about 120

developing 15

Java classes

for EJB components 92

Java clients

compiling 99

Java Connection Manager classes 135

JavaBean components, creating in PowerJ 121

jConnect database interface 138

JDBC

Java client/server applications 137

L

LDAP server 177

lifecycles

component states in 52

of components in general 51

listeners

configuring 8, 163

- creating 164
- default host name 163
- deleting 164
- modifying 164
- preconfigured 163
- properties 164
- localhost
 - default listener settings 163

M

- managing connection caches 153
- managing XA resources 159
- mapping roles 94
- menus
 - PowerJ 126
- method
 - restarting after modifying 146, 155
- middle-tier servers *see* EJB Server
- modifying
 - listeners 164
- multitier
 - application development overview 19

N

- name binding 169
- naming conventions
 - for Java stub files 99
- naming services
 - about 167
 - explanation of 12
 - initial context 168
 - LDAP server 177
 - name binding 169
 - password 177
 - persistent storage 170
 - support for 12
 - transient storage 170
- network
 - addresses 8
 - protocols 8
- number of user connections parameter 152

O

- object-oriented programming
 - PowerJ 15
- OTS/XA
 - transaction options 58
- overview
 - EJB Server features 1
 - multitier application development 19

P

- package
 - restarting after modifying 155
- package, EJB Server
 - definition 18
 - installing components in 75
 - modifying 93
 - properties of 94
 - refreshing after modifying 10
 - restarting after modifying 146
 - uses of 10
- package, Java
 - for generated stub classes 99
- parentheses ()
 - in SQL statements xv
- persistence
 - container managed 110
 - for entity components 109
 - for stateful components 114
 - of component state 109
- persistent storage 170
- pooling
 - component property 79
- port numbers
 - configuring for servers 8
 - specifying in EJB clients 102
- PowerJ 9
 - about 15
 - application logic 127
 - building EJB components 141
 - classes 123, 127
 - code window 124
 - component palettes 125
 - component targets 131
 - components 120

Index

- database access 126
- database forms 138
- data-bound controls 127, 140
- defining EJB components in 38
- deploying components 48
- development process 119
- events 127
- forms 125
- managed classes 127
- menus 126
- methods 127
- projects 122
- query object 126, 139
- targets 122
- transaction object 126, 138
- user interface 125
- preconfigured listeners
 - security profiles 163
- primary keys
 - specifying for EJB components 89
- properties
 - listeners 164
 - of components 75
 - of EJB Server packages 94
 - to configure component instance allocation 78
 - to configure threading behavior 78
 - to control transactional behavior 77
- protocols
 - HTTP 8
 - IIOP 8
 - supported 8
- proxies
 - purpose of 8
- proxy objects
 - and stubs 18
 - definition of 8

R

- refresh
 - disabling for components 85
- replacing an EJB Server 165
- restarting server after modifying components 146, 155
 - methods 146, 155

- packages 146, 155
- result sets
 - explanation of 14
- roles
 - EJB Server 15, 94
 - mapping of 94
- runtime
 - server engine 6
- runtime environment 20

S

- server
 - naming service 167
 - server applications *see* components / EJB Server
 - server debugging and trace properties
 - log file name 148
 - log file size 148
 - truncate log on start-up 148
 - server log
 - srv.log file 148
 - server properties
 - General 146
 - initial context 168
 - Log/Trace 147
 - Naming Service 148, 167
- servers
 - as managed in the Adaptive Server plug-inr 10
 - configuring network addresses for 8
 - overview of 6
 - protocols supported by 8
 - services provided by 6
 - use during development and testing 44
- service components
 - definition of 6
- services
 - provided by EJB Server 6
- session
 - client, management of 10
 - definition 19
- session Bean
 - EJB component type 33
 - stateful 33
 - stateless 34
- session management 10

- shared-memory connections 152
- sharing
 - component property 80
- shutting down EJB Server 26
- skeleton
 - definition 18
- sp_extengine stored procedure 26
- sp_serveroption stored procedure 25
- square brackets []
 - in SQL statements xv
- srv.log file
 - server log 148
- starting EJB Server 25
- state primitives, for transactions 61
- stateful components
 - definition of 54
- stateful session Bean 33
- stateless components
 - creating 54
 - deactivation and instance pooling of 52
 - definition of 54
- stateless session Bean 34
- states
 - in component lifecycle 52
- storage components
 - configuring 116
 - definition of 116
- stored procedures
 - sp_extengine 26
 - sp_serveroption 25
- stub object
 - definition of 8
- stubs
 - and proxy objects 18
 - compiling 99
 - explanation of 8
 - generating 86
- Sybase Central
 - Adaptive Server plug-in for 9
 - explanation of 9
- syntax conventions
 - Java-SQL xiii
 - Transact-SQL xiv
- System Administrator role 22

T

- terminology
 - component based applications 18
- thread safety
 - explanation of 13
- threading models
 - component properties to configure 78
- threads
 - management of 13
- timeouts
 - configuring properties for 80
 - for EJB clients 103
 - transaction 62
- trace flag properties 148
- transaction options
 - OTS/XA 58
- transaction timeout
 - component property 80
- transaction, EJB Server
 - definition of 55
- transactions
 - and intercomponent calls 56
 - benefits of using 56
 - component properties to configure 77
 - configuring timeout property for 62, 80
 - controlling outcome of 61
 - defining how components participate in 57
 - examples of 56, 63
 - how to commit and roll back 61
 - management by EJB Server 132
 - multi-component 61
 - overview of 55
 - semantics of 57
 - server processing of 55
 - specifying coordinators for 58
 - specifying how a component participates in 58
 - state primitives for 61
 - use in EJB components 35
- transient storage 170

U

- URL for JDBC calls 155
- user interface, designing in PowerJ 125
- user names

specifying in EJB clients 102

V

verifying status of EJB Server 27

W

Web applications

about 119

X

XA resources

afconfig.dat file 159

managing 159