Programmer’s Guide

EAServer

5.0

DOCUMENT ID: DC38036-01-0500-01

LAST REVISED: December 2003

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

Contents

About This Book

Subject

This book contains information about how to build distributed applications that run on Sybase EAServer.

Audience

The *EAServer Programmer's Guide* is written for application developers who are familiar with their chosen programming languages, specifically Java, C++, C, or an ActiveX scripting language.

Though you do not need to know all of these languages to create EAServer components or clients, the chapters that pertain to each language assume a basic familiarity with that language.

How to use this book

For an overview of EAServer design concepts, and the application development process, see these chapters:

- Chapter 1, "Creating Component-Based Applications" provides a level overview of a typical development process.
- Chapter 2, "Understanding Transactions and Component Lifecycles" describes how EAServer manages multi-component transactions and component lifecycles.

For general information on developing components for EAServer, see these chapters:

- Chapter 3, "Managing Applications and Packages in EAServer Manager" describes how to create applications and packages in EAServer Manager. These items are required vehicles for component deployment to EAServer.
- Chapter 4, "Defining Components" describes how to define packages and components in EAServer Manager and configure component properties.
- Chapter 5, "Defining Component Interfaces" describes how to create, view, and edit component interfaces in EAServer Manager.

For information on developing Enterprise Java Beans (EJB) components, see these chapters:

- Chapter 6, "Enterprise JavaBeans Overview" introduces the EJB component model.

- Chapter 7, “Creating Enterprise JavaBeans Components” describes how to create Enterprise JavaBeans components.
- Chapter 8, “Creating Enterprise JavaBeans Clients” describes how to implement a client that uses the EJB client interfaces to call EAServer component methods.
- Chapter 9, “EAServer EJB Interoperability” describes how to call non-EJB components from EJB clients or components, and how to call EJB components from non-EJB clients.
- Chapter 10, “Creating Application Clients” describes how to create and deploy EJB application clients.

For information on developing components using the Java/CORBA model, see these chapters:

- Chapter 11, “Creating CORBA Java Components” contains information about building CORBA-Java components.
- Chapter 12, “Creating CORBA Java Clients” describes how to implement a client that uses EAServer’s CORBA-compatible Object Request Broker (ORB) to call EAServer component methods. This chapter is also useful to Java developers that use another vendor’s Java ORB to interact with EAServer components.

For information on developing components using the CORBA/C++ model, see these chapters:

- Chapter 13, “CORBA C++ Overview” describes EAServer’s C++ support and explains how EAServer maps CORBA IDL datatypes to C++ datatypes.
- Chapter 14, “Creating CORBA C++ Components” contains information about building C++ components.
- Chapter 15, “Creating CORBA C++ Clients” describes how to develop C++ clients that connect to EAServer.

For information on developing EAServer components with PowerBuilder®, see these chapters:

- Chapter 16, “Creating PowerBuilder Components” contains information about building PowerBuilder components.
- Chapter 17, “Creating PowerBuilder Clients” describes how to develop PowerBuilder clients that connect to EAServer.

For information on developing ActiveX components, see these chapters:

- Chapter 18, “ActiveX Overview” describes EAServer’s ActiveX support, including how EAServer maps CORBA IDL datatypes to ActiveX datatypes.
- Chapter 19, “Creating ActiveX Components” contains information about building ActiveX components.
- Chapter 20, “Creating ActiveX Clients” describes how to develop clients that connect to EAServer using the EAServer ActiveX client proxy server.

For information on developing, configuring, and running Web applications, servlets, and Java Server Page, see these chapters:

- Chapter 21, “Creating Web Applications” describes how to define and configure Web applications.
- Chapter 22, “Creating Java Servlets” describes how to create and run Java servlets in EAServer.
- Chapter 24, “Creating JavaServer Pages” describes how to create and run Java ServerPages in EAServer.

For information on advanced component features, see these chapters:

- Chapter 25, “Sending Result Sets” describes how to send result sets from a method coded in C, C++, or Java.
- Chapter 26, “Using Connection Management” describes how to access connection caches from a method coded in C, C++, or Java.
- Chapter 27, “Creating Entity Components” describes how to create CORBA or EJB components that manage data using the EJB entity bean model.
- Chapter 28, “Configuring Persistence for Stateful Session Components” describes how to create stateful components that use a persistence mechanism to support passivation for single-server deployments and load balancing and failover for clustered server deployments.
- Chapter 29, “Configuring Persistence Mechanisms” contains reference material that is useful in configuring stateful session components and entity components.
- Chapter 30, “Configuring Custom Java Class Lists” describes how to configure custom Java class lists for components, Web applications, packages, J2EE applications, and servers.

- Chapter 31, “Using the Message Service” describes how to use EAServer’s asynchronous messaging service to implement event- or message-driven application logic in clients and components.
- Chapter 32, “Using the Thread Manager” describes how to create threads to perform asynchronous processing in EAServer components.
- Chapter 33, “Creating Service Components” describes how to create components that run as EAServer services.
- Chapter 34, “Creating and Using EAServer Pseudocomponents” describes EAServer’s C++ and Java pseudocomponent support.
- Chapter 35, “Creating JavaMail” describes how to use the JavaMail API to access an Internet mail server from Java components or servlets.
- Chapter 36, “Configuring Java XML Parser Support,” describes how to configure Java components, application clients, and Web applications to use standard APIs to parse XML.

If you have developed applications with Sybase Open Server™ or previous EAServer versions, you may be interested in these features explained in the Appendixes:

- Appendix A, “Executing Methods As Stored Procedures” contains reference pages for invoking EAServer methods from any front-end tool that can execute Sybase® stored procedures.
- Appendix B, “Migrating Open Server Applications to EAServer” explains how to adapt existing Open Server applications to run in EAServer.
- Appendix C, “Creating C Components” contains information about building C components.

Finally, for information on generating stubs and skeletons with the command-line IDL compiler, see Appendix D, “Using the Command Line IDL Compiler”.

Conventions

The formatting conventions used in this manual are:

Formatting example	To indicate
commands and methods	When used in descriptive text, this font indicates keywords such as: • Command names used in descriptive text • C++ and Java method or class names used in descriptive text • Java package names used in descriptive text • Property names in the raw format, as when using jagtool to configure applications rather than EAServer Manager

Formatting example	To indicate
variable, *package*, or *component*	Italic font indicates: • Program variables, such as *myCounter* • Parts of input text that must be substituted, for example: *Server*.log • File names • Names of components, EAServer packages, and other entities that are registered in the EAServer naming service
File | Save	Menu names and menu items are displayed in plain text. The vertical bar shows you how to navigate menu selections. For example, File | Save indicates "select Save from the File menu."
`package 1`	Monospace font indicates: • Information that you enter in EAServer Manager, a command line, or as program text • Example program fragments • Example output fragments

Accessibility features

EAServer 5.0 has been tested for compliance with U.S. government Section 508 Accessibility requirements. The online help for this product is also provided in HTML, JavaHelp, and Eclipse help formats, which you can navigate using a screen reader.

EAServer Manager supports working without a mouse. For more information, see "Keyboard navigation" in Chapter 2, "Sybase Central Overview," in the *EAServer System Administration Guide*.

The WST plug-in for Eclipse supports accessibility features for those that cannot use a mouse, are visually impaired or have other special needs. For information about these features refer to Eclipse help:

1 Start Eclipse

2 Select Help | Help Contents

3 Enter Accessibility in the Search dialog box

4 Select Accessible user interfaces or Accessibility features for Eclipse

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

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

Related documents

Core EAServer documentation The core EAServer documents are available in HTML format in your EAServer software installation, and in PDF and DynaText format on the *Technical Library* CD.

What's New in EAServer summarizes new functionality in this version.

The *EAServer Cookbook* contains tutorials and explains how to use the sample applications included with your EAServer software.

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

The *EAServer System Administration Guide* explains how to:

- Start the preconfigured Jaguar server and manage it with the EAServer Manager plug-in for Sybase Central™
- Create, configure, and start new application servers
- Define connection caches
- Create clusters of application servers to host load-balanced and highly available components and Web applications
- Monitor servers and application components
- Automate administration and monitoring tasks with jagtool

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

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

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

- Understand the EAServer security architecture
- Configure role-based security for components and Web applications

- Configure SSL certificate-based security for client connections using the Security Manager plug-in for Sybase Central
- Implement custom security services for authentication, authorization, and role membership evaluation
- Implement secure HTTP and IIOP client applications
- Deploy client applications that connect through Internet proxies and firewalls

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

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

The *EAServer Troubleshooting Guide* describes procedures for troubleshooting problems that EAServer users may encounter. This document is available only online; see the EAServer Troubleshooting Guide at http://www.sybase.com/detail?id=1024509.

Message Bridge for Java™ Message Bridge for Java simplifies the parsing and formatting of structured documents in Java applications. Message Bridge allows you to define structures in XML or other formats, and generates Java classes to parse and build documents and messages that follow the format. The *Message Bridge for Java User's Guide* describes how to use the Message Bridge tools and runtime APIs. This document is included in PDF and DynaText format on your *EAServer 5.0 Technical Library* CD.

Adaptive Server Anywhere documents EAServer includes a limited-license version of Adaptive Server Anywhere for use in running the samples and tutorials included with EAServer. Adaptive Server Anywhere documents are available on the Sybase Web site at http://sybooks.sybase.com/aw.html.

jConnect for JDBC documents EAServer includes the jConnect™ for JDBC™ driver to allow JDBC access to Sybase database servers and gateways. The *Programmer's Reference jConnect for JDBC* is available on the Sybase Web site at http://sybooks.sybase.com/jc.html.

Other sources of information

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

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the Technical Library CD. It is included with your software. To read or print documents on the Getting Started CD you need Adobe Acrobat Reader (downloadable at no charge from the Adobe Web site, using a link provided on the CD).
- The Technical Library CD contains product manuals and is included with your software. The DynaText reader (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.
- The 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 will find links to EBFs/Updates, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  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 Products from the navigation bar on the left.
3. Select a product name from the product list.
4. Select the Certification Report filter, specify a time frame, and click Go.
5. 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.

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.

PART 1

Overview

This part provides an overview of EAServer features, design concepts, and the application development process.

CHAPTER 1

Creating Component-Based Applications

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

Application architecture

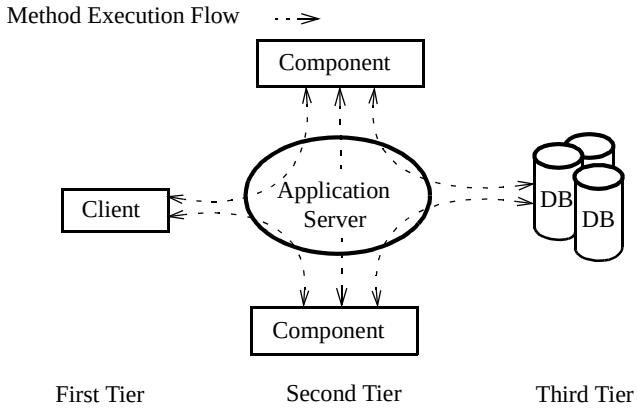
EAServer applications are composed of clients and one or more application servers to host business-logic components and Web components. The clients can run on different machines; the components execute on the server machine as part of the EAServer process. Some components, in turn, connect to databases on other machines.

Building EAServer applications is different from building standard client/server applications in that the parts of the application communicate with each other across network lines in a three-tiered architecture.

EAServer three-tiered architecture

In this figure, the client resides on the first tier, the application server and components reside on the second tier, and databases reside on the third tier. Executing methods on a component from the client or another component, retrieving data from databases, and other communications are managed by the application server. Because EAServer handles the details of transactions, threads, security, database connections, and network communication, you can concentrate on writing the business logic and user interface for the components and clients.

Figure 1-1: Component application architecture



As in traditional 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 server. 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 EAServer 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 EAServer application. If these components have been built outside EAServer, you can import them using EAServer Manager. 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 EAServer application

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

Follow these steps to design the application:

Plan for server infrastructure needs

For an enterprise application implemented by several developers, you may need to create several application 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. EAServer components (other than ActiveX components) are portable between Windows and UNIX. Your developers can develop and test components on inexpensive Windows workstations; later, you can deploy production versions to a high-end UNIX or Windows server.
- **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, EAServer Manager generates source for a stubbed implementation when you generate the component skeleton.)

  For a large application, you will need a dedicated QA team to test the application and a QA server to host configurations that are candidates for production release. The QA server and the client testing server can be the same. The QA server machine should have the same hardware architecture and operating system software as the production machine.
- **Production** You will need to install EAServer 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 EAServer 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. “Defining packages” on page 41 describes how to create packages in EAServer Manager.

Packages are the units of deployment for your application; you can use EAServer Manager 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 11.

Packages are also one level in the EAServer authorization hierarchy. You can edit the package's required Role Memberships to restrict which users can access components in the package. (Access can also be configured on the individual component level within the package.) Chapter 2, "Securing Component Access," in the *EAServer Security Administration and Programming Guide* describes options for configuring user authorization for package and component access.

Define components

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

Choose the Component Models Choose the component model based on your development team's expertise. See Chapter 3, "EAServer Components," in the *EAServer Feature Guide* if you are not familiar with the supported component models.

Design the Component Interface and Transactional Semantics Chapter 5, "Defining Component Interfaces" describes how to define interfaces. The component interface defines the methods that the component will implement. EAServer stores component interfaces as CORBA IDL; however, you can define and edit interfaces using your choice of EAServer Manager's method editor, Java, or IDL. For ActiveX components, you can also import method definitions from a DLL or type library file that you create using your ActiveX development tool.

While designing the interface, you must decide what transactional semantics the component will follow and how the component lifecycle will be managed. Chapter 2, "Understanding Transactions and Component Lifecycles" explains the design concepts for transaction and lifecycle control in EAServer components.

The following design decisions determine how EAServer manages your component's transactions:

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

If your component interacts with remote databases, you must specify a transactional attribute that determines how the component's database work is grouped within EAServer 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 EAServer 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 EAServer's transaction state primitives to indicate the status of the component's transactional work. "Using transaction state primitives" on page 25 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 EAServer Manager. 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, EAServer 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 or throwing the CORBA TRANSACTION_ROLLEDBACK exception. In EAServer Manager, the Automatic demarcation/deactivation property is set in the Component Properties window, beneath the Transactions tab. "Configuring component properties" on page 52 describes how to view and modify component properties in EAServer Manager.

For any component, transactional or not, you must decide how your component's instance lifecycle will be managed. "Component lifecycles" on page 13 describes the general instance lifecycle model and your options for instance lifecycle management.

Define the Component in EAServer Manager Use EAServer Manager to define the components. If you have already created Java or ActiveX components, you can import the component interfaces into EAServer Manager—you do not need to define method prototypes again in EAServer Manager.

"Defining components" on page 49 describes how to define components in EAServer Manager.

Define connection caches

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

You must define a connection cache for each remote database that your components interact with, and then implement your components to use cached connections. See the following sections for more information:

- Chapter 4, “Database Access,” in the *EAServer System Administration Guide* describes how to define connection caches in EAServer Manager.
- Chapter 26, “Using Connection Management” in this book describes how to access cached connections from your component implementation.

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 a Java-CORBA or EJB component, use a Java development tool to create the Java component. You can perform deployment tasks with EAServer Manager, jagtool, or jagant.

To create a C or C++ component, generate skeletons using EAServer Manager, code the method bodies in the method implementation templates, and compile and install the C DLL in your EAServer installation.

To create an ActiveX component, use an ActiveX-enabled IDE to create the ActiveX component DLL, import the ActiveX definitions for the component into EAServer, and install the ActiveX component.

To create a PowerBuilder component, use the EAServer Component wizard in PowerBuilder to define the interface, code the component in PowerScript, and deploy to EAServer.

To learn how to develop components, see these references:

Type of component	Chapter
EJB	Chapter 7, “Creating Enterprise JavaBeans Components”
Java-CORBA	Chapter 11, “Creating CORBA Java Components”
CORBA C++	Chapter 14, “Creating CORBA C++ Components”
PowerBuilder	The *Application Techniques* manual included in the PowerBuilder documentation.
ActiveX	Chapter 19, “Creating ActiveX Components”
C	Appendix C, “Creating C Components”

Design and implement the clients

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).

Choose Client Types Before creating client programs, decide which of the following EAServer client models best suits your needs, based on your preferred implementation languages and administrative requirements:

- **Web applications** You can invoke components from Java servlets and JavaServer Pages (JSPs) in a Web application. This approach allows the user interface to run anywhere a Web browser is installed. However, complex user interfaces with a high degree of interaction are difficult to implement.
- **Java** Java applets do not require customer installation and simplify the task of providing upgrades. The customer always downloads the most recent applet. Applets require that the customer's browser support JDK 1.2 or later.

  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.

  For Java development, you can use an IDE such as Borland JBuilder with the EAServer plugin. You can also use Jakarta Ant with jagtool tasks, as described in Chapter 12, "Using jagtool and jagant," in the *EAServer System Administration Guide*.
- **PowerBuilder** PowerBuilder is a Rapid Application Development (RAD) environment that supports drag-and-drop user interface generation. You can implement PowerBuilder clients that execute EAServer component methods using NVO proxies generated within PowerBuilder. As with C++ clients, the PowerBuilder runtime files must be distributed to each client workstation.
- **C++** C++ clients offer the proven performance of a native compiled executable. Some developers may prefer C++ user-interface generators such as Microsoft Visual C++. Finally, your company may have a large investment in existing C++ user-interface classes. C++ clients do require installation by the customer, however.

- **ActiveX** If you are more familiar developing applications with an ActiveX-enabled IDE rather than Java, you can create an ActiveX client. An ActiveX client requires the same runtime installation as a C++ client, plus an additional step to register EAServer's client proxy ActiveX interfaces.
- **Methods As Stored Procedures (MASP)** EAServer's MASP interface allows component methods to be executed as if they were database stored procedures. Any front-end tool that can execute Adaptive Server Enterprise stored procedures can execute EAServer methods using the MASP interface.

In some situations, you might want to implement different versions of a client for different users. For example, you may implement a Web client 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 and a richer user interface.

To learn how to create clients, see these references:

Type of client	Chapter
Java	Chapter 8, "Creating Enterprise JavaBeans Clients" Chapter 12, "Creating CORBA Java Clients"
C++	Chapter 15, "Creating CORBA C++ Clients"
PowerBuilder	The *Application Techniques* manual included in the PowerBuilder documentation.
ActiveX	Chapter 20, "Creating ActiveX Clients"
MASP	Appendix A, "Executing Methods As Stored Procedures"

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 Java 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 11.

2 "Deploying clients" on page 12.

Deploying components

For production deployment, you must copy component definitions and implementation files from your test server to the production server, or deploy directly from your IDE another project-based tool. There are several ways to deploy:

- Using PowerBuilder
- Using jagtool and jagant
- Using the Synchronize feature

Using PowerBuilder

PowerBuilder NVO components can be deployed directly to EAServer from the PowerBuilder IDE, using the Project Painter. See the PowerBuilder *Application Techniques* manual or online help for more information.

Using jagtool and jagant

jagtool is a command line interface that allows you to automate EAServer development and deployment tasks. jagant allows you to run the same tasks from Jakarta Ant build scripts. For more information, see Chapter 12, "Using jagtool and jagant," in the *EAServer System Administration Guide*.

Using the Synchronize feature

EAServer Manager's Synchronize feature allows you to replicate packages, components, connection cache definitions, and other configuration information from one server to another. Though this feature is intended primarily for synchronizing servers within a cluster, it is a convenient and quick way to replicate the server-side of your application from one server to another. Chapter 6, "Clusters and Synchronization," in the *EAServer System Administration Guide* describes this feature in detail.

Deploying clients

The client deployment process varies depending on what type of clients your application uses.

Java clients

To deploy a Java applet, you must create an HTML page that loads the applet. To deploy a Java application, you must supply all classes required by your application to end users for installation on their machines. See “Deploying and running Java clients” on page 230 for complete instructions.

ActiveX clients

To deploy ActiveX clients, you must install the EAServer ActiveX proxy on client workstations and register the proxy interfaces in the COM Automation Server Registry. See “Deploy the ActiveX client” on page 365 for more information.

C++ clients

To deploy C++ clients, you must copy the EAServer C++ client runtime libraries to the client machine and configure a few environment variables. See “Deploying C++ clients” on page 294 for more information.

PowerBuilder clients

PowerBuilder provides a variety of options for deploying applications. You can build an executable file and use the PowerBuilder Runtime Packager to package it with the PowerBuilder VM and other required files. You can also deploy a Web site that uses 4GL Web pages to access EAServer components. See the PowerBuilder *Application Techniques* and *Working with Web and JSP Targets* manuals for more information.

CHAPTER 2

Understanding Transactions and Component Lifecycles

This chapter explains the EAServer 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. EAServer's 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.

Component lifecycles

The EAServer 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, EAServer 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.

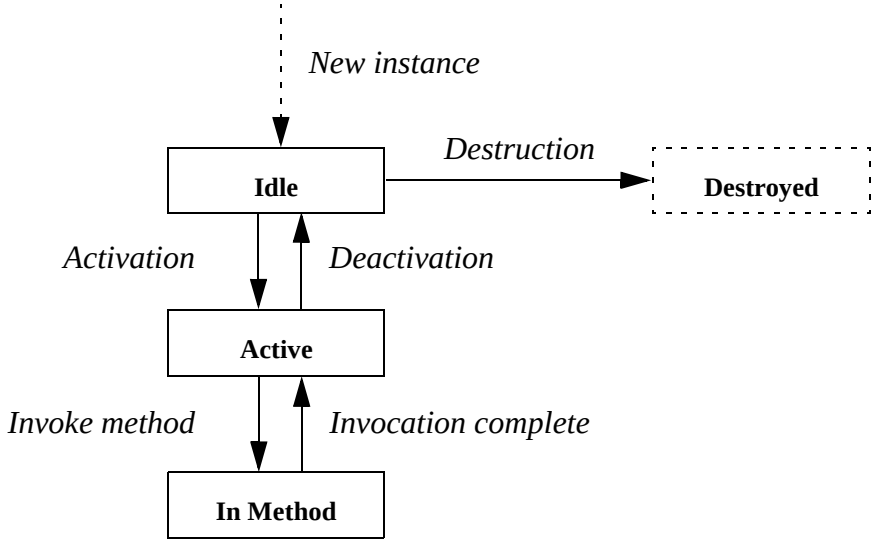
Early deactivation allows a component's methods to specify when deactivation occurs. Early deactivation prevents a client application from tying up the resources that are associated with a component instance and allows the instance to serve more clients in a given time frame. To achieve early deactivation, you can code or configure your component as described in "Supporting early deactivation in your component" on page 16.

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 EAServer components in any component model follow the state diagram illustrated in this figure:

Figure 2-1: States in the EAServer component lifecycle



The state transitions are as follows:

- **New instance** The EAServer 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 EAServer runtime calls the corresponding method in the component. The next state depends on which of the transaction state primitives the method calls before returning. (For Java components, the state transition also depends on whether the method returns with an uncaught exception.) See "Using transaction state primitives" on page 25 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 25, 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. "Component properties: Resources" on page 65 describes how to configure this property.

  If an exception occurs in a user transaction, you must call rollbackWork after catching the exception; otherwise, a transaction deadlock may occur in the database, which can cause client applications to fail.
- **Destruction** Destruction occurs if the component instance cannot be recycled. "Supporting instance pooling in your component" on page 16 describes how to ensure instance reuse. If the component cannot be reused, deactivation is followed by destruction of the instance.

The EAServer 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 early deactivation in your component

Early deactivation prevents a client application from tying up the resources (such as connections) that are associated with a component instance.

EJB stateless session beans and entity beans support early deactivation by design. If you have coded the component according to the EJB specification, no additional code or configuration is required to run in EAServer.

For components of other types, there are several ways to support early deactivation:

- Configure the component to implement a control interface as described "Configuring a control interface" on page 72. If using the CtsComponents::ObjectControl interface, you can enable the Stateless option on the Instances tab in the EAServer Manager Component Properties dialog box. If using another control interface, enable the Auto demarcation/deactivation option on the Transactions tab of Component Properties window (see "Component properties: Transactions" on page 58 for more information). With the appropriate option enabled, the component is automatically deactivated after every method invocation.
- Code your component to call one of the completeWork or rollbackWork transaction state primitives to cause explicit deactivation of the instance. This technique is useful when your design requires deactivation to occur after some, but not all, method invocations. If the component is transactional, the completeWork and rollbackWork primitives also affect the outcome of the transaction in which the component is participating. See "Using transaction state primitives" on page 25 for more information.

Supporting instance pooling in your component

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

For Java and ActiveX 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:

- *C++ or Java CORBA components* can implement a control interface as described "Configuring a control interface" on page 72.
- *EJB components* must implement EntityBean or SessionBean interface for lifecycle control. For more information, see Chapter 6, "Enterprise JavaBeans Overview."
- *ActiveX components* can implement the IObjectControl interface and the GetObjectContext method. See Chapter 2, "ActiveX C++ Interface Reference," of the *EAServer API Reference* for details.

For PowerBuilder components, you can enable the Pooling option on the PowerBuilder wizard that you use to create your component. You can then write event scripts that respond to changes in an instance's lifecycle. See the *Application Techniques* manual in the PowerBuilder documentation for more information.

For C and C++ components, you can enable instance pooling using EAServer Manager. Display the Instances tab in the Component Properties window, then select the Pooling option. This option also allows you to configure pooling for Java and ActiveX components that do not implement the ServerBean or IObjectControl interfaces, respectively.

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). Both the Java and ActiveX interfaces have methods that allow an instance to selectively refuse pooling: canReuse in Java, canBePooled in ActiveX. For PowerBuilder components, you can script the canBePooled event to selectively refuse pooling.

When the component Pooling option is set in EAServer Manager, the Java canReuse or ActiveX canBePooled method is not called, even if the component implements the ServerBean Java interface or IObjectControl ActiveX interface.

You can configure the component pooling properties to control how many instances are pooled, and to assign different components to a shared pool. For information on tuning these settings, see "Instance pooling" in Chapter 3, "Component Tuning," in the *EAServer Performance and Tuning Guide*.

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 “Component properties: Resources” on page 65 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 EAServer Manager, 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 a remote database** Use connection caching and database commands to accumulate data in a remote database. This is the preferred technique. If you deploy your component to a cluster, it may run on multiple servers and the database provides a central location available from all servers.
- **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.
- **Use the EAServer shared objects feature** EAServer provides a shared objects interface that allows components to store references to shared data. For more information, see the following sections:
  - Chapter 19, “Creating ActiveX Components”, describes the shared objects interface for ActiveX components.

- "Share data between C or C++ components" on page 688 describes the shared objects interface for C components.

EAServer's transaction processing model

An **EAServer transaction** is a transaction whose boundaries and outcome are determined by EAServer. Components can be marked as transactional in EAServer Manager. If a component is transactional, the EAServer transaction manager ensures that the component's third-tier database queries execute as part of a transaction. Multiple components can participate in an EAServer transaction; the EAServer 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 EAServer transactions work

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

The component lifecycle is tightly integrated with EAServer's 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 EAServer transactions

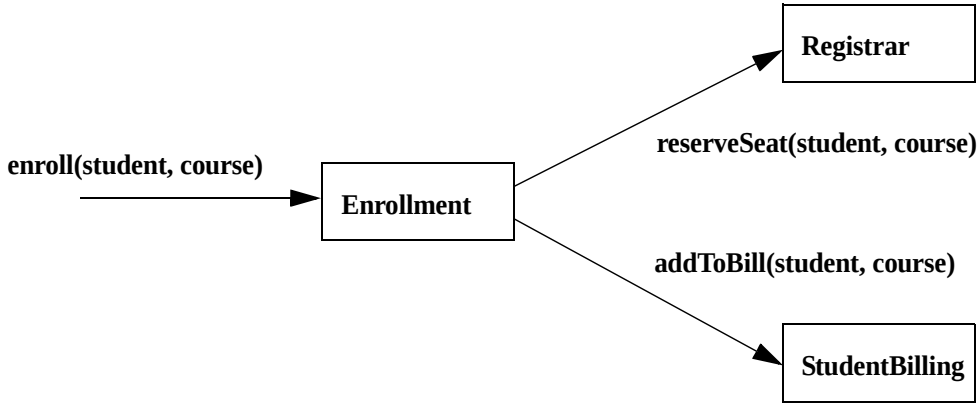
The benefits of using transactions to group database updates are clear. You can easily code methods in a single component to implement transactions that run against a single data source. However, those methods may in turn be executed by another component, which itself is defining a transaction. In this 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:

- Registar.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 2-2: An example EAServer 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 EAServer 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 EAServer transactions, you can be sure that the work performed by the components that participate in a transaction occurs as intended.

Defining transactional semantics

❖ **Defining how a component participates in transactions**

1 Choose a transaction coordinator. The transaction coordinator manages the flow of transactions that involve more than one connection. "Transaction coordinators" on page 21 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 22 describes the attribute settings and their meanings.

3 Code methods to call EAServer's 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 25 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 EAServer Manager. See "Transaction Timeout property" on page 27 for more information.

Transaction coordinators

All components installed in one server share the same transaction coordinator.

Choices for transaction coordinator include:

- **Java Transaction Service (JTS)** This option complies with the JTS and the Object Transaction Service (OTS) and X/Open Architecture (XA) standards. The JTS transaction coordinator integrates the functionality of the shared connection, OTS/XA, and JTS/JTA transaction modes, and uses two-phase commit to coordinate transactions among multiple databases.
- **Microsoft Distributed Transaction Coordinator (DTC)** DTC uses two-phase commit to coordinate transactions among multiple databases. DTC is available on Windows 2000 and Windows NT platforms as part of Microsoft SQL Server 6.5 or later versions.

  DTC transaction support in EAServer requires the following:

  - Microsoft DTC must be installed and running on the server host.
  - Any database servers used by your application must be DTC-compliant.
  - Your components must connect to the DTC-compliant databases using an ODBC connection cache or a JDBC connection cache that uses the JDBC-ODBC driver.

Note To verify that your EAServer edition supports two-phase commit, check the server console or the *$JAGUAR/bin/<server_name>.log* file.

The default coordinator is the JTS coordinator. To view or change the coordinator, use the Server Properties dialog box in EAServer Manager.

More transaction coordinators may be added in the future. The components you create now will not have to be changed to take advantage of the new transaction coordinators as they become available.

Transactional component attribute

Components in EAServer have a transaction attribute that indicates how a component participates in transactions. You can view and change a component's transaction attribute using EAServer Manager; the attribute is displayed on the Transactions tab in the Component Properties window. For PowerBuilder components, you can specify the attribute in the PowerBuilder wizards (doing so ensures that it is saved with the PowerBuilder project and not overwritten by redeployment). 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 an EAServer 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, EAServer 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) or the Current interface (for C++ components).
- **OTS Style** Uses OTS transactional behavior. The component can inherit a client or other component's transaction. If called without a transaction, the component can explicitly begin, commit, and roll back transactions by using the CORBA Current interface.

Current interface and OTS-style are incompatible Although you can set a Java-CORBA component's transaction attribute to OTS Style, you will not have access to the Current interface. Since an OTS-style component can inherit a transaction from a parent component, the component behaves as in the Supports Transactions attribute case.

Table 2-1 lists design scenarios and the transaction attributes that apply to each.

Table 2-1: Deciding on a transaction attribute

Design scenario	Applicable transaction attributes
Your component interacts with remote databases, and its methods may be called by another component as part of a larger transaction. Multiple updates are issued before calling completeWork, or an update depends on the results of queries that were issued since the last call to completeWork.	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 completeWork 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 remote 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 20, 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 a remote 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 EAServer 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 25.

Use a stub or proxy object for the called component For transactions to occur with the intended semantics, you must perform intercomponent calls using a stub or proxy object for the called component. Do not invoke another component's methods directly.

Using transaction state primitives

EAServer 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. Table 2-2 on page 26 lists the API mappings for each component type.

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 and PowerBuilder components. For information on the Java methods, see Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference*. For information on the PowerBuilder TransactionServer object, see the *Application Techiques* manual in the PowerBuilder documentation and the PowerBuilder online help.

Table 2-2: Java and PowerBuilder transaction primitives

Transaction primitive	Java InstanceContext method	PowerBuilder TransactionServer function
completeWork	completeWork	SetComplete
rollbackWork	rollbackWork	SetAbort
continueWork	continueWork	EnableCommit
disallowCommit	None. You can achieve the same effect by calling, and then raising an exception if deactivate is called before the next method invocation.	DisableCommit
isInTransaction	inTransaction	IsInTransaction
isRollbackOnly	isRollbackOnly	IsTransactionAborted

ActiveX, C, and C++ components call the methods and routines in the following table to invoke transaction primitives. See the *EAServer API Reference* for documentation of these methods and routines:

Table 2-3: ActiveX, C, and C++ transaction primitives

Transaction primitive	ActiveX IObjectContext method	C/C++ routine
completeWork	SetComplete	JagCompleteWork
rollbackWork	SetAbort	JagRollbackWork
continueWork	EnableCommit	JagContinueWork
disallowCommit	DisableCommit	JagDisallowCommit
isInTransaction	IsInTransaction	JagInTransaction
isRollbackOnly	Not supported	JagIsRollbackOnly

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 EAServer 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. “Supporting early deactivation in your component” on page 16 discusses how this feature can improve application performance.

Transaction Timeout property

The root instance’s Transaction Timeout property specifies the maximum duration of an EAServer transaction. The default timeout period is infinite. You can configure finite timeouts in EAServer Manager, as described in “Component properties: Resources” on page 65.

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.

When using the UserTransaction interface, the default timeout for transactions is 300 seconds (five minutes). To change this value, edit the *UserTxnManager.props* file, located in the EAServer *Repository/Component/CosTransactions* subdirectory, and set the value of the `com.sybase.jaguar.component.tx_timeout` property. A value of "0" means no timeout exists. You can also set the timeout value from a client (within a transaction it initiated) or in a bean-managed server component with the UserTransactions method setTransactionTimeout(secs).

Example

As discussed in "Benefits of using EAServer transactions" on page 20, EAServer 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 20. 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

EAServer 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 deenlistment 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, EAServer enlists the XAConnection in the transaction. If you start a transaction before you create an XAConnection, EAServer creates the connection and enlists it in the transaction.

Dynamic enlistment allows you to do this:

```
conn1 = ds1.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.

Earlier versions of EAServer required you to get and release connections within a single component method. In bean-managed transactions, you had to get and release a connection within the scope of a transaction.

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

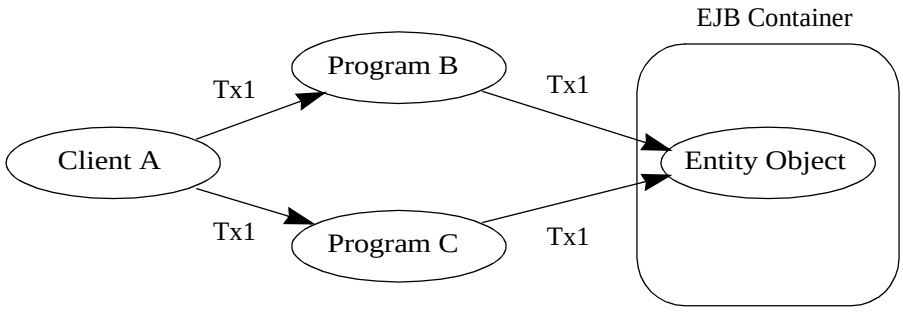
Note XA performance diminishes when connections span across methods.

Entity bean local diamonds

An entity object accessed from more than one path in the same transaction, as shown in Figure 2-3, is called a **diamond**. A **local diamond** exists when the access paths originate from, and the entity object resides on, the same server.

Typically, EAServer 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 a session 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 EAServer detects a diamond, it uploads data at method invocation and downloads data when the method completes.

Figure 2-3: Entity object diamond



EAServer Transaction Manager

The EAServer Transaction Manager supports the specifications for the Java Transaction API (JTA) 1.0 and the OTS/XA standards. The Transaction Manager supports the integrated functionality of these transaction coordinators: shared connections, OTS/XA, and JTS/JTA, and includes:

- Resource recovery and transaction logging
- Transaction interoperability
- Resource manager

The EAServer Transaction Manager enables EAServer to control the scope and duration of transactions across multiple resource managers. It also provides the ability to synchronize transactions and to communicate with other transaction managers using CORBA OTS. Connections and resources are dynamically enlisted into a transaction when they are requested.

Two-phase commit ensures that all changes to recoverable resources (for example, multiple database servers) occur automically, 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.

You can define components and component methods so that the transaction coordinator automatically handles transactions (implicit control). You can also write component and client code to manage transactions (explicit control).

EAServer implements the javax.transaction.TransactionManager interface, which allows it to control transaction boundaries, and to manage the interaction between Java and Encina transaction objects.

EAServer's implementation of the javax.transaction.Transaction interface enables it to manage a set of javax.transaction.xa.XAResource resources that participate in a transaction. To determine the boundaries and outcome for these transactions, EAServer uses the CosTransaction::Resource interface.

❖ **Configuring EAServer to use JTS/JTA transactions**

1 In EAServer Manager, highlight the server you want to configure.

2 Choose File | Server Properties.

3 In the Properties dialog box, select the Transactions tab.

4 Select JTS/JTA Transactions.

A component with the JTS transaction attribute enabled follows the standard component lifecycle as described in "Component lifecycles" on page 13.

Resource recovery and transaction logging

Resource recovery is a configurable option that provides object persistence and recovery operations. Basic persistence is achieved by writing transactions to a transaction log that contains all the information necessary to re-create the transaction. Persistence is supported for the CosTransactions::Resource and CosTransactions::Synchronization objects. Recovery is supported for JDBC connectors and native type resources that are registered with EAServer. When EAServer starts, the recovery manager is called, which reads the transaction log and starts transaction recovery.

Note Recovery operations can be performed only for transaction logs that were created for EAServer version 5.0.

A transaction log provides enhanced debugging and integrates with the standard EAServer logging functionality. Monitoring functionality is also provided, which allows you to use EAServer Manager to view statistics, such as the total number of transactions, currently active transactions, average duration of transactions, failed transactions, and remotely started transactions.

When EAServer starts, the TransactionLogManager verifies the transaction log's integrity, automatically does necessary repairs, then runs the transaction log defragmenter. This helps to allocate space for new transactions. The recovery manager passes transaction information to the TransactionLogManager, which is responsible for storing and deleting the transaction record from the transaction log.

Recovering XA resources registered by user components

In this version of EAServer, you cannot directly recover XA resources that are registered by user components. However, you can enable EAServer to accomplish this task by using the following technique:

1 Create a wrapper DataSource class; for example, WrapperDataSource.

2 WrapperDataSource.getXAConnection() returns an XAConnection class that corresponds to the XA connection with the resource.

3 Create an XA-type connection cache, and set its class name to the WrapperDataSource class that you created.

Once these steps are implemented, EAServer takes care of the recovery process. This is useful when using a third-party JMS service with XA resources.

Transaction interoperability

EAServer Transaction Manager provides transaction interoperability in accordance with the OTS specifications.

When EAServer runs in JTS mode, it can share the transaction coordinator across multiple servers. If a transactional component on one server invokes a component method on another server, both components can participate in the same transaction. Also, a client can invoke components on multiple servers that all participate in the same transaction. This feature is useful for load balancing.

Figure 2-4 illustrates a scenario in which a client calls a component method on Server A, which calls a component method on Server B. Server A and Server B use different databases. To ensure that all the database updates occur within the scope of a single transaction, EAServer passes the transaction context between servers.

Figure 2-4: Transaction interoperability

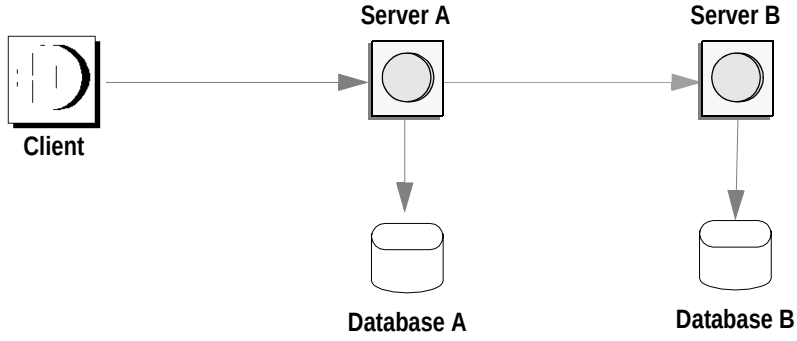
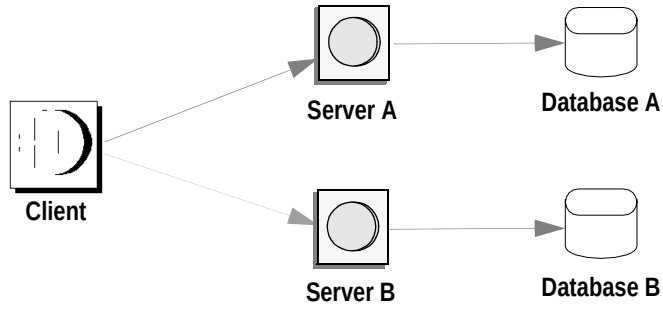


Figure 2-5 illustrates an example where a client calls components on multiple servers, which all participate in the same transaction. The client manages the transaction by calling component methods on each server and passing the transaction context.

Figure 2-5: Server to server



Resource manager

The EAServer Transaction Manager includes an integrated resource manager that supports JDBC 1.0, JDBC 2.0, connectors, and XA resources for both Java and C++. The resource manager allows you to dynamically register resources and synchronize coordinators in accordance with OTS specification for CosTransactions. The resource manager is based on the functionality of both the Java Connection Manager and the Jaguar Connection Manager, which allows you to easily integrate new and existing resources. In future EAServer versions, customers will be able to use the resource manager to create and configure resources that EAServer can use.

Enlisting XA resources with Transaction Manager

When EAServer is running in two-phase commit mode, which is the default for version 5.0, you can enlist XA resources with EAServer Transaction Manager.

❖ **Enlisting XA resources**

To enlist an XA resource into a current EAServer transaction:

1 Get the instance of Transaction Manager:

```
javax.transaction.TransactionManager tm =
com.sun.jts.jta.TransactionManager.getTransactionManagerImpl();
```

2 Get the instance of the transaction:

```
javax.transaction.Transaction trans = tm.getTransaction();
```

3 Register the XA resource with the transaction:

```
trans.enlistResource(xaresource);
```

EAServer manages this XA resource with respect to its transaction boundaries.

CHAPTER 3

Managing Applications and Packages in EAServer Manager

In EAServer Manager, packages allow you to group related components as a logical unit, and applications allow you to group related packages and Web Applications.

Defining applications

In EAServer Manager, Applications allow you to group related packages and Web applications into a single entity. In this way, you can deploy related business logic components, Web application components, and Web pages as a single unit between servers. For information on packages and Web applications, see:

- "Defining packages" on page 41
- Chapter 21, "Creating Web Applications"

You can import and export applications that have been archived in the standard J2EE Enterprise Archive (EAR) file format or the EAServer Jaguar JAR format. For details, see Chapter 9, "Importing and Exporting Application Components," in the *EAServer System Administration Guide*.

Creating and installing applications

You can create applications manually or by importing an EAR file as described in Chapter 9, "Importing and Exporting Application Components," in the *EAServer System Administration Guide*. An application must be installed into a server before the application's EJBs, servlets, or JSPs can run on that server.

❖ **Creating an application manually**

1 Highlight the top level Applications folder and choose File | New Application.

2 Enter a unique name for the application and click Ok.

❖ **Installing a package**

You can only install a package in one application. Once a package is installed in an application, it cannot be installed directly in a server. Install a package in your application as follows:

1 If necessary, expand the top level Applications folder.

2 If necessary, expand the icon for your application.

3 Highlight the Installed Packages folder beneath the application and choose File | Install Package.

4 Select the name of the package to install and click OK.

❖ **Installing a Web application**

You can only install a Web application in one application. Once a Web application is installed an application, it cannot be installed directly in a server. Install a Web application in your application as follows:

1 If necessary, expand the top level Applications folder.

2 If necessary, expand the icon for your application.

3 Highlight the Installed Web Applications folder beneath the application and choose File | Install Web Application.

4 Select the name of the Web application to install and click OK.

❖ **Installing an application in a server**

You must install your application in a server before the server's clients can call the application's components, servlets, and JSPs. Install your application as follows:

1 If necessary, expand the server's icon by double clicking on it.

2 Highlight the Installed Applications folder beneath the server icon and choose File | Install Application.

3 Choose the application to be installed from the list and click Ok.

Deleting and removing applications

To delete an application, highlight the application icon in EAServer Manager and choose File | Delete. You can choose between simple and full deletion. Simple deleletion removes only the application properties file and the properties files for components and Web applications installed in the application. Full deletion removes all files that have been generated by the deployment of the application, including component stubs and skeletons and IDL interface and datatype definitions. By default, EAServer performs a full deletion.

If you have installed an application in a server, you can remove it by highlighting the application icon in the server's Installed Applications folder and choosing File | Remove. This operation does not delete any files associated with the application.

Configuring application properties

To display an application's properties, highlight the application's icon, then choose File | Application Properties. You can configure the settings described below in the Application Properties dialog box that displays.

Application properties: General

You can enter optional text in the Description field to document your application.

Application properties: Role Mapping

These settings map role names used in the application's packages and Web applications to role names that exist in EAServer Manager.

❖ **Mapping a J2EE role to an EAServer role**

1 Select the Role Mapping tab from the Web application properties window.

2 Click Add. Double-click the J2EE role and enter a name. You can also enter a description for the role in the provided field.

3 Select an EAServer role from the drop-down list. This is the role from which the J2EE role inherits its permissions and members.

Refer to “Configuring EAServer roles” in the *EAServer Security Administration and Programming Guide* for more information about EAServer roles.

Application properties: Java Classes

This tab allows you to define a custom class list shared by all components and Web applications that are installed in the J2EE Application. See “Custom class lists for packages, applications, or servers” on page 560 for more information.

Application properties: Additional Files

Configures the `com.sybase.jaguar.application.files` property, which specifies additional files that are to be archived when the application is exported or replicated to another server with the synchronize feature. By default, the file set includes the files associated with Web applications, application clients, and packages installed in the application.

The rules for setting this property are the same as for the com.sybase.jaguar.component.files component property. See “Component properties: Additional Files” on page 69 for more information.

Application properties: JAXP Support

Configures the default JAXP, DOM, and XSLT parser implementations used by EJB components and Web applications in the application. See Chapter 36, “Configuring Java XML Parser Support,” for more information on these properties.

Application properties: Security

Security properties include:

Property	Description
Trusted Identities	Trusted identities must be configured for EJB 2.0 caller propagation from another server to the server where this application is deployed.Trusted identities can “vouch” for the client identity specified in the intercomponent call. For outgoing calls from this application to another server, configure the Security Identity property..
Security Identity	Specifies the identity used for outgoing component invocations when propagating client credentials to another server.
Run-as Identity	Specifies the identity used for intercomponent calls issued from EJBs or servlets. The identity specified here can be overridden in the package, Web application, or component properties.

For more information on these settings, see “Intercomponent authentication for EJBs and servlets” in the *EAServer Security Administration and Programming Guide*.

Application properties: Advanced

For advanced users only. The Advanced tab allows you to hand edit property settings in the EAServer configuration repository.

For information on repository properties, see Appendix B, “Repository Properties Reference,” in the *EAServer System Administration Guide*

Defining packages

In EAServer Manager, a package contains a group of related components. Typically, components in a package work together to provide a coherent service or function.

Refresh when you modify a package, component, or method
When you modify an existing package, component, or method, you must refresh the server, package, or component for the changes to take effect. To refresh, highlight the icon for the server, package, or component and select File | Refresh. If you modify a component’s supported interfaces, you must regenerate stubs and skeletons for the component and clients that access the component.

You must install your components in packages before applications can access the components. Packages serve the following purposes:

- **They are a unit of deployment** Using EAServer Manager, 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 components** Packages form one level in the EAServer authorization hierarchy. A package is not available on a server unless it is installed in the server's Installed Packages folder. To further restrict access for non-EJB components, you can edit the package's required Role Memberships to restrict which users can access components in the package. You can also control access on the individual component level. Chapter 2, "Securing Component Access," in the *EAServer Security Administration and Programming Guide* describes options for configuring user authorization for package and component access. EJB components use a different security mechanism described in "Configuring role references and method permissions" on page 133.
- **In a cluster, they allow you to partition the load** By installing different subsets of packages to the servers in a cluster, you can control which components execute on which servers within the cluster. See Chapter 7, "Load Balancing, Failover, and Component Availability," in the *EAServer System Administration Guide* for more information.

Use EAServer Manager to create, modify, and delete packages, as described in the sections below:

- "Creating a new package" on page 42
- "Installing packages to a server" on page 44
- "Modifying packages" on page 45
- "Configuring package properties" on page 45

You can also export and import package archives in the standard EJB-JAR format or in the Jaguar JAR format. For details, see Chapter 9, "Importing and Exporting Application Components," in the *EAServer System Administration Guide*.

Creating a new package

❖ **Creating a new package**

1 Start EAServer Manager if it is not running, and connect to your server.

2 Expand the EAServer Manager icon.

3 Highlight the Packages icon.

4 Select File | New Package.

5 Enter the name of the new package. The name must not match any existing package defined in the EAServer repository. To avoid name collisions, you can use the Java reverse domain naming style; for example, "com.foo.finance."

6 Supply the package information. The properties are described in "Configuring package properties" on page 45.

The new package appears on the right side of the screen when you highlight the package icon.

Package names must begin with a letter, are not case sensitive, and must be unique

Package names must be unique among other packages in the same EAServer installation, and begin with a letter.

Names are not case sensitive. Your packages must have unique names that differ in ways other than letter case. For example, you cannot define two packages named *MyPack* and *mypack* in the same EAServer installation. You cannot have two packages with the same name, even if one is installed in an application and the other is not.

❖ **Copying package definitions**

Use the Copy item in the package Edit menu to create a copy of a package. EAServer Manager creates a copy of the package definition and the definitions of the components it contains. You can modify the new package and component properties without affecting the original. However, the copied and original definitions refer to the same IDL interfaces and implementation files.

You cannot copy packages that are installed in an application. Packages that appear in the top-level Packages folder can be copied as follows:

1 In the top-level Packages folder, highlight the icon for the package to be copied.

2 Choose File | Copy.

3 Enter a unique name for the new package and click Ok.

4 EAServer Manager creates a copy with the specified name.

Note The Paste command in the package Edit menu pastes copied component definitions. See "Copying and pasting components" on page 52.

Installing packages to a server

Except for packages used internally by EAServer, packages to be run on a server must be installed in that server, using one of two methods:

- Add the package to the server's Installed Packages folder.
- Add the package to an application's Installed Packages folder, then install the application to the server. See Chapter 3, "Managing Applications and Packages in EAServer Manager" describes this method.

Packages that you create must be installed in a server before that server's clients can access components in the package.

You can only install a package in one application. Once a package is installed in an application, it cannot be installed directly in a server.

Default packages EAServer includes a set of default packages that include components used internally by EAServer. These packages are available whether or not they are installed to a server's Packages folder. These include the packages: CosConcurrencyControl, CosNaming, CosTransactions, CtsComponents, CtsSecurity, DataWindow, EncinaInternal, EncinaOTS, JTS, Jaguar, JaguarOTS, JaguarProxy, JaguarServlet, OtsAdmin, PBDebugger, Proxy, TranLog. The list of default packages is subject to change without notice.

❖ **Installing packages**

1 Double-click the Servers folder to expand it.

2 Double-click the server (listed on the left side of the screen) to which you want to install a package.

3 Highlight the Installed Packages icon. A list of installed packages appears on the right.

4 Select File |Install Package. Then select one of the following options from the Package Wizard:

- **Install an existing package** A list of uninstalled packages appears in the dialog box. Highlight the package you want to install, and click Ok.
- **Create and install a new package** Enter the name of the new package you want to install. Supply the package information, and click Ok. The properties are described in "Configuring package properties" on page 45.

Modifying packages

❖ **Modifying an existing package**

1 Highlight the package you want to modify. You can highlight the package icon displayed in a server's package folder or in the main Packages folder (both icons represent the same package as long as the package names are identical).

2 From the File menu, select one of the following options:

- **Package Properties** Displays the Package Properties window described in "Configuring package properties" on page 45. Make any modifications required, and click Ok.
- **Remove Package** If you have selected a package that is installed in a server or application, this option removes the package from the server.
- **Delete Package** Deletes the package from the system. You can choose between simple and full deletion. Simple deleletion removes only the package properties file. Full deletion removes all files that have been generated by the deployment of the package, including component stubs and skeletons and IDL interface and datatype definitions. By default, EAServer performs a full deletion.

Default packages cannot be modified or deleted
EAServer's default packages cannot be modified or deleted, and you cannot modify or delete components installed in default packages. These components are run internally by EAServer. See "Default packages" on page 44 for more information.

Configuring package properties

The Package Properties window has two tabs, General and Advanced.

Package properties: General

The following table describes the properties on the General tab.

Table 3-1: 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.

Package properties: Java Classes

This tab allows you to define a custom class list shared by all components that are installed in the package. See "Custom class lists for packages, applications, or servers" on page 560 for more information.

Package properties: Additional Files

Configures the `com.sybase.jaguar.package.files` property, which specifies additional files that are to be archived when the package is exported or replicated to another server with the synchronize feature. By default, the file set includes the files associated with components in the package.

The rules for setting this property are the same as for the com.sybase.jaguar.component.files component property. See "Component properties: Additional Files" on page 69 for more information.

Package properties: Role Mapping

These settings map role names used in EJB components to role names that exist in EAServer Manager.

❖ **Mapping a J2EE role to an EAServer role**

1 If necessary, define a new EAServer role as described in "Configuring EAServer roles" in the *EAServer Security Administration and Programming Guide* for instructions.

2 Select the Role Mapping tab from the Web application properties window.

3 Click Add. Double-click the J2EE role and enter a name. You can also enter a description for the role in the provided field.

4 Select an EAServer role from the drop-down list. This is the role from which the J2EE role inherits its permissions and members.

Package properties: JAXP Support

Configures the default JAXP, DOM, and XSLT parser implementations used by EJB components in the package. See Chapter 36, “Configuring Java XML Parser Support,”for more information on these properties.

Package Properties: Advanced

The Advanced tab allows you to edit package property settings as they are stored in the EAServer configuration repository. You can only delete properties that you have added—you cannot delete default properties, such as the com.sybase.jaguar.package.name property.

For information on repository properties, see Appendix B, “Repository Properties Reference,” in the *EAServer System Administration Guide*.

❖ **Setting 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.

When to use the Advanced tab

Though you can use the Advanced tab to set any property prefixed with com.sybase.jaguar.package, Sybase recommends that you use this tab to set properties only as specified by the EAServer documentation or by Sybase Technical Support. Most properties can be configured graphically elsewhere in the EAServer Manager user interface.

CHAPTER 4

Defining Components

EAServer supports Java, PowerBuilder, C++, ActiveX, and C components. If you are developing components with PowerBuilder, you can create and deploy components directly from your development environment. See the PowerBuilder *Application Techniques* manual for more information.

When developing with other tools, you must define packages and components in EAServer Manager.

Defining components

You can define components in EAServer Manager, Sybase PowerBuilder, or an EJB-compatible Java development tool such as Borland JBuilder.

PowerBuilder components are installed into EAServer using the PowerBuilder user interface. After a PowerBuilder component is installed in EAServer, you can view the settings in EAServer Manager. Sybase recommends that you edit all PowerBuilder component settings from PowerBuilder so that the EAServer component definition remains in sync with the PowerBuilder object definition and project settings. See the *Application Techniques* manual in the PowerBuilder documentation for more information.

If you are developing EJB components using Jakarta Ant or a Java IDE, you can deploy them to EAServer in an EJB-JAR file. The EJB-JAR format is specified by the EJB specification and allows portability between different J2EE-based application servers. EJB-JAR files can be deployed using EAServer Manager, jagtool or jagant, or the EAServer plug-in for Borland JBuilder. Chapter 9, "Importing and Exporting Application Components," in the *EAServer System Administration Guide* describes how to import EJB-JAR files. If you follow this process, most all component properties are configured properly to match the deployment descriptor provided with the EJB-JAR file. The exceptions are properties that depend on the server environment, such as resource, environment, or EJB reference properties.

❖ **Defining components in EAServer Manager**

1 Decide how you will define the component interface. Your options are:

Option	Description
Importing a compiled Java file	EAServer Manager reads method definitions from a compiled Java class or interface. "Importing interfaces from compiled Java files" on page 81 describes this feature in detail.
Importing an ActiveX file	EAServer Manager reads the interface definition from your ActiveX DLL or a separate ActiveX type library file. "Importing ActiveX components" on page 328 describes this feature in detail.
Defining the interface manually	You can define methods manually using EAServer Manager graphical controls or by creating the interface in an IDL file.

If you import from an ActiveX or Java file, skip to step 3. (The import process installs the component and sets the properties on the General tab in the Component Properties window.)

2 Install the component in an EAServer package. See "Installing components" on page 51 for more information.

3 Configure the settings in the Component Properties window, as described in "Configuring component properties" on page 52.

Component name limitations
Components in a package must have unique names that differ in ways other than letter case. For example, you cannot install two components named *MyComp* and *mycomp* in the same package. The maximum number of characters you can use to define the component's name is 227. You can use any characters in the name except these:

```
\ / : ; , * ? " < > |
```

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.

❖ **Creating a new component and installing it to a package**

1 Double-click the Packages folder to expand it, or if the package is installed in a server, expand the server's Installed Packages folder.

2 Highlight the package to which the component will be added.

3 Select File | Install Component from the menu.

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 Finish.

The Component Properties window displays.

6 Configure the settings as described in "Configuring component properties" on page 52.

The new component appears in the package's list of installed components, and the Component Properties window displays.

❖ **Renaming a component**

- Highlight the component and choose Rename, then enter the new name.

❖ **Copying and pasting components**

Use copy/paste to copy a component's definition to another package.

1 Highlight the icon for the component to be copied.

2 Choose File | Copy Component.

3 Highlight the icon for the package to which you want to copy the component.

4 Choose File | Paste Component.

EAServer Manager installs a copy of the component's definition into the specified package. You can modify the new component's properties without affecting the original. However, the copied and original definitions refer to the same IDL interfaces and implementation files.

❖ **Deleting a component**

1 Expand the EAServer package that contains the component.

2 Highlight the component you want to delete.

3 Select File | Remove Component from the menu.

When you delete a component, EAServer Manager does not delete the IDL interfaces and types that were used by the component. If you are sure that the component's interface and types are not used by any other component, you can delete unused types as described in "Editing IDL types, exceptions, and interfaces" on page 88. Alternatively, you can delete the package in which the component is installed and specify full deletion as described in "Modifying packages" on page 45.

Configuring component properties

The Component Properties window configures the settings that EAServer uses to load the component and invoke its methods. Component properties are organized on the following tabs:

Tab	Description
Component properties: General	Defines basic information about the component, including the component type and implementation details such as the Java class name or the C++ library name.

Tab	Description
Component properties: Transactions	Defines the components transactional properties, such as how the component participates in transactions and whether the component explicitly commits its work.
Component properties: Instances	Defines how instances of the component are managed, including instance creation, thread binding, and client/component bindings.
Component properties: Environment	For EJB 2.0 or 1.1 components, allows you to specify read-only site specific data for use by the component.
Component properties: EJB Local Refs	For EJB 2.0 components, allows you to configure aliases for EJB components that this component calls using EJB local references.
Component properties: EJB Refs	For EJB 2.0 or 1.1 components, allows you to configure aliases for components called by this component.
Component properties: Resource Refs	For EJB 2.0 or 1.1 components, allows you to configure aliases for resources used by the component such as JavaMail sessions or JDBC connections.
Component properties: Resource Environment Refs	For EJB 2.0 components, configures logical names for objects administered by EAServer.
Component properties: Role Refs	For EJB 2.0 or 1.1 components, allows you to map role names used in method permissions to role names defined in the EAServer repository.
Component properties: Resources	Configures properties that govern the component's use of server and database resources.
Component properties: Persistence	Specifies the primary key type for EJB entity Beans, and configures properties used to save state information for stateful components that can fail over between servers in a cluster.
Component properties: Run-As Identity	For EJB 2.0 components, specifies the authentication credentials that are used when methods call other components..
Component properties: Run-As Mode	For EJB 1.0 components, specifies the user name and password that are used for intercomponent calls to components installed in the same server or cluster.
Component properties: MDB Type	Applies to Message-Driven Bean (MDB) components only. See "Message-driven beans" on page 577 for more information.
Component properties: Mirror Cache	Configures properties required to support in-memory failover for stateful components running in a cluster. See "Mirror Cache tab component properties" on page 551 for more information.
Component properties: Java Classes	Configures the custom class list for Java and EJB components.
Component properties: Additional Files	Configures the file set to be included when the component is exported in Jaguar JAR format or replicated to another installation using the synchronize feature.

Tab	Description
Component properties: JAXP Support	For EJB 2.0 components, configures the XML parser implementations used by the component.
Component properties: Advanced	Allows you to manually edit component property settings in the EAServer configuration repository. For advanced users.

Component properties: General

The General tab defines basic information about the component, including the supported IDL interfaces, the component type, and implementation details. If you imported a Java or ActiveX component, these properties have already been configured correctly by the import process. The following table describes the window controls.

Table 4-1: General tab component properties

Property	Description	Notes
Description	Specifies description of the component. The description can be up to 255 characters.	Enter a comment that describes the purpose of the component.
Codeset (*PowerBuilder, C, and C++ components only*)	Specifies the name of the coded character set used by a C or C++ component. By default, the component uses the server's coded character set (specified on the General tab in the Server Properties window). This field does not display for Java and ActiveX components. These components always use 16-bit Unicode. For the list of supported values, list the subdirectories of the *charsets* directory. Each subdirectory matches the name of a supported character set.	Input values for string parameters (and string fields within complex datatype values) are converted to this code set before each method invocation. Upon return, output values are converted from the component's code set to the client's code set. **Note** If your C or C++ component uses Client-Library connection caches, you cannot specify a code set that is different than the server code set. Character data read over a cached Client-Library connection is always in the server's code set.

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 - Message Driven Bean** An EJB component that responds solely to JMS messages and lacks a client interface. **Java - CORBA** A Java component that uses the Java/IDL datatypes as defined by the CORBA specification for IDL-to-Java type mappings. **Java - JDBC** A Java component that uses the JDBC column types for parameter and return types.These type mappings are deprecated. See "Choose implementation datatypes" on page 185. **PowerBuilder NVO** A PowerBuilder nonvisual object adapted to run as an EAServer component. **COM/ActiveX** An ActiveX component adapted to run as an EAServer component. **C++** A C++ class adapted to run as an EAServer component. **C** A collection of C routines adapted to run as an EAServer component.	EJB components must be implemented in accord with version 1.0, 1.1, or 2.0 of the Enterprise JavaBeans specification. Version 2.0 is recommended for new development. ActiveX components are supported only in the Windows version of EAServer. All other component types can run on any platform that is supported by EAServer. PowerBuilder components should be configured and deployed using the PowerBuilder IDE. Otherwise, EAServer Manager settings may be overwritten when you redeploy from your PowerBuilder project. See the PowerBuilder *Application Techniques* manual for more information.
EJB Version *only for EJB components*	Choose to match the EJB specification version number. EAServer's interaction with the component is governed by the specification version.	Version 2.0 is recommended for new development.
CMP Version *only for EJB 2.0 entity beans*	For EJB 2.0 entity beans that use container-managed persistence (CMP), sets the CMP version number. If you do not specify a value, the default is 1.1.	In EJB 2.0 entity beans, you can use the CMP models from the EJB 2.0 or EJB 1.1 specifications. Version 2.0 is recommended for new development. Version 1.1 allows you to use existing implementation code that requires the EJB 1.1 CMP model.
Bean Class *only for EJB components*	The name of the class that implements the bean, in Java dot notation.	

Property	Description	Notes
MDB Class *only for EJB MDB components*	The name of the class that implements the message-driven bean, in Java dot notation.	

JNDI Name
only for EJB components
	The unqualified name used by client applications to look up the bean's home interface in the naming service. For example:
	`finance/account`
	The fully qualified name is obtained by appending the JNDI name to the server's initial naming context, for example:
	`/finserver/finance/account`
		If you do not specify a name, the default is *package/component*, where *package* is the EAServer package name, and *component* is the component name.

Home Interface Class *only for EJB components*	The Java class that defines the bean's home interface, in Java dot notation.	This field is read only. The class name is determined from the IDL home interface. You can add, view, or edit the IDL home interface using the component's interfaces folder.
Remote Interface Class *only for EJB components*	The Java class that defines the bean's remote interface, in Java dot notation.	This field is read only. The class name is determined from the IDL remote interface. You can add, view, or edit the IDL remote interface using the component's interfaces folder.
Local Home Interface Class *only for EJB components*	The Java class that defines the bean's local home interface, in Java dot notation. Blank if the bean does not have local interfaces.	This field is read only. The class name is determined from the IDL home interface. You can add, view, or edit the IDL home interface using the component's interfaces folder.
Local Interface Class *only for EJB components*	The Java class that defines the bean's local interface, in Java dot notation. Blank if the bean does not have local interfaces.	This field is read only. The class name is determined from the IDL remote interface. You can add, view, or edit the IDL remote interface using the component's interfaces folder.
Primary Key Class *only for EJB entity Beans*	The Java class that defines the entity bean's primary key type, in Java dot notation.	This field is read only. The class name is determined from the IDL struct type that defines the bean's primary key. You can add a primary key type to a module listed in EAServer Manager IDL folder. Typically the bean's primary key structure, home interface, and remote interface are defined in the same IDL module. To set the bean's IDL primary key type, enter the type name in the Primary Key field on the Persistence tab.

Property	Description	Notes
Fully Qualified Java Class *only for non-EJB Java components*	The fully qualified name of the Java class file that implements the component's methods, specified in Java dot notation, as in: `com.yourcorp.YourCompImpl`	
DLL Name *only for C and C++ components*	The name of the Windows DLL or UNIX shared library that contains the component methods. You can omit platform standard file extensions if desired (such as *.dll* on Windows or *.so* on Solaris).	Before running the component, the library files must be copied to *cpplib* subdirectory in the EAServer installation directory.
C++ Class (*only for C++ components*)	The name of the C++ class that implements the component.	
C++ Executable (*only for C++ components*)	The name of an external process in which the C++ component runs.	Run the component externally if you do not completely trust the implemenation not to crash. See "Running C++ components externally" on page 269 for details.
Use Platform Independent Library Naming (*only for C++ components*)	If selected, the platform name is included in the component library and executable name, to allow deployment to mixed architecture clusters.	See "Creating C++ components for multiplatform clusters" on page 271 for details on this feature.
Prog ID (*only for ActiveX components*)	The progid that the component uses in the COM Automation Server Registry.	EAServer Manager does not register the ActiveX component DLL. Before running the component, you must register the DLL with the Windows regsvr32 command or by using the registration feature in your ActiveX development tool.
PowerBuilder Class Name *only for PowerBuilder components*	Matches the name of the nonvisual object that implements the component's methods.	Set by PowerBuilder, defined in the EAServer Component Wizard.
PowerBuilder Library List *only for PowerBuilder components*	A list of the PowerBuilder library files that are required to run the object, separated by semicolons. For example: `mylib.pbl;anotherlib.pbl`	Set by PowerBuilder, defined in the EAServer Component Wizard.

Property	Description	Notes
PowerBuilder Application *only for PowerBuilder components*	The name of the PowerBuilder application that contains the NVO that implements the component.	Set by PowerBuilder, defined in the EAServer Component Wizard.

Component properties: Transactions

The Transactions tab configures the component's transactional properties. Chapter 2, "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; at the component level, the setting affects all methods. You can also set a transaction attribute for methods within a component (see "Method properties" on page 77). Values set at the method level override the component setting.

Transaction attribute in imported EJB components
EJB 2.0 or 1.1 components imported from an EJB JAR file have the transaction attribute set for each method. To use the component level setting, set the transaction attribute to Default to Component for each method.

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 EAServer 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, EAServer 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.
- **Never** The component's methods never execute as part of a transaction, and the component may cannot be called in the context of a transaction. If a client or another component calls the component with an outstanding transaction, EAServer throws an exception.
- **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 EAServer 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, EAServer logs an error and returns an exception to the client. Stateful session Beans can create transactions that remain open across several method calls.
- **OTS Style** For non-EJB components only. The component can inherit a client's transaction. If called without a transaction, the component can explicitly begin, commit, and rollback transactions by using the CORBA CosTransactions::Current interface. See Chapter 2, "Understanding Transactions and Component Lifecycles," for more information.
- **Default to component** (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.

EAServer allows only one transaction per component instance
A component instance may not execute in two transaction contexts. You cannot set a transaction attribute at the method level that conflicts with the component level setting. For example, you cannot set the component transaction attribute to Mandatory and a method transaction attribute to Requires New. If a method invocation would cause this rule to be violated, the server returns an exception to the client and logs the error in the server log file.

Transaction isolation level

Specifies the isolation level for transactions begun by the component's methods. This setting can be configured for the component and for individual methods (see "Method properties" on page 77). The choices are:

- Read Committed
- Read Uncommitted
- Repeatable Read
- Serialized
- None (for component only)
- Default to component (for methods only)

Note The transaction isolation level is supported for EJB 1.0 components only.

Automatic demarcation/ deactivation

Applies to components that use a control interface in which the instance activation and deactivation correspond to transaction boundaries. In other words, the option does not apply to EJB components or any omponent that uses the control interface CtsComponents::ObjectControl (the control interface property is "Configuring a control interface" on page 72).

For EJB components and components that use the CtsComponents::ObjectControl control interface, this option is ignored; for these compnents, the Stateless option on the Instances tab determines whether the component is deactivated after every method invocation.

When Automatic demarcation/deactivation is enabled, EAServer deactivates the component instance after every method invocation. Your component need not call the completeWork or rollbackWork transaction primitives when this property is enabled. If your component is transactional, calling rollbackWork or throwing the CORBA TRANSACTION_ROLLEDBACK exception aborts the transaction. Setting any other transaction state commits the transaction.

By default, this option is enabled for new components.

If component is stateful disable Automatic demarcation/deactivation
If your component maintains state across method invocations, you must disable the automatic transaction demarcation property. For example, if you read and modify class member fields in response to method invocations, you must disable this option.

Automatic failover

When this option is enabled, client proxies for the component can transparently failover to alternate servers when a server becomes unavailable. This option cannot be enabled unless you have enabled the Automatic demarcation/deactivation option.

Automatic failover requires that your application use a cluster of servers, so that redundant servers are available to run the application's components. The cluster must include at least one name server and clients must resolve proxy references using naming services. See Chapter 7, "Load Balancing, Failover, and Component Availability," in the *EAServer System Administration Guide* for more information.

Component properties: Instances

Properties on the Instances tab configure how instances of the component are created and bound to server-side threads and client-side object references. Table 4-2 describes the settings:

Table 4-2: Instances tab component properties

Property	Description
Concurrency	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. If this option is disabled, EAServer serializes all method calls to the component. Concurrency applies to execution of all instances. With concurrency disabled, a call to one instance cannot overlap the execution of another instance. If a PowerBuilder component is Shared, disable Concurrency. PowerBuilder is thread safe at the session level only. **Concurrency option disabled** If the Sharing and Bind Thread options are selected, the Concurrency option is implicitly disabled.

Property	Description
Bind Object	Applies to stateful components only (Automatic Demarcation/Deactivation must be disabled on the Transactions tab or the component must be a stateful session EJB). When this property is enabled, an instance is bound to a client's proxy reference until the client destroys or releases the reference. 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. Component instances are destroyed when the client instance reference times out (the time out period is configured on the Instances tab–see "Component properties: Instances" on page 61). Instances are not pooled. Bind Object is most commonly used for storage components, which are used to store a component's state information in a database. See "Component properties: Persistence" on page 67 for more information on storage components.
Bind Thread	When this option is enabled, component instances are bound to the creating thread. Enable this option if the component uses thread-local storage. For ActiveX components, this option must be enabled. For other component types, enable the option only if you are sure that your component uses thread-local storage. 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. When Bind Thread is enabled, instances are pooled if the Pooling option is enabled. The thread is pooled with the instance in this case.
Pooling	When this option is enabled, component instances are always pooled after deactivation. For Java and ActiveX components, you can also configure pooling by implementing interfaces with a `canReuse` (Java) or `canBePooled` (ActiveX) method. If you enable the Pooling option in EAServer Manager, your component is always pooled, and these methods are not called. See "Supporting instance pooling in your component" on page 16 for more information on instance pooling.
Sharing	When this option is enabled, a single, shared instance of the component services all client requests. 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. If a PowerBuilder component is Shared, disable Concurrency. PowerBuilder is thread safe at the session level only. **Sharing setting overrides Pooling setting** If you select both Sharing and Pooling, Sharing takes precendence.

Property	Description
Stateless	This option applies only to EJB session Beans and non-EJB components that use the control interface CtsComponents::ObjectControl. For EJB session Beans, the Stateless option is set correctly when the component type is set, and must not be changed. For other component types, the option must be set manually.
Transient	Applies to stateful components only. Specifies whether instances can be run on multiple servers in a cluster or survive a server restart. If this option is enabled, the server guarantees that proxy references can only be used within the same server process. 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 "Component properties: Persistence" on page 67).
Reentrant	Applies to entity components only (including EJB entity Beans). 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.
LWC	Applies to EJB components only. Enables the EAServer lightweight container (LWC) for intercomponent EJB invocations or calls to EJBs from servlets and JSPs hosted in the same server. For more information, see "Lightweight container" in the *EAServer Performance and Tuning Guide*.
LWC/ Skeleton Support	If LWC is enabled, the Skeleton Support option enables calls to the component from servlets and JSPs hosted in the same server. Such calls are not supported unless this option is set.

Component properties: Environment

Applies to EJB 2.0 or 1.1 components only. Environment properties allow you to specify read-only site specific data for use by the component. For example, you may have environment properties to specify the name of a logging file, or to tune cache usage, or to specify an email address for the site administrator. See "Configuring environment properties" on page 133 for more information.

Component properties: EJB Local Refs

Applies to EJB 2.0 components only. EJB local references provide an alias mechanism for the JNDI names used to call other EJB components using local interfaces. The JNDI names used in your compoent must be cataloged on this tab. When deploying the component, a site administrator can map site-specific EJB JNDI names to the references used by your component. To add or edit local references, follow the instructions in “Adding an EJB local reference” on page 384, or “Editing an EJB local reference” on page 384, respectively.

Component properties: EJB Refs

Applies to EJB 2.0 or 1.1 components only. EJB references provide an alias mechanism for the JNDI names used to create proxies for intercomponent calls to non-EJB components or to EJB components using the EJB remote interface. The JNDI names used in your compoent must be cataloged on this tab. When deploying the component, a site administrator can map site-specific EJB JNDI names to the references used by your component. To add or edit references, follow the instructions in “Adding an EJB reference” on page 383, or “Editing an EJB reference” on page 383, respectively.

Component properties: Resource Refs

Applies to EJB 2.0 or 1.1 components only. Resource references are used to obtain database connections and JavaMail sessions. The reference allows you to obtain resource factories using JNDI, rather than hard-coding connection parameters in your application. See “Configuring resource references” on page 133 for more information.

Component properties: Resource Environment Refs

For EJB 2.0 components, resource environment references are logical names applied to objects administered by EAServer.

❖ **Adding or editing a resource environment reference**

1 Open the Component Properties dialog box.

2 “Resource environment references” on page 386 describes how to add and edit a resource environment reference.

Component properties: Role Refs

Applies to EJB 2.0 or 1.1 components only. Role references are required if you call the isCallerInRole Java method to restrict access. Each reference maps a string used in isCallerInRole calls to a J2EE role that is configured in the package Role Mappings. See "Configuring role references and method permissions" on page 133 for more information.

Component properties: Resources

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 EAServer transaction. See Chapter 2, "Understanding Transactions and Component Lifecycles" for more information on EAServer transactions.

  You can set the timeout for components and at the server level, with server property com.sybase.jaguar.server.tx_timeout (set on the Advanced tab in the Server Properties dialog box). EAServer determines the transaction timeout period as follows:

  - If the component Transaction Timeout property is set to a non-zero value, this is the timeout period.
  - Otherwise, the server transaction timeout property is checked (the server transaction timeout is specified by the `com.sybase.jaguar.server.tx_timeout` property). If the server transaction timeout is non-zero, this specifies the timeout period.
  - Otherwise, the component Instance Timeout value is checked. If this value is non-zero, this specifies the transaction timeout period as well as the instance timeout period.
  - Otherwise, the transaction timeout is infinite.

  For both the component and server setting, the timeout period is configured in seconds, with 0 indicating infinity (that is, no timeout). The default for a new server is 0. When specifying timeouts, a resolution of 5 seconds is recommended. 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.

EAServer 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.

- **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 17 explains this term.) The setting has no effect for stateless components.

  When the timeout period is exceeded, EAServer 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 Advanced 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.

- **Maximum Active Instances** Specifies the maximum number of instances that can exist at the same time. For a C++ component that runs as an external process, specifies the maximum number of simultaneously running external processes. If a request arrives when the maximum number of instances exist and are all busy, the request blocks, with blocking time constrained by the Maximum Wait setting.

- **Maximum Pooled Instances** When instance pooling is enabled with the Pooling checkbox on the Instances tab, specifies the maximum pool size. If the maximum pool size is reached, EAServer destroys excess instances after deactivation. The default is 0, which means no maximum pool size is in effect.

- **Minimum Pooled Instances** When pooling is enabled, specifies the minimum pool size. The default is 0.
- **Maximum Wait** This setting applies only when the Maximum Active Instances property is set to specify a limit on the number of simultaneous active instances. If a request arrives when the maximum number of instances exist and are all busy, the request blocks, with blocking time constrained by the Maximum Wait property. If the blocking time expires, the caller receives a CORBA::NO_RESOURCE_EXCEPTION.
- **Named Instance Pool** Constrains the component to run in the specified instance pool.

For information on tuning the instance pool size properties, see "Instance pooling" in Chapter 3, "Component Tuning," in the *EAServer Performance and Tuning Guide*.

Component properties: Persistence

The Persistence tab allows you to specify an EJB entity bean's primary key and configure settings that allow EAServer to save component state to a database server or using inter-server in-memory replication to support failover in clustered deployments. For more information, see these chapters:

- Chapter 27, "Creating Entity Components"
- Chapter 28, "Configuring Persistence for Stateful Session Components"

Component properties: Run-As Identity

This tab applies to EJB 2.0 components only. These properties specify the authentication credentials that are used when methods call other components. By default, the client credentials are used. You can specify an alternate credential with these settings:

- **Run as** Choose `specified` to specify an alternate identity. The default, `client`, means the component calls made from this component inherit the client identity.
- **Role** Specify a role name. The identity specified in the Mapped to Jaguar identity field should be in this role. This name is used if the component is exported to an EJB-JAR file.
- **Run as identity** Specify a logical identity name.

- **Mapped to Jaguar identity** Choose an EAServer identity from the pull down menu. This is the identity with which the component executes.
- **Description** Enter an optional text comment. This field can be used to provide identity mapping instructions for the deployer when the component is deployed to another server.

The Existing Mappings on the Package table displays logical identity names that are mapped to EAServer identities by components in the same package.

To enable use of the run-as identity for EJB component calls to remote servers, you must specify `corbaname` URLs in the EJB Reference properties for the EJB component that issues the call. For more information, see "Interoperable naming URLs" on page 157 and "Configuring EJB references" on page 132.

Component properties: Run-As Mode

This tab applies to EJB 1.0 components only. These properties specify the user name and password that are used when methods call other EAServer components installed in the same server or server cluster. The setting in the Component Properties dialog box is the default for all methods in the component. You can override the component level setting for individual methods (see "Method properties" on page 77).

The choices for Run As Mode are:

Value	Description
Client	Use the client's user name and password. This is the default setting.
System	Use the system user name and password. The system account effectively belongs to any role, and can execute any method on any component that is installed in the same server or cluster.
Specified	Use the user name and password associated with the identity name specified in the Run-as Identity column.

When Run-As Mode is set to Specified, you must enter a logical entity name, then map the logical identity name to an identity that is defined in the EAServer Manager identities folder. If there are no identities defined, you must close the Component Properties dialog, go to the Identities folder, and create at least one identity to map logical identities to. For more information, see "Configuring identities" in the *EAServer Security Programming and Administration Guide*.

Mappings specified in the Component Properties dialog are stored as package properties and apply to all other components in the package.

Component properties: Java Classes

For CORBA/Java and EJB components, allows you to define the list of classes that must be custom loaded in addition to the component implementation class. Custom loading allows you to refresh the component installation without restarting the server, and to deploy classes in JAR files without changing the server CLASSPATH environment variable and restarting the server. For more information, see Chapter 30, "Configuring Custom Java Class Lists."

In versions earlier than EAServer 4.0, this property was configured by setting the `com.sybase.jaguar.component.java.classes` property on the Advanced tab.

Component properties: Additional Files

Configures the com.sybase.jaguar.component.files property, which specifies additional files that are included when the component is archived in Jaguar JAR format or replicated with the synchronize feature.

By default, the following files are included when you export packages or synchronize between servers:

- The IDL files that define interfaces and types used by the component.
- For C or C++ components, the DLL or shared library that is specified on the General tab of the Component Properties window. If your component requires additional DLLs or shared libraries, you must specify them in the list.
- For Java components, the implementation class, any classes listed in the Java Classes tab, and stub classes listed in the com.sybase.jaguar.component.files.corbastubs, and com.sybase.jaguar.component.files.ejbstubs properties.
- For PowerBuilder components, the libraries starting with $ (dollar sign) that are referenced by the property com.sybase.jaguar.component.pb.librarylist

Note Java and C++ stubs are not included by default in the component's file set. These can be regenerated on the target server after synchronization or installing the archive. If you do not want to regenerate, add the stub files to the list on the Additional Files tab.

Any additional files that are required to run the component must be listed on the Additional Files tab. Use the Additional Files wizard to enter a list of file names, separated by commas. Files may be specified as follows:

- Specify Java classes and packages using the Java dot notation. For example, `com.sybase.CORBA` adds all files in the com.sybase.CORBA package. These classes must be deployed under the EAServer *java/classes* subdirectory.
- If a DLL or shared library is deployed in the EAServer *cpplib* subdirectory, you can enter the filename itself. For example `myutils.dll`.
- Other files must be specified using full paths or paths that are relative to the EAServer *Repository* subdirectory. For example, *../dll/debug/MyDebugLibrary.dll* or *d:\mydir\myfile.ext*. If you use full paths, you will only be able to synchronize or import package archives on machines which share the same directory structure as your development machine.

Configuring the Additional Files list

When you include additional files, you can either enter the file names individually, or you can use the Additional Files wizard to add multiple files, packages, classes, and directories.

❖ **Entering file names individually**

1. Click Add. This opens the Add a File Name to the List dialog box.
2. Enter the file name and click Ok.

❖ **Adding multiple items**

1. Click Additional Files Wizard. This open the Additional Files dialog box. Each item that you add is appended to the list.
2. To add Java packages or classes:
   a. Click Browse
   b. Choose a **.class* file and click Select.

   The class files must be deployed under EAServer's *java/classes* directory.
3. To add files or directories:
   a. Optionally, specify a file filter, such as **.txt*.
   b. Optionally, select to use the JAGUAR environment variable.
   c. Click Browse.
   d. Choose a file or directory and click Select.

4 To add property files from other entities:

   a Click Browse.

   b Choose a **.props* file from under the *Repository* directory and click Select.

5 To add file lists from other entities:

   a Click Browse.

   b Choose an entity's **.files* file and click Select.

6 Click Add Files to Additional Files List.

Component properties: JAXP Support

For EJB 2.0 components, configures the JAXP, DOM, and XSLT parser implementations used in the component. See Chapter 36, "Configuring Java XML Parser Support," for more information on these properties.

Component properties: Advanced

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

For information on repository properties, see Appendix B, "Repository Properties Reference," in the *EAServer System Administration Guide*.

❖ **Setting 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:

   a Enter the property name in the Name field

   b Enter the value in the Value field.

3 If modifying a property, edit the displayed value in the Modify Property window.

When to use the Advanced tab
Though you can use the Advanced tab to set any property prefixed with "com.sybase.jaguar.component", Sybase recommends that you use this tab to set properties only as specified by the EAServer documentation or by Sybase Technical Support. Most properties can be configured graphically elsewhere in the EAServer Manager user interface.

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

Configuring a control interface

The `com.sybase.jaguar.component.control` property pecifies the name of the component's IDL control interface. EAServer calls each control interface method 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.
JaguarEJB::ServerBean	A lifecycle model based on the EJB 0.4 specification. This is the default for Java/CORBA components that do not have persistent state (that is, when the Persistence field is None).
CtsComponents::ObjectControl	A CORBA lifecycle model based on the EJB entity bean model. The default for Java/CORBA and C++/CORBA components with persistent state (that is, when the Persistence field is Component Managed).
JaguarCOM::ObjectControl	For ActiveX components.

These interfaces are documented in the generated IDL documentation, which is available in HTML format in the *html/ir* subdirectory of your EAServer installation. If you use a control interface other than JaguarEJB::ServerBean, EAServer generates the control interface methods in the implementation template when you generate a C++ or Java skeleton.

CHAPTER 5

Defining Component Interfaces

A component's interfaces define the methods that clients can invoke. Though EAServer stores interface information in CORBA IDL, you do not need to know IDL in order to define interfaces in EAServer Manager. You can define interfaces using several techniques, as described in the sections of this chapter.

Related documents

If you are developing components with PowerBuilder, you can visually define component interfaces within the PowerBuilder development environment. See the *Application Techniques* manual in the PowerBuilder documentation for more information.

Defining interfaces graphically

A component's Interfaces folder contains icons for the IDL interfaces implemented by the component. These interfaces define the methods that can be called by client applications.

When you define a new component, EAServer Manager creates an interface for the component. If you defined the component by importing a Java or ActiveX class, the interface contains IDL definitions matching the Java or ActiveX methods. If you are defining a component from scratch, EAServer Manager creates a new IDL interface with no methods. If you have imported an EJB component, the importer has also created an IDL home interface for the component.

❖ **Adding interfaces**

Use these steps to configure which interfaces a non-EJB component implements, or to add interfaces to an EJB component in addition to the home and remote interfaces:

1. Highlight the Interfaces folder beneath the component icon and choose File | Add Interfaces. The Install Interfaces dialog displays.
2. You can perform the following operations in the dialog:
   - **Add an existing interface** Choose from available interfaces as follows:
     a. Select the module that defines the interface in the dropdown menu.
     b. Highlight the interface name from the list of interfaces on the right side of the dialog box.
     c. Click Add.
   - **Add and define a new interface** Use these steps if the interface you want to implement in the component does not exist. You can add methods to the interface after exiting the dialog:
     a. Enter a name for the new interface, in the format module::interface.
     b. Click Add New.
3. When you are done, click Install to close the dialog.
4. If you defined new interfaces, add methods to the interfaces as described below.

❖ **Removing interfaces**

These steps remove an interface from the list of interfaces supported by a component, but do not affect the IDL definition:

1. Expand the Interfaces folder beneath the component icon. The list of interfaces supported by the component displays.
2. Highlight the interface to remove, and choose File | Remove.

Rules for removing interfaces:

- An EJB component must have a home and remote interface; you cannot remove them. You can change them as described in "Changing the EJB remote or home interface" on page 75.
- A non-EJB component must support at least one interface to serve as its remote interface. You cannot remove the last interface supported by a component; instead, change the remote interface as described in "Changing the EJB remote or home interface" on page 75.

❖ **Changing the EJB remote or home interface**

Home and remote interfaces are used by EJB clients and components. To change a component's home or remote interface:

1 Expand the Interfaces folder below the component icon.

2 Highlight the Interfaces folder and choose File | Set Home interface if changing the home interface, or choose File | Set Remote Interface if changing the remote interface.

3 Enter an IDL name for the interface, specifying an IDL module hierarchy and interface name to match the intended Java package hierarchy and interface name. For example, if the Java class will be foo.bar.MyBeanHome, enter foo::bar::MyBeanHome. EAServer Manager creates specified modules and interfaces if they do not already exist.

Editing interfaces

You can edit method signatures graphically in the Method Properties dialog, or by editing CORBA IDL directly. Some method properties such as the transactional attribute and run-as mode properties are not defined in IDL. To configure these settings, you must display the Method Properties window.

Interface and method name conventions
Sybase recommends that you begin interface names with a capital letter, and method names with a lowercase letter.

❖ **Editing methods in IDL**

You define or edit method signatures by editing the CORBA IDL interface definition directly. Use this procedure if you are comfortable with CORBA IDL and prefer it to point-and-click interface editing.

1 Highlight the icon for the interface of interest.

2 Choose File | Edit IDL

3 The IDL interface definition displays in the IDL Editor window.

4 Make any changes, then choose File | Save. If the changes have introduced syntax errors, EAServer Manager displays the error text in a dialog box. Fix the errors, then try to save again.

5 When you have made all changes and saved them, choose File | Exit.

❖ **Adding methods graphically**

1 Highlight the icon for the interface of interest.

2 Choose File | New Method.

3 Enter a name for the method and click Create New Method.

4 The Method Properties dialog box displays. Use the controls on the General tab to define the method parameters, return type, and exceptions raised. See “Method properties” on page 77 for more information.

❖ **Editing method properties**

Use this procedure to display the method properties window, which provides a point-and-click controls to edit the method’s IDL signature and also configures settings that are not represented in IDL.

1 Highlight the icon for the interface of interest.

2 Highlight the icon for the method to be edited.

3 Choose File | Method Properties.

4 The Method Properties dialog box displays. See “Method properties” on page 77 for more information.

❖ **Removing methods graphically**

1 Expand the icon for the interface of interest.

2 Highlight the icon for the method to be deleted.

3 Choose File | Delete Method.

Method properties

Method properties are organized on the following tabs:

- **General** Shows the method's return type, parameters, and the exceptions that can be thrown. The fields are described in the table below:

Table 5-1: General method properties

Property	Description	Notes
Description	Specifies a description of what the method does. The description can be up to 255 characters.	You can enter comments about the method here.
Exceptions Raised	Specifies the user-level exceptions raised by this method. Enter exceptions in the form: *Module*::*ExceptionName* You can enter multiple exceptions, separated by commas.	User-level exceptions must be defined in IDL before you can specify that a method raises the exception. See "User-defined exceptions" on page 98 for more information.
Read only	Applies to entity components that use component-managed persistence (also called bean-managed persistence). Specifies whether the method can change the instance state.	For best performance, set this property for all entity component business methods that do not modify the instance state. When this property is enabled, the entity components ejbStore or ctsStore method is not invoked after the business method returns. This property has no effect on entity components that use automatic persistence. The ejbStore or ctsStore method is always called, but never performs data storage.
Returns	Specifies the return type of the method. Select the return type from the drop-down list or enter the name of an IDL or Java datatype into the input field. See "Parameter and return value datatypes" on page 79 for more information.	Method implementations cannot return null values. If there are cases where the method must return no value, specify an IDL sequence type as the return value, and implement the method to return an empty sequence to indicate the no-value case.

Property	Description	Notes
Parameters	Displays the name, type, and mode of each parameter.	To add a parameter, click Add, and complete the information described in "Parameter properties" on page 78. To modify a parameter, highlight the parameter you want to modify, click Modify, and complete the information described in "Parameter properties" on page 78. To delete a parameter, highlight the parameter you want to delete, and click Delete.

- **Transactions** Allows you to configure transaction properties for the method. The Transaction Attribute and Transaction Isolation Level settings have the same meaning as the like-named settings for components. See "Component properties: Transactions" on page 58.
- **Permissions** Applies to EJB 2.0 and 1.1 components only. The settings allow you to restrict which users can invoke the method. See "Configuring role references and method permissions" on page 133 for more information.
- **Run-As Mode** Applies to EJB 1.0 components only. Allows you to configure the user name and password for to be used for intercomponent calls. These settings have the same meanings as the Run-As Mode component settings. See "Component properties: Run-As Mode" on page 68.
- **Advanced** Allows you to manually edit method property settings in the EAServer configuration repository. For advanced users.

Parameter properties

The New Parameter and Modify Parameter dialog boxes allow you to configure the type and modality of each method parameter using the controls described in Table 5-2.

Table 5-2: Parameter properties

Property	Description	Notes
Parameter Number	Displays the parameter number	No input is required.
Name	Specifies the name of the parameter	A name is required.
Mode	Specifies how values are passed for the parameter.	Supported modes are as follows: • *in* – Input only. No new value is returned when the method completes. • *inout* – Input and output. Input values are not ignored, and output values are returned to caller. • *out* – Output only. Input values will be ignored; output values are returned to caller.
Type	Specifies the datatype of the parameter.	Select a datatype from the drop-down list or type the name of an IDL or Java datatype in the input field. See "Parameter and return value datatypes" on page 79 for more information.
Description	Describes how the parameter is to be used.	Optional. You can use this field to describe how the parameter is to be used.

Parameter and return value datatypes

For method parameters and return values, you can choose predefined types from the drop-down list or enter a Java or IDL datatype name by typing it in the input field.

Predefined datatypes

The following table lists the predefined EAServer Manager datatypes and their IDL equivalents. These types display when you change the datatype of a parameter or change the method's return type.

Table 5-3: Predefined EAServer IDL datatypes

EAServer Manager display datatype	CORBA IDL type	Description
boolean	boolean	One bit of binary data; a value that is either true or false
integer<16>	short	A 16-bit integer
integer<32>	long	A 32-bit integer
integer<64>	long long	A 64-bit integer
float	float	Single-precision IEEE floating point numbers
double	double	Double-precision IEEE floating point numbers
string	string	A sequence of characters of any length
binary	BCD::Binary	Sequence of bytes
decimal	BCD::Decimal	Fixed-point decimal
money	BCD::Money	Same as decimal
date	MJD::Date	A date including year, month, day, hour, minute, second, and millisecond values
time	MJD::Time	Holds the time of day, including hours, minutes, seconds, milliseconds
timestamp	MJD::Timestamp	Holds the same data as date, plus a nanoseconds value
ResultSet	TabularResults::ResultSet	A single table of relational database rows
ResultSets	TabularResults::ResultSets	A sequence of 0 or more ResultSet objects

Using IDL and Java datatypes

In addition to the predefined types listed in “Predefined datatypes” on page 79, you can also apply IDL and Java datatypes to parameters and return values by typing the name of an IDL or Java datatype.

IDL datatypes You can define your own datatypes and use them when defining method signatures. “Defining modules, interfaces, and types in IDL” on page 85 discusses IDL in more detail.

To specify an IDL type name in the Method Properties dialog box, simply enter the type name in the Returns or Parameter Datatype field—for example, `MyModule::MyType`. The IDL module must be present in the EAServer IDL repository, and the module must contain a declaration for that type name.

Java datatypes You can specify Java datatypes as input parameters or return types. You cannot specify Java datatypes for parameters that use the inout or output modes. "Java class names used as IDL datatypes" on page 97 describes which interfaces and classes can be used.

To specify a Java datatype, simply type the full class or interface name in the Returns or Parameter Datatype field—for example, `java.util.Properties`.

Java datatypes and interoperability
If a method is defined using a Java datatype, only Java components can implement the method and only Java clients can invoke the method.

Importing interfaces from compiled Java files

EAServer Manager provides a Java import feature that creates a component definition by reading method definitions from a compiled Java class or interface file (to import a JavaBeans component, you must specify the class that implements the JavaBeans component). The import process creates a corresponding IDL interface in the EAServer interface repository. This feature is primarily used to adapt existing Java classes to be run as Java components within EAServer. However, you can import a Java interface to define a component of any type.

Coding classes, interfaces, and JavaBeans for import

Before using the importer for the first time, you should read this section to understand how Java methods are mapped to EAServer component methods.

Determining eligible methods

Each method in a class or interface (including those inherited from a base class or interface) is inspected to see if they use allowable parameter and return types. Suitable methods are added to the component's IDL interface. EAServer Manager displays warning dialog boxes describing any methods that are not imported. The importer accepts methods that use the following datatypes:

- **Java equivalents for the predefined EAServer datatypes** "Choose implementation datatypes" on page 185 describes the Java equivalents for the predefined EAServer datatypes. If your component uses the IDL/Java datatype mappings, the importer sets the Component Type field to `Java/IDL`. Otherwise, the Component Type field is set to `Java/JDBC`. If you import a class that ran as an EAServer version 1.1 component, it will be assigned the Java/JDBC component type.

  An inout parameter must use the **holder** classes as described in "Choose implementation datatypes" on page 185.

- **User-defined classes** With restrictions, user-defined classes are allowed as parameters or return types. The importer creates an IDL definition to match the class. User-defined classes must contain only fields (no methods). Fields may use the Java equivalents for predefined EAServer datatypes, as described above.

  For an inout parameter, the Java method definition must use a holder class that you have created, as described in "Holders for user-defined classes and arrays" on page 82.

- **Single-dimension arrays** Single-dimension arrays are allowed as parameters or return types. The base type can be any Java equivalent for the predefined EAServer types or a user-defined class. (User-defined classes are subject to the restrictions noted above.)

  For an inout parameter, the Java method definition must use a holder class that you have created as described in "Holders for user-defined classes and arrays" on page 82.

The method can throw any exception, but only exceptions that extend org.omg.CORBA.UserException are added to the IDL method's raises clause.

Holders for user-defined classes and arrays For an inout parameter declared as a user-defined class or a single-dimension array, the Java method definition must use a holder class that you have created. For a user-defined class, the template for the holder class is as follows:

```
package comp-package;

class TypeHolder {
    Type value;
    // Default constructor:
    TypeHolder();
    // Initial-value constructor:
    TypeHolder(Type value);
}
```

where

- *comp-package* is the same package that contains the class or interface that you are importing.
- *Type* is the user-defined class name.

For an array, the template for the holder class is as follows:

```
package comp-package;

class TypenameHolder {
    BaseType value;
    // Default constructor:
    TypenameHolder();
    // Initial-value constructor:
    TypnameHolder(BaseType[] value);
}
```

where:

- *comp-package* is the same package that contains the class or interface that you are importing.
- *Typename* is a legal Java identifier. The importer will create an IDL typedef statement for the array type using this declaration.
- *BaseType* is the base type for the array.

Importing Java interfaces

Methods to be imported from a Java interface must adhere to the restrictions described in “Determining eligible methods” on page 81. In addition, the interface cannot contain any fields.

You must specify a class that implements the interface before you can run the component. Specify the implementation class name in the Component Properties window (see “Component properties: General” on page 54). Make sure that the class has a default constructor (that is, a constructor with no arguments). EAServer calls the default constructor to create new component instances.

Importing Java classes

Methods to be imported from a Java class must adhere to the restrictions described in “Determining eligible methods” on page 81. In addition, the class must have a constructor method with zero parameters. Other constructors are not called by EAServer.

Note Classes that implement the ServerBean interface can be imported. The ServerBean methods are not added to the component’s IDL interface.

Importing JavaBeans components

To import method definitions from a JavaBeans component, you choose the Java Class option on the import screen, then specify the name of the class that implements the component.

Methods to be imported from a JavaBeans component must adhere to the restrictions described in "Determining eligible methods" on page 81. In addition, the class must have a constructor method with zero parameters. Other constructors are not imported.

The add and remove methods for the JavaBeans event listeners are not imported.

Importing a Java class or interface in EAServer Manager

❖ **Importing a Java class or interface in EAServer Manager**

1. If necessary, create the EAServer package that will contain the component. See "Creating a new package" on page 42 for details.
2. Specify the package to install the component in as follows:
   a. Double-click the Packages folder to expand it.
   b. Highlight the package to which the component will be added.
3. Choose File | New Component from the menu.
4. In the Component Wizard dialog box, select Import from Java File, and click Next.
5. Verify that the displayed importer CLASSPATH contains the JAR files and directories required to instantiate the bean's classes, specifically:
   - Verify that the code base under which the class files are deployed is included.
   - If the classes are in a JAR file, verify that the full path to the JAR file is included.
   - If the class definitions require other JAR files or directories not listed, list them as well.

   If necessary, use the Add, Modify, Delete, Move Up, and Move Down buttons in the Component Wizard to modify the CLASSPATH. The displayed CLASSPATH affects only this importer session, not the EAServer process.
6. Enter the component name in the Import Java Class File dialog box.

7 Choose the type of file to be imported:

- **Java Class** See “Importing Java classes” on page 83.
- **Java Interface** See “Importing Java interfaces” on page 83.

8 If importing a Java interface, choose the type of component to be defined in the drop-down list.

9 Browse for the Java class file that contains the class or interface that is being imported.

The importer will read the specified file, define an IDL interface as described below, and define a component that implements the IDL interface.

Importing interfaces from registered ActiveX components

EAServer Manager can import method signatures from ActiveX DLLs or type library files. This feature allows you to adapt a nonvisual ActiveX automation server as an EAServer component.

For more information on this feature, see “Importing ActiveX components” on page 328.

Defining modules, interfaces, and types in IDL

EAServer stores all component interfaces in Interface Definition Language (IDL) modules. In EAServer Manager, the IDL folder displays all modules available in EAServer’s interface repository.

Learning IDL

IDL is defined by the Object Management Group as a standard language for defining component interfaces.

Chapter 3, “OMG IDL Syntax and Semantics,” in the *CORBA V2.3 Specification* defines IDL. Printable versions of this document can be downloaded from the following URL:

```
http://www.omg.org/corba/index.html
```

Creating and editing IDL modules, interfaces, and types

EAServer Manager displays IDL modules as folders beneath the top-level IDL folder. Modules can be nested, that is, a module may be defined within another module.

❖ **Navigating nested IDL modules**

Follow this procedure to view the IDL entities defined within a module.

1 Expand the top-level IDL folder.

2 Each icon in the IDL folder represents a top-level IDL module. To navigate to a nested module, click the + sign next to the parent module’s icon, or double-click the parent module’s icon.

3 In the left pane, highlight the module of interest. EAServer Manager display the types and modules defined within the highlighted module in the right pane.

❖ **Defining new IDL modules**

1 If defining a new top-level module, highlight the IDL folder.

If defining a nested module, follow the steps in “Navigating nested IDL modules” on page 86 to highlight the parent module.

2 Choose File | New IDL Module. Enter the module name and click Create New Nested Module IDL. Module names must begin with a letter.

3 EAServer Manager displays the empty module definition in the IDL Editor window. Optionally make the following changes:

a Edit the HTML documentation comment and add a description of the module.

b If the module will contain datatypes and interfaces (and not just nested modules), optionally specify the Java package for stubs as described by “Specifying Java package mappings for IDL modules” on page 87.

4 When done, choose File | Save, then File | Exit to close the IDL Editor window.

❖ **Specifying Java package mappings for IDL modules**

- If an IDL module contains datatypes and interfaces (and not just nested modules), you can specify the Java package to be used for generated Java stubs. Stubs for each type of Java client must be in different packages, or deployed under different code bases.

  If you do not specify a Java package mapping, 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.

  Change the Java package mapping for a module by editing one of the following files:

  - *Repository/IDL/ejb.props* specifies the Java packages for EJB stubs.
  - *Repository/IDL/java.props* specifies the Java packages for Java/CORBA stubs.
  - *Repository/IDL/jdbc.props* specifies the Java packages for EAServer 1.1 stubs.

  To change the default Java package, create or edit an entry in the appropriate file with this format:

  ```
  idl-module=dotty-package
  ```

  Where:

  - *idl-module* is the IDL module name, for example, com::sybase::test::MyModule
  - *dotty-package* is the dot-format Java package name, for example, com.sybase.test.corba.

  For compatibility with IDL created in previous releases, EAServer also allows you to specify the Java package in a doc comment directives above the module declaration. These directives are translated to entries in the *java.props*, *ejb.props*, or *jdbc.props* files. You can enter multiple directives to specify packages for stubs of different types. Each package directive has the form:

  ```
  /*
  ** <!-- typePackage: dotty-package -->
  */
  ```

Where *dotty-package* is the dot-format Java package name and *type* is one of:

- `java`, if specifying the package for CORBA stubs.
- `ejb`, if specifying the package for EJB stubs.
- `jdbc`, if specifying the package for Jaguar 1.1 client stubs.

You can also create or change Java package mappings when generating stubs for the IDL module in EAServer Manager. Highlight the IDL module and choose File | Generate Stubs. Choose the stub type and enter a different Java package name in the Java Package field.

❖ **Creating IDL types, exceptions, and interfaces**

Follow this procedure to define new datatypes and exceptions in a module. You can also define new component interfaces with this procedure, but it is easier to define interfaces using the component's Interfaces folder (see "Defining interfaces graphically" on page 73).

1 Navigate to and highlight the module where the entity is being created, as described in "Navigating nested IDL modules" on page 86.

2 Choose File | New IDL Entity.

3 In the New IDL Entity dialog box, enter a name for the type or interface, then choose the type of entity being created. Click Create New IDL Entity.

EAServer Manager displays a template for the new IDL definition in the IDL Editor window.

4 Finish the definition, then choose File | Save and File | Exit to close the IDL Editor window.

EAServer allows forward IDL references
You can create new IDL types that refer to other IDL types that do not yet exist; among other benefits, this feature allows you to create mutually recursive interface definitions. However, you must be sure that all references are resolved before you can generate stubs and skeletons. When generating stubs and skeletons, EAServer Manager will report errors for any unresolved type references.

❖ **Editing IDL types, exceptions, and interfaces**

To edit or delete a type, exception, or interface:

1 Navigate to and highlight the module where the entity is being created, as described in “Navigating nested IDL modules” on page 86.

2 The module’s types, exceptions, and interfaces display in the right pane of the EAServer Manager window.

3 To edit an item, highlight it and choose File | Edit Entity IDL. Make your changes in the IDL editor window, save them, and close the window.

4 To delete an item, highlight its icon and choose File | Delete.

Unreferenced IDL definitions

The interfaces, types and exceptions associated with a component are not deleted when you delete the component from EAServer Manager unless you delete the package or application where it is installed and specify full deletion. Unused definitions cause no harm. When generating Java stubs, stub classes are generated for all types in a module, regardless of whether the component references them. You can delete unreferenced IDL types to prevent the generation of unnecessary Java stub classes. Verify that no other component references an IDL definition before deleting it.

When deleting packages, you can delete everything associated with the package, including IDL definitions, by choosing full deletion as described in “Modifying packages” on page 45.

Using the IDL editor window

The IDL editor window is displayed when you create a new module or interface. You can also display the source code for datatypes, exceptions, and interfaces by right-clicking on their icons and choosing Edit IDL from the popup menu.

The File menu contains the following options:

Option	Description
Open	Allows you to replace the editor’s current contents with the contents of an operating system file.
Save	Saves your changes in the EAServer IDL repository. When you save to the repository, EAServer Manager checks the syntax of the module or declaration and displays any syntax errors.

Option	Description
Save As	Allows you to save the contents of the editor window into a specified file. This option can be used to *export* IDL definitions of EAServer interfaces for use with other vendor's CORBA ORB implementations.
Exit	Closes the editor window without saving.

The current IDL editor does not have menu commands for copying, cutting, and pasting text. However, you can use the standard keyboard commands for your platform as described below:

Platform	What you do
Windows	Use the mouse to select text. Use Control+C to copy, Control+V to paste, and Control+X to cut.
UNIX (all)	Use the mouse to select text. Key mappings are defined by your X-Windows display configuration. Most workstation keyboards have Copy, Cut, and Paste keys that work as labeled with the manufacturer's default X-display configuration. See your X-Windows system documentation for more information.

Creating and editing interfaces

Interfaces can be added in EAServer Manager, creating a blank interface declaration, or you can declare the interface yourself by editing the module's IDL definition.

Choosing an interface name
Interface names are restricted as follows:

- Interfaces within a module must have unique names, irrespective of case. That is, you cannot define MyInterface and Myinterface in the same module.
- The interface cannot have the same name as the module that contains it.

Sybase recommends that you begin interface names with a capital letter, and operation names with a lowercase letter.

Supported preprocessor directives
No IDL preprocessor directives other than `#include` are supported.

❖ **Creating new interfaces in EAServer Manager**

1 Highlight the module's icon and choose File | New IDL Entity.

2 Type the name of the new interface, choose Interface in the dropdown list of IDL entity types, and click Ok.

3 Click Ok.

4 EAServer Manager displays a new, blank interface in the IDL Editor window. Edit the declaration if needed.

5 When done, choose File | Save, then File | Exit to close the IDL Editor window.

❖ **Editing an existing interface**

1 Select the interface's icon and choose File | Edit IDL.

2 EAServer Manager extracts the interface definition from the module and displays it in the IDL editor window.

3 Edit the declaration as needed.

4 When done, choose File | Save, then File | Exit to close the IDL Editor window.

IDL interface declarations

Interfaces are declared as shown below:

```
interface InterfaceName [: BaseInterface1,
BaseInterface2, ...] {
  operations
};
```

where:

- *InterfaceName* is the name of the interface.
- *operations* is a zero or more of IDL operation declarations. You can enter operations directly as IDL, or use EAServer Manager to define them graphically (see "Operation declarations" on page 93).
- *BaseInterface*, *BaseInterface2*, and so forth form an optional list of existing interfaces from which the new interface inherits definitions. If a new interface inherits from other existing interfaces, the existing interfaces that are inherited from are referred to as *base* interfaces, and the new interface is referred to as a *derived* interface.

For example, this interface, StockComponent, inherits from no other interface:

```
interface StockComponent {
};
```

This interface, C, inherits from interfaces A and B:

```
interface C : A, B {
}
```

Interfaces that inherit definitions from other interfaces are subject to the following constraints:

- *Operations and attributes* cannot be redefined in the new interface.
- *Operation and attribute names* defined in base interfaces must be unique. For example, if a method is defined in both interface A and interface B, you cannot define a new interface that inherits from both B and A.
- *Exceptions, constants, and types* from a base interface can be redefined in the derived interface.
- *References to type names, exception names, and constant names* that are used in multiple derived interfaces must be made unambiguous by prefixing references with the name of the interface that contains the definition of interest. For example, if the constant MAX is defined in both A and B, then A::MAX refers to the definition in A, and B::MAX refers to the definition in B.

The sections below describe how to define operations and attributes for the interface.

Interface stub generation directives

You can embed specially formatted comments in IDL to control the generation of Java stubs for IDL interfaces and structures. Directives must appear in a block comment located immediately before the IDL interface or struct declaration.

Imported class name This directive specifies that a structure or interface was imported from a Java class, and that a new version of the imported class must not be generated when stubs are generated. This directive is most commonly used for EJB home and remote interfaces and EJB primary keys that were defined by importing EJB classes or EJB-JAR files.

The format is:

```
** <!-- imported classname -->
```

Where classname is the Java class name, in dot notation. For example, `foo.bar.MyBeanHome` or `foo.bar.MyBeanPrimaryKey`.

Is home interface This directive identifies an interface as a home interface used by EJB clients and components. If you specify a home interface for a component as described in “Changing the EJB remote or home interface” on page 75, EAServer Manager adds this directive. The format is:

```
** <!-- home -->
```

Finder method return type Applies to multi-object finder methods in an EJB entity bean's home interface. If a finder method's Java form must return java.util.Enumeration, add a doc comment of this form above the IDL finder method declaration:

```
/*
** <!-- java.util.Enumeration -->
*/
::MyModule::MyRemoteList findByName(in string name);
```

See "Defining home interface methods" on page 127 for more information on EJB finder methods.

Operation declarations

Operations in an IDL interface become component methods when the interface is assigned to a component. You can define operations directly in IDL, or graphically as described in "Defining interfaces graphically" on page 73. If you define operations in IDL, follow the structure described here.

Operations are declared as follows:

```
returnType opName
(
[ ... parameterList ... ]
)
[  raises ( ... exceptionList ... ) ] ;
```

where:

- *returnType* is either a valid IDL datatype or void to indicate that the operation does not return a value. "Datatypes for parameters and return values" on page 95 discusses datatypes in detail.
- *opName* is the name of the operation. Sybase recommends operation names begin with a lowercase letter. Names in the same interface must be unique with respect to case, and capitalization of a name must be consistent wherever it is used.

  IDL operation names cannot be overloaded (that is, redeclared with the same return type and different parameter lists). However, you can define IDL operations that map to overloaded C++ or Java methods. To do so, create operation names by appending two underscores and a unique suffix to the method name that will be overloaded. EAServer strips the suffix when generating C++ or Java interface definitions. For example, consider the following IDL:

  ```
  void ov1__double(in double d);
  void ov1__string(in long l);
  ```

When mapped to C++ or Java, these operations translate to the following overloaded methods:

```
void ov1(double d);
void ov1(long l);
```

- *parameterList* is an optional parameter list enclosed in parentheses. The list (but not the parentheses) can be omitted to indicate that the operation takes no parameters. Otherwise, add datatypes and parameter names as shown below:

  ```
  void myMethod
  (
  qual1 type1 param1,
  qual2 type2 param2,
  ...
  );
  ```

  where:

  - *qual1*, *qual2*, and so forth are one of the argument modes in, inout, or out. Use in for parameters that are input-only; no new value is returned when the operation completes. Use inout or out if the operation returns new values for the parameter. An inout parameter's input value is meaningful; an out parameter's input value is not.
  - *type1*, *type2*, and so forth are valid IDL type names (other than the CORBA::Any type). "Datatypes for parameters and return values" on page 95 discusses datatypes in detail.
  - *param1*, *param2*, and so forth are parameter names.
- *exceptionList* is an optional list of user-defined exceptions. If the operation can throw user-defined exceptions, add a raises clause with a list of the IDL user-defined exception names that the operation can throw, as shown below:

  ```
  void myMethod ( in int n )
       raises ( Exception1, Exception2, ... );
  ```

  If the operation can throw only CORBA standard exceptions, omit the raises clause. For more information, see "User-defined exceptions" on page 98.

Attribute declarations

Attributes allow you to associate a value with an interface. IDL attributes are similar in concept to structure fields in languages such as C. However, when mapped to a programming language, attribute values can typically be accessed only by generated functions that allow you to set and retrieve the attribute's value.

Note Attributes are not supported by ActiveX components and clients.

Attributes are declared as shown below:

```
[ readonly ] attribute TypeSpec name;
```

where

- readonly is an optional keyword specifying that the attribute can be retrieved but cannot be set.
- *TypeSpec* is the name of a standard or user-defined type. "Datatypes for parameters and return values" on page 95 describes datatypes in detail.
- *name* is the attribute name.

In C++ and Java, a read-only attribute maps to a method with the same name that returns the attribute type. A writable attribute maps to a pair of overloaded methods with the same name as the attribute. For example, consider the following IDL declarations:

```
readonly attribute long days; // readonly
attribute long months;        // writable
```

In a C++ or Java implementation of the interface, these methods must be declared:

```
long days();
long months();
void months(long new_months);
```

Note Currently, attributes do not do not display with a component's methods in EAServer Manager. Use the IDL editor to view attribute definitions.

Datatypes for parameters and return values

To define parameter and return value datatypes, you can use EAServer's predefined IDL datatypes or your own user-defined IDL types. In addition, EAServer extends IDL to allow the use of Java class names. The sections below describe each option in detail.

- Predefined IDL datatypes

- User-defined IDL datatypes
- Java class names used as IDL datatypes

Predefined IDL datatypes EAServer ships with predefined datatypes for use in declaring parameter and return value datatypes. Predefined datatypes include all CORBA base types (except for the CORBA::Any type) and equivalents for database result sets and other commonly used database column types such as date, time, and timestamp.

EAServer Manager's Method Properties dialog box displays the predefined datatypes in the drop-down lists for Parameter and Return types. "Predefined datatypes" on page 79 lists EAServer's predefined IDL datatypes, the equivalent display names, and a description of each.

For descriptions of the datatypes defined in the BCD, MJD, or TabularResults modules, see the documentation in the *html/ir* subdirectory of your EAServer installation. (Or, load the main EAServer HTML page in your Web browser, and click the Interface Repository link). If you use types from these modules, add an include directive for the appropriate module at the top of the module that defines your interface. For example:

```
#include <TabularResults.idl>
```

Internally, *TabularResults.idl* includes both *BCD.idl* and *MJD.idl*. You need not include *BCD.idl* and *MJD.idl* explicitly if you have already included *TabularResults.idl*.

User-defined IDL datatypes In addition to EAServer's predefined datatypes, you can define your own datatypes in IDL and use them to declare return types and parameters.

All IDL type definitions are allowed, with these exceptions:

- Arrays are not yet supported. You can use sequences instead.
- The CORBA::Any type is not supported.
- constant declarations are supported.

User-defined types must exist in the EAServer IDL repository before you can use them in interface declarations. For information on defining datatypes, see Chapter 3, "OMG IDL Syntax and Semantics," in the CORBA 2.3 specification.

In some cases, you must use the full scope name. In a parameter list, use a type's full scope name if any of the following is true:

- The type is declared in another interface.

- The type is declared in another module.
- The type has the same local-scope name as a type declared in the interface or module that contains the operation.

For example, consider the IDL:

```
module MyMod {
    typedef string MyType;
    interface MyIntf {
        typedef double MyOtherType;
        ....
    };
};
```

With these declarations, MyMod::MyType is the full scope name for MyType and MyMod::MyIntf::MyOtherType is the full scope name for MyOtherType.

Java class names used as IDL datatypes EAServer's IDL compiler extends IDL to allow Java class names as parameter and return types for methods. This feature provides functionality that is similar to the proposed Objects by Value CORBA extension (OMG TC Document orbos/98-01-18, *Objects By Value*). Specifically, you can pass a copy of an object rather than passing an interface pointer that refers back to the original object.

You can specify any Java class name for a method input parameter or return type as long as:

- The class containing the type name is in the CLASSPATH environment variable both when the interface is defined and when the server is run.
- At run time, you specify a class instance that is serializable. That is, a class must implement the java.io.Serializable interface or inherit from another class that does so, and an interface must extend the java.io.Serializable interface. If the instance is not serializable, the call fails with a CORBA::MARSHALL exception.

Note the following restrictions for methods that are defined using Java datatypes rather than IDL or predefined EAServer Manager types:

- Only Java components can implement the method and only Java clients can invoke the method.
- Only in parameters and return values can be declared with Java class names.

- Java datatypes are not marshaled as efficiently as an equivalent IDL datatype. *Marshaling* is the process of reading and writing parameters and return values from the network. More bytes are required to marshal values defined with a Java datatype than to marshal an equivalent IDL type. Consequently, invocations of a method defined with Java datatypes are slower than invocations of an equivalent method defined with IDL datatypes.
- IDL that contains Java class names may not be portable to other CORBA client ORB implementations unless they offer this extension to standard CORBA IDL.

User-defined exceptions

Exceptions can be declared in a module or interface. Exceptions are declared as follows:

```
exception name {
    ... memberList ...
};
```

where *name* is the name of the exception and *memberList* is an optional list of member field declarations. This list has the form:

```
exception MyException {
    type1 member1;
    type2 member2;
    ...
};
```

Where *type1*, *type2*, and so forth are IDL type names (other than CORBA::Any) and *member1*, *member2*, and so forth are the names of the member fields.

Once you have defined an exception, you can use it in the raises clause when defining operations for an interface, as described in “Operation declarations” on page 93.

Note User-defined exceptions are not supported by ActiveX components and clients.

Adding IDL documentation comments

EAServer Manager creates HTML documentation files for each IDL module in the *html/ir* subdirectory.

At a minimum, the HTML file lists the datatypes and interfaces defined in the module. You can embed additional documentation text for a datatype, interface, or method in a C-style comment placed immediately above the declaration. EAServer ignores C++-style line-end comments when generating HTML documentation. That is, text within comments that use double slashes, //, to delineate the comment text is ignored.

Within the C-style comment, add text describing the item to the comment, as in the example below. If desired, you can use HTML codes to format the text. But do not use heading tags such as <H1>, <H2>, and so forth, because they conflict with tags that are already used to structure the sections of the generated output.

The IDL fragment below contains an example of a documentation comment:

```
/**
 ** Example method to demonstrate user-defined
 ** exceptions.
 ** <P>Pass <I>yes_no</I> as <code>true</code>
 ** if you want an exception thrown.
 ** <P>Returns input value of <I>yes_no</I>
 ** parameter.
 */
boolean throwException
(
in boolean yes_no
)
raises
(
myException
);
```

You need not use the spacing conventions illustrated in this example. EAServer Manager treats any C-style comment as an IDL documentation comment. However, when you save in the IDL Editor window, EAServer Manager reformats all C-style comments to match this example's spacing convention.

Stub generation directives in IDL comments

You can embed directives in IDL comments to affect the Java stubs generated for a module or interface. See "Interface stub generation directives" on page 92 for more information.

Refreshing the HTML documentation

HTML documentation is not generated automatically. You must use EAServer Manager to create or update documentation for new or changed IDL modules. In EAServer Manager, highlight a component, package, server, or module, then select File | Generate HTML. The top level *index.html* file is updated only when you generate HTML for a server.

To update documentation for all IDL modules in the EAServer interface repository, generate HTML for any server. To selectively update documentation for interfaces used by components, generate HTML for a component or package; EAServer Manager will generate documentation for all IDL modules used in the component or components within the package. To update only the documentation for a single module, highlight that module then select File | Generate HTML.

Viewing HTML documentation for IDL modules

EAServer creates HTML documentation for all imported IDL modules in the style of Sun's javadoc tool. At a minimum, this documentation lists the datatypes and interfaces defined in the module, including structure fields, array lengths, parameter names and datatypes, exceptions thrown by methods, and so forth. When editing IDL, you can also create specially-formatted comments that provide descriptions of entities declared in the IDL file, as described in "Adding IDL documentation comments" on page 98.

Module documentation can be viewed in a Web browser by connecting to your server with this URL:

```
http://yourhost:yourport/ir/
```

where *yourhost* is the host name and *yourport* is the HTTP port number.

Importing existing IDL modules

You can import interfaces defined in CORBA IDL into the EAServer interface repository. There are two ways to import a module:

- Organize the modules so that one module is declared per file, and each file has the same name as the module it declares.

For example, module MyModule should be declared in the file *MyModule.idl*. Copy the files to the EAServer *Repository* subdirectory and restart the server. If the file contains no syntax errors, its declarations will be added to the EAServer Manager IDL folder. If the file does contain syntax errors, the server will log the errors during start-up and the module's declarations will not be added to the IDL repository.

- Create a new module in EAServer Manager, as described in "Creating and editing IDL modules, interfaces, and types" on page 86.

  While the new module declaration is displayed in EAServer's IDL editor, open the module to be imported in another text editor. Copy and paste the text of the module to be imported into the EAServer IDL editor.

To deploy IDL types and interfaces that are not declared within a module, place the IDL file that defines them in the EAServer *Repository* subdirectory and restart EAServer if it is running.

You can repeat the procedures above to redefine existing IDL definitions.

PART 2

Enterprise JavaBeans

This part explains how to create and/or deploy Enterprise JavaBeans components and clients in EAServer.

CHAPTER 6

Enterprise JavaBeans Overview

EJB 2.0 support is new in EAServer version 4.0. EJB 1.1 and 1.0 components are supported for backward compatibility with earlier EAServer versions and other EJB servers.

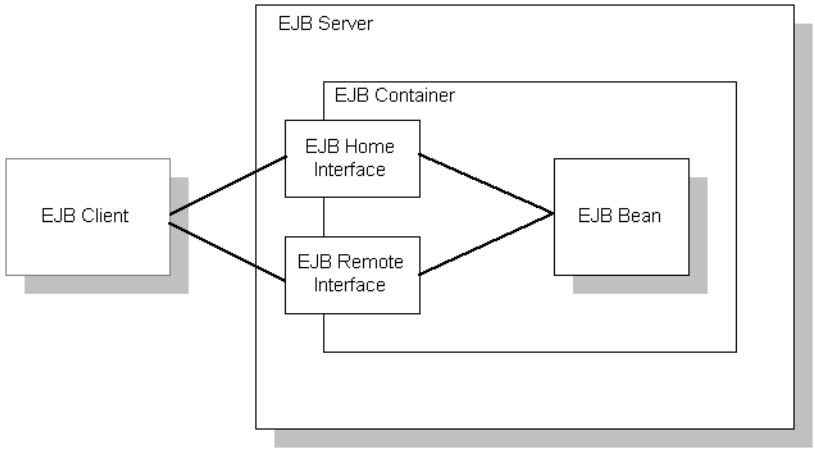
For more details on the EJB architecture, see the EJB specifications from Sun Microsystems at http://java.sun.com/products/ejb/.

About Enterprise JavaBeans components

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

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

The EJB architecture looks like this:



EJB server The EJB server contains the EJB container, which provides the services required by the EJB component. EAServer is an EJB server.

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.

Beginning in EJB version 2.0, clients can also execute EJB components using **local interfaces** if the client and component execute in the same virtual machine. Using the local interface can improve performance.

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 EAServer, 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 EAServer 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 unless they are run in a cluster that has persistent storage enabled for the component.
- There is an affinity between each instance and one client from the time the client creates the instance until it is destroyed by the client or by the server in response to an expired instance timeout limit.

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

Stateless session beans

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

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

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

Entity beans

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

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

EJB transaction attribute values

Each EJB component has a transaction attribute that determines how instances of the component participate in transactions. In EAServer, 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 with 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 6-1 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 6-1: Transaction attribute values

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

Attribute	Description
Bean Managed	(For EJB session beans only.) The EJB component can explicitly begin, commit, and roll back new, independent transactions by using the javax.transaction.UserTransaction interface. Transactions begun by the component execute independently of the client's transaction. If the component has not begun a transaction, the component's database work is performed independently of any EAServer transaction.
Default to component	(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.

EJB container services

The EJB container provides services to EJB components. The services include transaction, naming, 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 27, "Creating Entity Components," for more information.

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

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

See Chapter 5, "Naming Services," in the *EAServer System Administration Guide* for more information on JNDI.

EAServer EJB support

EAServer can host Enterprise JavaBeans (EJB) components developed according to version 2.0, 1.1, or 1.0 of the Enterprise JavaBeans specification. EAServer supports session beans and entity beans with bean-managed persistence or container-managed persistence. EAServer uses CORBA 2.3 as the basis for the EJB component support, allowing interoperability with other client and component models and with CORBA-2.3-compliant ORBs from other vendors.

Running EJB components in EAServer

EAServer can host Enterprise JavaBeans (EJB) components developed according to version 2.0, 1.1, or 1.0 of the Enterprise JavaBeans specification. EAServer supports session beans and entity beans with bean-managed persistence or container-managed persistence. EAServer uses CORBA 2.3 as the basis for the EJB component support, allowing interoperability with other client and component models and with ORBs from other vendors that are compliant with CORBA 2.3.

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

- Use Jakarta Ant to develop an EJB-JAR file and deploy it to EAServer with jagant. For more information on using jagant, see Chapter 12, "Using jagtool and jagant," in the *EAServer System Administration Guide*.

- If using the Borland JBuilder IDE, use the EAServer plugin to deploy the components to EAServer.
- If using another IDE, create an EJB-JAR file and use EAServer Manager or jagtool to import an EJB-JAR file that contains the classes and deployment descriptors for one or more EJB components. EAServer Manager 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. EAServer Manager 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 EAServer Manager Component Properties dialog box.
- Define an EJB component from scratch in EAServer Manager, using EAServer Manager's IDL generation tools to define the home interface, remote interface, and primary key type. EAServer Manager generates Java classes for the home and remote interfaces and primary key class, as well as a template for the implementation class.

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

EJB clients connecting to EAServer

EAServer also supports the Enterprise JavaBeans client model by generating EJB proxies and providing an EJB-compliant implementation of the JNDI NamingContext class. You can generate EJB-style proxies for any IDL interface (not just those associated with EJB components), 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.

For more information

For information about	See this chapter or section
Creating, importing, and exporting EJB components.	Chapter 7, “Creating Enterprise JavaBeans Components”
Creating EJB clients, generating EJB stubs, instantiating home and remote interface proxies, managing transactions, and serializing and deserializing bean proxies.	Chapter 8, “Creating Enterprise JavaBeans Clients”
Configuring container-managed persistence for entity beans and passivation of stateful session beans	“Configuring automatic or EJB CMP persistence” on page 507
Invoking non-EJB components from EJB clients and invoking EJB components from non-EJB clients, and using EAServer with EJB 2.0 containers from other vendors.	Chapter 9, “EAServer EJB Interoperability”

EJB 2.0 differences from 1.1

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

Message-driven beans

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

For information on JMS, see Chapter 31, “Using the Message Service.” For information on creating MDB components, see “Message-driven beans” on page 577.

Home interface methods

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

Create method syntax

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

Home interface business methods

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

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

Then the implementation class must contain:

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

Local interfaces

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

Using local interfaces can improve performance for calls to components hosted in the same server, but in coding you must be aware of the restrictions listed in "Calling local interface methods" on page 150.

CMP enhancements

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

- The deployment descriptor more fully describes the persistent fields in the bean and the required database queries, making for less work after deploying an EJB JAR file that contains CMP entity beans.

- CMP entity beans in the same EJB JAR (which maps to an EAServer package) can have container-managed relationships. For example, an *Order* bean may have an *items* field that consists of a collection of *Inventory* bean instances representing the items being purchased. Or, an *Employee* bean may be related to itself, with *manager* and *employees* fields that contain Employee instances.

For more information on EAServer CMP support, see Chapter 27, "Creating Entity Components."

EJB 2.0 interoperability

EAServer 4.0 complies with the interoperability requirements in the EJB 2.0 specification to allow interoperability with other EJB 2.0 servers. EAServer continues to support CORBA-2.2 based interoperability, for interacting with other CORBA-based application servers and to allow interoperability between EJB components hosted by EAServer and EAServer components of other types. For more information, see Chapter 9, "EAServer EJB Interoperability."

EJB 1.1 differences from EJB 1.0

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

For more details, see the EJB 1.1 and 1.0 specifications from Sun Microsystems at http://java.sun.com/products/ejb/.

Component differences

JNDI names in deployment descriptors

The EJB 1.1 JAR file format does not specify JNDI names for deployed EJB components. Consequently, EJB 1.1 components imported into EAServer use the default JNDI name of package/component, where package is the EAServer package name and matches the display-name attribute of the EJB deployment descriptor, and component is the EAServer component name and matches the bean's ejb-name element in the deployment descriptor.

If you have an existing client application that invoke the component, you may have to change the component's JNDI name or the name used in client application.

For intercomponent calls from EJB 1.1 components, you can use the EJB References property to alias the JNDI name used in the bean to an installed component with a different JNDI name.

Environment properties

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

Environment properties used within a bean must be cataloged in the bean's deployment descriptor. For EJB 1.1 components installed in EAServer, you configure environment properties on the Environment tab in the Component Properties dialog box. See "Configuring environment properties" on page 133

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

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

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

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

```
// Use these properties to customize the business logic
if (numberOfExemptions > max.intValue() ||
   (numberOfExemptions < min.intValue())
    throw new InvalidNumberOfExemptionsException();
```

.

EJB and resource references

EJB 1.1 allows components to use logical names to access database connections, JavaMail sessions, and the home interfaces of other components. These names must be catalogued in the bean's deployment descriptor. For components installed in EAServer, you configure references on the Resource References tab in the Component Properties dialog box. See these sections for more information:

- "Configuring resource references" on page 133
- "Configuring EJB references" on page 132

Security access-control changes

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

In EAServer Manager, you can configure role references for your component in the Component Properties dialog box. Role references allow you to map names used in isCallerInRole(java.lang.String) calls to role names that exist on the server. Role references allow your component to be deployed on servers that do not have the same security configuration.

Declarative access control for EJB 1.1 components uses method-level settings.

Role Membership folder does not apply to EJB 1.1 or 2.0 components
The Role Membership folder for packages and components in EAServer Manager does not apply to EJB 1.1 or 2.0 components. There are two ways to control which clients can call EJB component methods:

- You can use the Permissions tab in the Method Properties dialog box to configure access declaratively for each method. To limit access to all of a component's remote interface methods, configure the permissions for the home-interface create and finder methods.
- You can configure additional access control programmatically by calling the getCallerPrincipal and isCallerInRole methods in the component implementation. Programmatic access control enhances declarative access control, but does not replace it.

❖ **Configuring method permissions**

Method permissions allow you to restrict access without writing code. Configure method permissions as follows:

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

2 Verify that J2EE roles are mapped to EAServer roles in the properties of the package where the component is installed; check the Role Mappings tab in the Package Properties window. You must map a J2EE role name for each role to be used in method permissions.

3 For each method that requires limited access, display the Method Properties dialog and highlight the Permissions tab. A check box displays for each mapped J2EE role in the package that contains the component. Select the check box by each role that can call the method.

❖ **Configuring role references**

Role references are required if you call the isCallerInRole Java method to restrict access. Each reference maps a string used in isCallerInRole calls to a J2EE role that is configured in the package Role Mappings. To configure role references:

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

2 Verify that J2EE roles are mapped to EAServer roles in the properties of the package where the component is installed; check the Role Mappings tab in the Package Properties window. You must map a J2EE role name for each role to be used in role references.

3 For each component that calls the isCallerInRole method, display the Component Properties dialog and highlight the Role Refs tab. Add or modify roles as follows:

- To add a role, click Add and edit the new entry as described below.
- To modify a role, edit the Reference Name (used in isCallerInRole calls), and choose the mapped J2EE role (configured in the properties of the package where the component is installed).

Transaction isolation level

In accordance with the EJB 1.1 specification, you cannot set the transaction isolation level declaratively for EJB 1.1 components. The simplest way to configure the transaction isolation level is to configure the defaults on the databases that you access from your EJB components. If this is not possible, you must set the isolation level programmatically in the component implementation.

Client model differences

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

- **Finder method return types** Finder methods in EJB 1.1 clients can return java.util.Collection or java.util.Enumeration. Finder methods in EJB 1.0 must return java.util.Enumeration. The use of java.util.Collection is recommended for new development.

Configuring Java finder method return types

You can specify the return type for finder methods that return multiple keys with an IDL directive, as described in “Specifying Java package mappings for IDL modules” on page 87. If you import interfaces from an EJB-JAR file or EJB class files, these directives are created automatically.

When generating EJB stubs, choose the Java version to specify the default return type for IDL finder methods that lack an EJB package directive. See “Generating EJB stubs” on page 138 for more information.

- **Home interface serialization** You can call the Home.getHandle method to serialize a home interface proxy in an EJB 1.1 client.
- **EJBMetaData enhancements** The EJBMetaData interface, used by development tools to dynamically inspect EJB components, provides an isStatelessSession method that returns true if the component is a stateless session bean.

CHAPTER 7

Creating Enterprise JavaBeans Components

This chapter describes how to install Enterprise JavaBeans as components in EAServer Manager. You can use any development tool to develop EJB components, including EAServer Manager and any JDK 1.2 or later Java compiler. EAServer also supports the standard EJB-JAR import and export format for deployment of packages containing related EJB components.

Defining an EJB component

There are three ways to define EJB components in EAServer:

- **Importing an EJB-JAR file** An EJB-JAR file contains the implementation classes, interface classes, and deployment descriptor for one or more beans archived in a standard format. Chapter 9, "Importing and Exporting Application Components," in the *EAServer System Administration Guide* describes how to import EJB-JAR files.
- **Importing class files** EAServer Manager can import the method information for the home, remote, and local interfaces from Java class files. Use this method if:
  - You have created a bean's interfaces and implementation class, but have not created the deployment descriptor that is required to create an EJB-JAR file. You will need to manually configure properties that would otherwise be read from the deployment descriptor.

- You have created nothing, but prefer editing Java in your code editor to editing IDL in EAServer Manager.

- **Defining the component from scratch** You can define the component and it's interfaces in EAServer Manager, using the IDL editing facilities in EAServer Manager to define the home, remote, and (optional) remote interfaces.

❖ **Importing EJB class files**

1 If necessary, create class files for the home, remote, and (optionally) local interfaces, following the EJB standards for these interfaces.

2 Specify the package to install the component in as follows:

   a Double-click the Packages folder to expand it.

   b Highlight the package to which the component will be added.

3 Choose File | New Component from the menu.

4 In the Component Wizard, select Import from EJB Class File, and click Next.

5 Verify that the displayed importer CLASSPATH contains the JAR files and directories required to instantiate the bean's classes, specifically:

   - Verify that the code base under which the class files are deployed is included.
   - If the classes are in a JAR file, verify that the full path to the JAR file is included.
   - If the class definitions require other JAR files or directories not listed, list them as well.

   If necessary, use the Add, Modify, Delete, Move Up, and Move Down buttons in the Component Wizard to modify the CLASSPATH. The displayed CLASSPATH affects only this importer session, not the EAServer process.

6 Enter the component name and EJB class and interface names as follows:

   - **Component name** The name of the component to be created in EAServer Manager, for example, FinanceBean.
   - **Component type** Choose one of the following to match your implementation:

Type	Description
JaguarEJB::StatelessSessionBean	A stateless session bean

Type	Description
JaguarEJB::StatefulSessionBean	A stateful session bean
JaguarEJB::EntityBean	An entity bean with bean-managed persistence.

- **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.
- **Specify remote interface** If the Bean has remote interfaces, select this option and configure the following:
  - *Home interface class* – The full path to the Java class file that contains the Bean's home interface.
  - *Remote interface class* – The full path to the Java class file that contains the Bean's remote interface.
- **Specify local interface** If the Bean has local interfaces, select this option and configure the following:
  - *Local home interface class* – The full path to the Java class file that contains the Bean's local home interface.
  - *Local interface class* – The full path to the Java class file that contains the Bean's local interface.

7 EAServer Manager 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 generating skeletons and running the bean.

8 Generate stubs and skeletons for the component as follows:

a Highlight the component icon.

b Choose File | Generate Stub/Skeleton.

c Follow the wizard pages to generate skeletons.

d Click Generate.

Stubs generated automatically
When you generate skeletons, EAServer Manager generates stubs under the same code base. You do not need to enable the Generate Stubs options.

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

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

1 Select the EAServer Manager package that will contain the component.

2 Select File | New Component.

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

4 Enter a name for the component and click Finish.

5 The Component Properties dialog box displays. Make the following changes on the General tab:

a Set the 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
EJB - Message Driven Bean	A message-driven bean

b In the EJB Version field, select 2.0. (You can select 1.1. or 1.0, but EJB 2.0 is recommended for new development.)

c In the Bean Class 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.

d Enter a value for the JNDI name field. This field specifies the name by which client applications look up the home interface. The full name consists of the server's initial naming context followed by a slash (/) and the bean's JNDI name.

6 If you are creating an entity bean, specify the primary key as follows:

a Define the primary key type as one of the "Defining the primary key type" on page 126.

b Display the Component Properties dialog box for the component, click on the Persistence tab, and type the name of the IDL primary key type into the Primary Key field. If using mean managed persistence, the Persistence must be set to Component Class (the default). If using container managed persistence (CMP), configure the additional settings described in "Configuring automatic or EJB CMP persistence" on page 507.

7 Click OK to close the Component Properties dialog box.

8 If methods in your Java remote interface throw exceptions other than java.rmi.RemoteException, define equivalent IDL exceptions now. See Chapter 5, "Defining Component Interfaces," for more information.

9 Define home and remote interfaces. EAServer Manager has created default home and remote interfaces named *package*::*component*Home and *package*::*component*, respectively, where *package* is the EAServer Manager package name, and *component* is the component name.

a To change the home or remote interface, follow the steps in "Changing the EJB remote or home interface" on page 75.

b Edit the home interface methods, following the design patterns described in "Defining home interface methods" on page 127.

c Edit the remote interface methods. See "Defining remote interface methods" on page 129. If portability to other EJB servers is required, use only in parameters in remote interface methods.

An EJB 2.0 component may have local interfaces, but no remote interfaces. To remove the remote interfaces generated by EAServer Manager, highlight the Interfaces folder under the component icon, then choose File | Remove Remote Interfaces.

10 Define local interfaces. EAServer Manager has created default local home and local interfaces named *package*::*component*LocalHome and *package*::*component*Local, respectively, where *package* is the EAServer Manager package name, and *component* is the component name.

a If you wish to keep the local interfaces, define methods for them as described in "Defining local interfaces" on page 130.

b If you do not need local interfaces, highlight the Interfaces folder under the component icon, then choose File | Remove Local Interfaces.

11 Generate stubs and skeletons for the component as follows:

a Highlight the component icon.

b Choose File | Generate Stub/Skeleton.

c Follow the wizard pages to generate skeletons.

d Click Generate.

Stubs generated automatically
When you generate skeletons for your component, stub source is also generated under the same code base. You do not need to select the Generate Stubs option.

12 EAServer Manager 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. EAServer Manager also generates Java equivalents for the home and remote interfaces, and for an entity bean, the primary key type.

If you are creating a stateful session bean with synchronization methods, add implements SessionSynchronization to the class declaration in the implementation template, and add code to implement the methods in the javax.ejb.SessionSynchronization interface.

13 Compile the component source files, and make sure they are correctly deployed. See “Deploying the component classes” on page 134.

14 If you are testing the component with a Java applet, generate and compile stubs using the *html/classes* subdirectory as the Java code base.

Defining the primary key type

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. See “Creating IDL types, exceptions, and interfaces” on page 88 for more information.

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.

Interoperability and key types
Define your entity bean's primary key as an IDL structure or string if other types of clients besides Java will use the bean.

Defining home interface methods

You can add methods to a home interface using the techniques described in Chapter 5, "Defining Component Interfaces." 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. In Java, create methods must have names that begin with create, as in createAccount. (If defining an EJB 1.1 or 1.0 bean, create is the only valid name.)

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 in IDL, append two underscores and a unique suffix. (This is the standard Java to IDL mapping for overloaded method names. When generating stubs for C++ and Java, EAServer 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__overload-suffix
(
  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:

- **Single-object finder methods** 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.

- **Multi-object finder methods** 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.

  By default, the Java form of multi-object finder methods returns java.util.Collection. For compatibility with older EJB code, you can specify that generated stub methods should return java.util.Enumeration. To do so, add an IDL doc comment before the IDL method definition with this form:

```
/**
** <!-- java.util.Enumeration -->
**/
::MyModule::MyRemoteList findByName(in string name);
```

Sequence types are automatically generated
EAServer Manager 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 *component*List and a sequence of primary keys is *component*Keys, where *component* is the component name.

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

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

Then the implementation class must contain:

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

Home interface business methods cannot be used in EJB 1.1 or 1.0 beans.

Defining remote interface methods

The IDL for your 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 graphically or using the IDL editor window. The procedure is the same as for any other IDL interface. See Chapter 5, “Defining Component Interfaces,” for more information.

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

Defining local interfaces

The EJB 2.0 architecture introduces local interfaces for calls to an EJB component from within the same Java Virtual Machine. In EAServer, you can use local interfaces for intercomponent calls, and for component invocations made from servlets and JSPs hosted in the same server as the component.

Using local interfaces can improve performance, but in coding you must be aware that:

- Parameters are passed by reference rather than by copy, so object instances passed through a local invocation can be shared by the client and component. If the component modifies the object, the client sees the changes.
- Local interfaces are not location transparent. The called component must be hosted in the same server process as the calling component, and both components must be configured to use the same custom class loader. See “Calling local interface methods” on page 150 for more information.

Defining local interfaces in Java The Java local home interface must extend javax.ejb.EJBLocalHome. Other than the base interface, the requirements are the same as for defining the home interface.

The Java local interface must extend javax.ejb.EJBLocalObject. Other than the base interface, the requirements are the same as for defining the local interface.

Defining local interfaces in IDL In IDL, local home interfaces can contain create and finder methods. The local home for an entity bean can also contain business methods. The IDL syntax is the same as for remote home interfaces, namely:

- IDL create methods must return the local interface type and raise CtsComponents::CreateException.

- The IDL findByPrimary key method must return the local interface type, accept the primary key type as the sole parameter, and raise CtsComponents::FinderException.
- Any additional IDL finder methods must return a sequence of the primary key type and raise CtsComponents::FinderException.

The local interface can be defined in IDL with the same restrictions as for the IDL remote interface.

Configuring the component properties

After you have defined the component and its methods, you may need to configure the properties described here.

❖ **Configuring EJB component properties**

1. If you are defining a stateful session bean, optionally switch to the Resources tab and enter a time limit in the Instance Timeout 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.
2. If creating an entity bean with container-managed persistence, configure the persistence settings as described in “Configuring automatic or EJB CMP persistence” on page 507.
3. 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. See “Component properties: Transactions” on page 58 for more information.
4. If defining a version 2.0 or 1.1 EJB that calls other components, configure the properties described in “Configuring EJB references” on page 132.
5. If defining a version 2.0 or 1.1 EJB that uses database connections or JavaMail sessions, configure the properties described in “Configuring resource references” on page 133.
6. If defining a version 2.0 or 1.1 EJB, configure method security constraints as described in “Configuring role references and method permissions” on page 133.

7 If defining a version 2.0 EJB that calls other components, optionally configure the Run-As Identity properties to specify the identity used in intercomponent calls. See “Component properties: Run-As Identity” on page 67 for more information.

8 If defining a version 2.0 EJB that uses JMS, configure the properties described in “Component properties: Resource Environment Refs” on page 64.

9 If defining a version 1.0 EJB that calls other components, configure the properties described in “Component properties: Run-As Mode” on page 68.

10 Optionally configure environment properties as described in “Configuring environment properties” on page 133.

Configuring EJB references

Your EJB can use EJB references to instantiate proxies for other EJBs. You do not need to create references in order to invoke other EJBs from your code. However, doing so ensures that EJB references will be cataloged in the deployment descriptor if you export the EJB. There are two types of references:

- Local references, for calls to EJB components hosted in the same server using the local home and local interfaces. To add or edit local references, follow the instructions in “Adding an EJB local reference” on page 384, or “Editing an EJB local reference” on page 384, respectively.
- Remote references, for calls to components of any type using the component’s home and remote interfaces. To add or edit remote references, follow the instructions in “Adding an EJB reference” on page 383, or “Editing an EJB reference” on page 383, respectively.

Stubs used for EJB references must be in the custom class list
You must list stubs used for intercomponent calls in the custom class list for your component, as described in “Custom class lists for Java and EJB components” on page 556.

Configuring resource references

Resource references are used to obtain connector and database connections, and to access JMS connection factories, JavaMail sessions, and URL links.

❖ **Adding or editing a resource reference**

1 Open the Component Properties dialog box.

2 Follow the instructions in "Adding a resource reference" on page 385, or "Editing a resource reference" on page 385.

Configuring role references and method permissions

To configure authorized access to an EJB 2.0 or 1.1 component, you must configure method permissions settings or call the isCallerInRole Java method to restrict access. See Chapter 2, "Securing Component Access," in the *EAServer Security Administration and Programming* guide for more information.

Configuring environment properties

Environment properties allow you to specify read-only data for use by an EJB. For example, you might use environment properties to tune the size of a data cache used in your implementation, or to specify the name of a log file. Use environment properties for any constant value that might change when the EJB is deployed to another server.

When coding your EJB, use JNDI to retrieve environment properties, using the prefix `java:comp/env` in JNDI lookups.

When you export your EJB, the deployment descriptor catalogs the environment properties used by your servlets and JSPs, as well as each property's Java datatype and default value. When the EJB is imported to another server, the deployer can override the default value for each environment property.

Environment properties for EJB 1.0 components
An EJB 1.0 component can only have environment properties with datatype String, and these properties must be configured in the Advanced window. Any property name that does not begin with `com.sybase.jaguar.component` is considered an environment property. In source code, use the EJBContext.getEnvironment method to retrieve property values. You cannot use the JNDI InitialContext.lookup method to retrieve these values.

❖ **Adding or editing an EJB environment property**

1 Open the Component Properties dialog box.

2 Follow the instructions in "Adding an environment property" on page 388, or "Editing an environment property" on page 388.

Deploying the component classes

If you are creating components from scratch in EAServer Manager, you must follow the steps in this section to deploy the component class and other classes that it depends on. If you deploy from JBuilder with the EAServer plugin, the plugin 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 Chapter 9, "Importing and Exporting Application Components," in the *EAServer System Administration Guide*. If you import an EJB-JAR file that calls EAServer 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.

EAServer 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 the server. Repeat the steps below to deploy new versions of your implementation.

In a production environment, you may wish to disable refresh to improve performance. See "Disabling refresh" on page 136 for details.

❖ **Deploying EJB component classes**

1 Deploy the component class files, stub and skeleton files, and other classes required by the implementation to EAServer. 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 *java/classes* package subdirectories. If deploying a JAR file, place it in the *java/classes* subdirectory.

The preferred code base is java/classes
For security reasons, it is preferable to deploy Java components to the *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 EAServer Manager 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 "Custom class lists for Java and EJB components" on page 556.

3 Use EAServer Manager to refresh the component by highlighting its icon and choosing File | Refresh. You can also refresh the component by refreshing the package, application, or server where it is installed.

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.

For more information, see "Troubleshooting class loader configuration issues" on page 562.

❖ **Disabling refresh**

In a production server, you may wish to disable refresh for Java components to decrease memory use and increase performance. When refresh is enabled, duplicate copies of common Java classes can be loaded for components. When refresh is disabled, you must restart the server in order for it to load a new version of your component class. You can also reduce duplicate in-memory classes by configuring the custom class list at the package, server, or application level as described in "Custom class lists for packages, applications, or servers" on page 560.

If your component calls another Java or EJB component, both must have refresh enabled or both must have refresh disabled.

To disable refresh:

1 Make sure the code bases for all classes used by your component are in the server's CLASSPATH environment variable. JAR files referenced in the custom class list must be added to the server CLASSPATH setting, or expanded into the *java/classes* EAServer subdirectory.

2 Display the Component Properties dialog box, and click on the Advanced tab.

3 Set the `com.sybase.jaguar.component.refresh` to false (the default is true).

4 Restart the server for the changes to take effect.

CHAPTER 8

Creating Enterprise JavaBeans Clients

This chapter describes how to implement EJB clients using the Sybase EJB client runtime. 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.

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 138.
2	Add code to create the initial naming context and instantiate the home interface proxies.	See “Instantiating home interface proxies” on page 140.
3	Add code to instantiate remote or local interface proxies.	See “Instantiating remote or local interface proxies” on page 148.

Step	Action	For more information
4	Add code to call remote or local interface methods.	See “Calling remote interface methods” on page 150 or “Calling local interface methods” on page 150.
5	Optionally add code to control transactions and serialize and deserialize instances.	See: • “Managing transactions” on page 151 • “Serializing and deserializing bean proxies” on page 152
6	Deploy the client application or applet.	See “Runtime requirements” on page 153.

Generating EJB stubs

Stub classes act as proxies for an instance of the EAServer component. You can generate EJB stubs for components that are implemented in any of EAServer’s supported component models. One stub interface is generated for each IDL interface that the component implements.

Before generating stubs

If you are generating stubs for a component that is not an EJB component, make sure the component has a home interface defined. See “Invoking non-EJB components from EJB clients” on page 160 for more information.

If you are generating stubs for multiple client models, stubs for each model must be generated to a different code base or Java package.

❖ **Generating EJB stubs**

1 Highlight a component, package, or module as follows:

- Highlight a component to generate stubs for all interfaces and types required by a component,
- Highlight a package to generate all stubs needed by components in the package, or
- Highlight a module in the IDL folder to generate stubs for IDL interfaces and types defined within that module.

Specifying a different Java package
If stub classes must be generated using a Java package other than the default, generate stubs by highlighting the IDL module that contains the interfaces and datatypes of interest. When generating stubs for a module, you can override the default package name.

2 Select File | Generate Stub/Skeleton. The Generate Stubs & Skeletons Wizard displays. Follow the instructions on each page to generate EJB stubs. See the online help for descriptions of any input fields that you do not understand.

For each IDL interface that is assigned to a component, EAServer Manager 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, EAServer Manager creates the source files listed in the following table:

Table 8-1: Java stub source files for example interface calc

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

EAServer Manager 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.

If you did not elect to compile the stubs in EAServer Manager, compile the stub classes. Make sure that the CLASSPATH setting contains the code base directory and the following JAR files in the EAServer installation directory:

- *java/lib/easserver.jar*
- *java/lib/easclient.jar*
- *java/lib/easj2ee.jar*

Instantiating home interface proxies

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

Obtaining an initial naming context

The core JNDI interface used by client applications is javax.naming.Context, which represents the initial naming context used to resolve names to bean proxies. To obtain an initial naming context, initialize a java.util.Properties instance and set the properties listed in Table 8-2. 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. To specify properties used by a Java applet, use parameters in the HTML Applet tag that loads the applet.

Sybase InitialContext properties

The Sybase InitialContext implementation recognizes the properties in the following table. You can create multiple contexts with different properties. For example, you might create one context for proxies that connect with plain IIOP and another for proxies that connect using SSL.

Table 8-2: Sybase EJB InitialContext Properties

Property name	Description
java.naming.factory.initial	Specifies the fully qualified Java class name of the class that returns javax.naming.InitialContext instances that interact with the naming provider. Use `com.sybase.ejb.InitialContextFactory` for EAServer EJB clients. **When using corbaname URLs** The EJB client runtime supports corbaname URLs to support EJB 2.0 interoperability features, as described in "Interoperable naming URLs for EJB clients" on page 157. When using corbaname URLs, you must specify the username and password using the JAAS API as described in Chapter 10, "Using the JAAS API," in the *EAServer Security Administration and Programming Guide*. The context principal and username properties do not apply to contexts that use a corbaname URL.
java.naming.provider.url	Specifies the URL to connect to the EAServer name server. Set the value to a URL with the following format: `iiop://hostname:iiop-port/initial-context` where: • *hostname* is the host machine name for the server that serves as the name server for your application. If omitted, the default is `localhost`. • *iiop-port* is the IIOP port number for the server. • *initial-context* is the initial naming context. This can be used to set a default prefix for name resolution. For example, if you specify *USA/Sybase/*, all names that you resolve with the context are assumed to be relative to this location in the name hierarchy. When specifying the initial context, the trailing slash is optional; it is added automatically if you do not specify an initial context that ends with a slash. If you do not set this property, the default is `iiop://localhost:9000/`.

Property name	Description
java.naming.security.principal	Specifies the user name for the EAServer session. Required if user name/password authentication is enabled for your server. **When using corbaname URLs** The EJB client runtime supports corbaname URLs to support EJB 2.0 interoperability features, as described in "Interoperable naming URLs for EJB clients" on page 157. When using corbaname URLs, you must specify the username and password using the JAAS API as described in Chapter 10, "Using the JAAS API," in the *EAServer Security Administration and Programming Guide*. The context principal and username properties do not apply to contexts that use a corbaname URL.
java.naming.security.credentials	Specifies the password for the EAServer session. Required if user name/password authentication is enabled for your server.
com.sybase.ejb.ConnectionTimeout	For applications that run in a cluster, sets a time limit to receive a server response before the connection fails over to try another server in the cluster. Setting this property ensures that failover happens without an unreasonable delay. Specify the timeout period in seconds. The default of 0 indicates no time limit.
com.sybase.ejb.forceSSL	If set to true when using a a reverse proxy server, forces use of SSL for the connection to the reverse proxy. Set this property to true if the connection to the reverse proxy must use SSL (HTTPS) tunnelling, but the connection from the proxy to the server does not use SSL tunnelling. For more information, see Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide*.
com.sybase.ejb.GCInterval	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.
com.sybase.ejb.http	Specify whether proxies should use HTTP tunnelling without trying to use plain IIOP first. The default is `false`. With the default setting, the proxy tries to open a connection using plain IIOP, and switches to HTTP tunnelling if the plain IIOP connection is refused. The default is appropriate when some users connect through firewalls that require tunnelling and others do not; the same application can serve both types. If you know tunnelling is required, set this property to `true`. This setting eliminates a slight bit of overhead that is incurred by trying plain IIOP connections before tunnelling is used.

Property name	Description
com.sybase.ejb.http.jaguar35Compatible	When set to true, specifies that HTTP tunnelling must be compatible with version 3.5 or older Jaguar servers. The default is false. **Compatibility with version 3.5 or older servers** The default tunnelling model is incompatible with servers older than version 3.6. If you do not set the `com.sybase.ejb.jaguar35Compatible` property to true, clients using the EAServer 3.6 or later Java client runtime cannot connect to older-version servers using HTTP tunnelling. Note that HTTP tunnelling may happen automatically when clients connect to the server through firewalls.
com.sybase.ejb.HttpUsePost	When using HTTP tunnelling, specifies the HTTP request type used. A value of true indicates that POST requests are to be used. A value of false (the default) specifies that GET requests are to be used. Some Web browsers cannot handle the long URLs generated when using HTTP tunnelling with GET requests. Setting this property to true can work around the issue.
com.sybase.ejb.IdleConnectionTimeout	Specifies the time, in seconds, that a connection is allowed to sit idle. When the timeout expires, the ORB closes the connection. The default is 0, which specifies that connections can never timeout. The connection timeout does not affect the life of proxy instance references; the 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. A finite timeout also allows rebalancing of server load in an application that uses a cluster of servers.
com.sybase.ejb.isApplet *Applicable only to Java applets.*	Specifies whether the client application is a Java applet. The default is `false`. You must set this property to `true` in Java applets if the applet connects to EAServer using SSL (https).
com.sybase.ejb.local *Deprecated.*	For server-side component use only. Specifies whether the proxy references can be used to issue intercomponent calls in user-spawned threads. The default is `true`, which means that intercomponent calls are made in memory and must be issued from a thread spawned by EAServer. Set this property to `false` if your component makes intercomponent calls from user-spawned threads. **This property is deprecated** This property is not needed when calling components from threads spawned by the Thread Manager. The Thread Manager is the recommended way to spawn threads in Java components. See Chapter 32, “Using the Thread Manager” for more information.
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.

Property name	Description
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.
com.sybase.ejb. socketReuseLimit	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. The default may not be appropriate for a long-running client program that calls many methods from servers in a cluster. If sockets are reused indefinitely, the client may build an affinity for servers that it has already connected to rather than randomly distributing its server-side processing load among all the servers in the cluster. In these cases, the property should be tuned to best balance client performance against cluster load distribution. 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.
com.sybase.ejb. ProxyHost	Specifies the machine name or the IP address of a reverse proxy server. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.ejb. ProxyPort	Specifies the port number of a reverse proxy server. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.ejb. SSLCallback *Applicable only to Java application clients.*	Required if you are using SSL and you wish to provide a callback class to set required SSL settings on an as-needed basis. Specify the name of a Java class that implements the CtsSecurity.SSLCallbackIntf interface. For example: `com.acme.AcmeSSLCallback` Chapter 5, "Using SSL in Java Clients," in the *EAServer Security Administration and Programming Guide* describes how to code a callback class.
com.sybase.ejb. pin *Applicable only to Java application clients.*	Always required when using SSL. Specifies the PKCS #11 token PIN. This is required for logging in to a PKCS #11 token for client authentication and for retrieving trust information. This property cannot be retrieved. If not set, set to `any`, or set incorrectly, the ORB invokes the getPin callback method.
com.sybase.ejb. certificateLabel *Applicable only to Java application clients.*	Required when using SSL mutual authentication. Specifies the client certificate to use if the connection requires mutual authentication. The label is a simple name that identifies an X.509 certificate/private key in a PKCS #11 token. If the property is not set and the connection requires mutual authentication, the ORB invokes the getCertificateLabel callback method, passing an array of available certificate names as an input parameter.
com.sybase.ejb. qop *Applicable only to Java application clients.*	Always required when using SSL. Specifies the name of a security characteristic to use. See "Choosing a security characteristic" on page 146 for more information.

Property name	Description
com.sybase.ejb. useEntrustID *Applicable only to Java application clients.*	Specifies whether to use the Entrust ID or the Sybase PKCS #11 token for authentication. This is a Boolean (true or false) property. If this property is set to false, Sybase PKCS #11 token properties are valid and Entrust-specific properties are ignored. If this property is set to true, Entrust-specific properties are valid and Sybase PKCS #11 token properties are ignored.
com.sybase.ejb. entrustUserProfile *Applicable only to Java application clients.*	Specifies the full path to the file containing an Entrust user profile. This property is optional when the Entrust single-login feature is available and required when this feature is not available. If not set, the ORB invokes the getCredentialAttribute callback method.
com.sybase.ejb. entrustPassword *Applicable only to Java application clients.*	Specifies the password for logging in to Entrust with the specified user profile. This property is optional when the Entrust single-login feature is available and required when this feature is not available. If the password is required but not set or set incorrectly, the ORB invokes the getPin callback method. This property cannot be retrieved.
com.sybase.ejb. entrustIniFile *Applicable only to Java application clients.*	Specifies the path name for the Entrust INI file that provides information on how to access Entrust. This is required when the useEntrustid property is set to true. If not set, the ORB invokes the getCredentialAttribute callback method.
com.sybase.ejb. userData *Applicable only to Java application clients.*	Specifies user data (String datatype). This is an optional property. Client code can set user data during NamingContext initialization and access it using SSLSessionInfo::getProperty method in the SSL callback implementation. This may be useful as a mechanism to store context information that is otherwise not available through the SSLSessionInfo interface.
com.sybase.ejb. useJSSE	Use the Java Secure Sockets Extension (JSSE) classes for secure HTTP tunnelled (HTTPS protocol) connections. JSSE provides an alternative to the built-in SSL implementations when secure connections are needed from an applet running in a Web browser. Additional configuration may be required to use this option. See Chapter 5, "Using SSL in Java Clients," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.ejb. WebProxyHost *Applicable only to Java application clients.*	Specifies the host name or IP address of a Web proxy server. Applies to Java applications only. Java applets running in a Web browser will use the proxy address specified by the browser's proxy configuration. In Java applications, there is no default for this property, and you must specify both the host name and port number properties. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.

Property name	Description
com.sybase.ejb. WebProxyPort *Applicable only to Java application clients.*	Specifies the port number at which the Web proxy server accepts connections. Applies to Java applications only. Java applets running in a Web browser will use the proxy address specified by the browser's proxy configuration. In Java applications, there is no default for this property, and you must specify both the host name and port properties. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.ejb. HttpExtraHeader *Applicable only to Java application clients.*	An optional setting to specify what extra information is appended to the header of each HTTP packet when connecting through a Web proxy. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.

Choosing a security characteristic To use SSL, you must specify the name of an available security characteristic as the value for the com.sybase.ejb.qop property. The characteristic describes the CipherSuites the client uses when negotiating an SSL connection. When connecting, the client sends the list of CipherSuites that it uses to the server, and the server selects a CipherSuite from that list. The server chooses the first CipherSuite in the list that it can use. If the server cannot use any of the available CipherSuites, the connection fails.

"Configuring security profiles" in the *EAServer Security Administration and Programming Guide* describes the security characteristics that are provided with EAServer.

Set the qop property to `sybpks_none` to prevent any use of SSL on a connection.

Secure server addresses Client proxies will only connect to a server listener that uses an equivalent or greater level of security as requested in the `com.sybase.ejb.qop` setting. The URL specified with `java.naming.provider.url` cannot specify a server address that uses a higher level of security than specified by the qop property. For example, if your server uses the typical port configuration, you can specify port 9000 (no SSL) in the name service URL if the qop property specifies mutual authentication. However, you cannot specify port 9002 (mutual authentication) in the name service URL and set the qop property to request server-only authentication.

Configuring error output

The client runtime writes errors to the console by default. In Java applications, you can modify this behavior by creating a logging profile and specifying the profile name in the Java system properties. For more information, see "Using log profiles in Java client applications" in the *EAServer System Administration Guide*.

Running in Java applets

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

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

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

// URL for the Server's IIOP port. Host defaults to
// the applet download host.
p.put(Context.PROVIDER_URL, "iiop://:9000");

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

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

Setting the APPLET parameter activates the following convenient features:

- The host name can be omitted in the initial context URL that is specified as the PROVIDER_URL context parameter. The default host is the applet download host.
- You can set the com.sybase.ejb.autoProxy property and it will work as documented in Chapter 11, “Deploying Applications Around Proxies and Firewalls,” in the *EAServer Security Administration and Programming Guide*.

Resolving JNDI names

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

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

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

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

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

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

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

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

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

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

Instantiating remote or local interface proxies

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

Instantiating proxies for a session bean

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

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

Instantiating proxies for an entity bean

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

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

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

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

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

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

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

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

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

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

Calling remote interface methods

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

Calling local interface methods

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

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

- Local interfaces are available only to EJB components, Java servlets, and JSPs hosted on the same server as the target component. You must configure a local EJB reference for the call, as described in “Adding an EJB local reference” on page 384 and “Editing an EJB local reference” on page 384.
- If local interfaces are used, both the caller and the called component must be loaded by the same custom class loader.

When an EJB 2.0 component provides local interfaces, any other component or Web application that calls the local interface must use the same class loader. ClassCastException errors occur when local interface calls are made from entities that use a different class loader. Configure the custom class lists for the calling and called components and parent entries to allow sharing of the class instances as described in “Custom class lists for packages, applications, or servers” on page 560.

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 “Component properties: Transactions” on page 58 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
Stateful1 proxy;     // Active proxy instance

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

To deserialize the proxy

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

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

try {
```

```
    System.out.println("Deserializing proxy from "
      + _serializeFrom);
    FileInputStream istream = new
    FileInputStream(_serializeFrom);
    ObjectInputStream p = new ObjectInputStream(istream);
    Handle handle = (Handle)p.readObject();
    proxy = (Stateful1) handle.getEJBObject();
    istream.close();
  } catch (Exception e)
  {
    System.out.println(
      "Deserialization failed. Exception "
      + e.toString());
    e.printStackTrace();
    return;
  }
```

Runtime requirements

EJB clients require JDK 1.2 or later. If running applets, make sure your browser supports JDK 1.2. Most browsers require Sun's Java Plug-in to support JDK 1.2.

At run time, the following EAServer JAR files must be in the CLASSPATH for Java applications and included with the class files for applets:

- *java/lib/easclient.jar*
- *java/lib/easj2ee.jar*

Unlike earlier versions, EAServer 4.0 does not provide runtime class files in the *html/classes* directory. To run applets, you must include the JAR files in the applet's ARCHIVE tag, or expand these JAR files to the *html/classes* directory.

Chapter 4, "Creating Enterprise JavaBeans Components and Clients," in the *EAServer Cookbook* provides tutorial that describes how to deploy EJB clients and components.

CHAPTER 9

EAServer EJB Interoperability

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

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

This chapter describes:

Intervendor EJB interoperability

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

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

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

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

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

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

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

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

  - Interoperable naming, when using the interoperable name formats described in "Interoperable naming URLs" on page 157.
  - Transaction propagation, when using the OTS transaction model as described in "Transaction interoperability" on page 34
  - Security context propagation in accordance with the CSIv2 requirements outlined in the EJB 2.0 specification. For more information on this feature, see "Intercomponent authentication for EJB 2.0 components" in the *EAServer Security Administration and Programming Guide*.
  - Parameter and exception type inheritance and null value propagation in method invocations.

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

EAServer can simultaneously support RMI/IIOP and CORBA 2.2 clients. The client's interoperability requirements are automatically detected at run time. To use RMI/IIOP from another vendor's EJB 2.0 container, you must use the EAServer classes described in "Classes for RMI/IIOP connections from third-party containers" on page 159.

Interoperable naming URLs

You can use interoperable naming URLs for EJB 2.0 components and clients. Using an interoperable naming URL causes the EAServer runtime to use the RMI/IIOP protocol, which is required for EJB 2.0 interoperability features such as caller credential propagation. For more information on interoperable naming services, see Chapter 5, "Naming Services," in the *EAServer System Administration Guide*.

Interoperable naming URLs for EJB clients

To use RMI/IIOP as the network protocol, an EJB client must specify a `corbaname` interoperable naming URL as the value of the JNDI context's PROVIDER_URL property. When using `corbaname` URLs, you must specify the user name and password using the JAAS API, as described in "JAAS on the client" in Chapter 10, "Using the JAAS API," in the *EAServer Security Administration and Programming Guide*.

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

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

Or to use the default IIOP version number:

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

Where:

ver	Is an optional version number. Supported versions are 1.1 and 1.2. The default version is 1.1, unless you append the `#rmi:/` suffix, which forces the IIOP version to 1.2.
host	Is the server host name.
port	Is the server's IIOP port number.
[rmi]	Is the optional naming prefix `#rmi:/`, which specifies RMI Valuetype semantics. Valuetype semantics are required to propagate null parameter values in method calls. Using this option forces the IIOP version to 1.2.

For example, this URL specifies a connection to the host moxy at port 9000, using IIOP 1.2 with Valuetype semantics:

corbaname:iiop:1.2@moxy:9000/NameService#rmi:/

As another example, this URL specifies a connection to the host moxy at port 9000, using IIOP 1.2 without Valuetype semantics:

corbaname:iiop:1.2@moxy:9000/NameService

This URL identifies a connection to the host moxy at port 9000, using IIOP 1.1:

corbaname:iiop:moxy:9000/NameService

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

corbaname:iiop:1.2@moxy:9000#rmi:/

Or:

corbaname:iiop:1.2@moxy:9000

Interoperable naming URLs for EJB references

Servlets, JSPs, application clients, and EJB 2.0 components can use EJB references to alias names used to resolve EJB home interfaces in the implementation code. To use RMI/IIOP for invocations of the called component, you must specify a `corbaname` URL in the Link Value setting for the EJB reference.

To specify a name server address and IIOP version number, use a URL of the form:

corbaname:iiop:*ver*@*host*:*port*/NameService#*[rmi]comp-name*

To specify a name server address and use the default IIOP version of 1.1:

corbaname:iiop:*host*:*port*/NameService#*comp-name*

To specify a component that is installed in the same server or cluster:

corbaname:rir:/NameService#*[rmi]comp-name*

Where:

ver	Is an optional version number. The default version is 1.1. Supported versions are 1.1 and 1.2.
host	Is the server host name.
port	Is the server's IIOP port number.

[rmi]	Is the optional naming prefix `rmi:/`, which specifies RMI Valuetype semantics. Valuetype semantics are required to propagate null parameter values in method calls. This option requires IIOP 1.2. When connecting to another vendor's name service, the service may require a different naming prefix to specify RMI Valuetype semantics.
comp-name	Is the name with which the component is bound to the name service. For EAServer components, this is the value of the `com.sybase.jaguar.component.bind.naming` property, which defaults to *package-name/component-name* if not set.

For example, this URL references a component named *Finance/Accounting*, using the local name service and IIOP 1.2 with Valuetype semantics:

```
corbaname:rir:/NameService#rmi:/Finance/Accounting
```

This URL references the same component name, running on moxy at port 9000, using IIOP 1.2 and RMI Valuetype semantics:

```
corbaname:iiop:1.2@moxy:9000/NameService#rmi:/Finance/Accounting
```

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

```
corbaname:rir:#rmi:/Finance/Accounting
```

Classes for RMI/IIOP connections from third-party containers

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

- EJBMetaData
- Handle
- HomeHandle

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

- getEJBMetadata

- getHandle
- getHomeHandle

To call EJB components in EAServer from a third-party container, the EJBs must have been deployed from an EJB-JAR file or EAR file with the Use Interoperable Naming option checked.

Invoking non-EJB components from EJB clients

To invoke a non-EJB component from an EJB client, you must first create a home interface for the component, then generate EJB stubs for the component's interfaces. Then you can instantiate proxies for the non-EJB component using the standard EJB client design pattern.

Create a home interface

Non-EJB components can implement several remote interfaces. To instantiate proxies for any remote interface from EJB clients, you must use a corresponding home interface. Use EAServer Manager to create a home interface for each IDL interface that the component implements. The home interface must have a single create method that takes no parameters and returns the remote interface.

Even with a home interface defined, EAServer creates component instances using the standard method for the component's lifecycle model. A non-EJB component does not need to implement any additional methods to support the home interface.

❖ **Creating the home interface**

1 Expand the component's icon, then highlight the Interfaces folder beneath the component. Choose File | Set Home Interface.

2 In the Home Interface dialog box, enter the home interface name as:

```
module::component_interfaceHome
```

where *module* is the IDL module where the remote interface is defined, and *component_interface* is the base (non-nested) name of the remote interface. For example, MyPackage::MyComponentHome.

3 Click Add New. EAServer creates the interface and an icon for the interface appears in the component's Interfaces folder.

4 Highlight the home interface icon, and choose File | New Method.

5 Enter "create" as the method name and click Create New Method.

6 Change the following in the Method Properties dialog, leaving other fields at their default settings:

- **Returns** Enter the name of the corresponding remote interface, for example, MyPackage::MyComponent.
- **Exceptions raised** Enter CtsComponents::CreateException.

7 Click OK to close the Method Properties dialog box.

Generate EJB stubs

Use EAServer Manager to generate EJB stubs for the component, specifying a different package or code base for the EJB stubs than used by the existing CORBA stubs.

CORBA and EJB stubs share class names but do not share the same implementation. Therefore, each type must be in a different Java package or use a different code base.

"Generating EJB stubs" on page 138 describes how to generate the stub files.

Instantiating the home interface

After defining a home interface and generating EJB stubs, you can instantiate a home interface for the component using the standard EJB technique, as described in Chapter 8, "Creating Enterprise JavaBeans Clients".

The component's home interface name matches the value of the com.sybase.jaguar.component.bind.naming property. For non-EJB components, you must view the value of this property using the Advanced tab in the Component Properties dialog box. If the property is not set, the default is *initial-context*/*package*/*component*, where *initial-context* is the server's initial naming context, *package* is the EAServer Manager package where the component is installed, and *component* is the component name as displayed in EAServer Manager.

Calling methods in the remote interface

The methods in the Java remote interface follow the standard CORBA IDL-to-Java datatype mappings. To allow increased interoperability between EJB clients and non-EJB components, EAServer allows use of out and inout parameters in the remote interface. In the Java remote interface, these are represented by the same holder classes as used in CORBA stubs. See "Holder classes" on page 225 for more information on using holder classes.

Invoking non-EJB components from EJB components

In EJB component code, you can make intercomponent calls using EJB stubs and the EJB client interface as described above. You can also use the CORBA client interface as described in "Issue intercomponent calls" on page 197.

Invoking EJB components from CORBA C++ clients

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

Chapter 15, "Creating CORBA C++ Clients," describes how to use EAServer's C++ client ORB. This chapter provides new information on implementing clients that call EJB component methods.

Supported datatypes

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

Generating C++ header files

❖ **Generating C++ stubs for the EJB component's interfaces**

1 Highlight the component icon, or, to generate stubs for all components in a package, highlight the package in which the EJB component is installed.

2 Choose File | Generate Stub/Skeleton.

3 In the Generate Stubs and Skeletons wizard, check Generate Stubs and check Generate C++ stubs. Unselect Generate Java Stubs and Generate Skeletons. Specify a location for the generated files. The default is the EAServer *include* subdirectory.

EAServer Manager generates a header file for each IDL module that defines an interface or type used by the component. All class and type definitions are generated as inline code, so you need not compile the header files separately.

In the case of nested IDL modules, EAServer Manager generates a separate file for each nested module, following this naming pattern:

```
OuterModule::Module1::Module2::InnerModule
```

In this case, OuterModule includes Module1 which includes Module2 which includes InnerModule.

For example, for the IDL interfaces com::foo::interfaces::MyInterface and com::foo::interfaces::MyHomeInterface, these files are generated:

- *com.hpp* includes *com_foo.hpp* and headers for other modules nested within module com.
- *com_foo.hpp* includes *com_foo_interfaces.hpp* and headers for any other nested modules.

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

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

Using the home interface

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

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

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

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

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

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


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

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

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

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

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

bookStore::custMaintenance_var customer;
long balance = 2000;

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

```
{
    cout << "Error creating account for customer " << customer_name ;
}
```

Serializing and deserializing instance references

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

In a CORBA client, you can obtain the same functionality using the Orb.object_to_string and Orb.string_to_object methods. The same restrictions apply when deserializing bean proxies that apply to any other remote object. See Chapter 15, “Creating CORBA C++ Clients” for details.

Invoking EJB components from PowerBuilder clients

There are two ways to call EJB components from PowerBuilder:

- If using PowerBuilder 7.0 or later, you can call EJB components hosted in EAServer by generating proxies for the home, local, and remote interfaces then calling the lookup method on the PowerBuilder Connection object to instantiate the home interface proxy. Call the appropriate home interface create method to instantiate a proxy for the remote interface, then call the business methods as you would for any other EAServer component.
- If using PowerBuilder 9.0 or later, you can use the PowerBuilder EJB client interfaces. These interfaces use Java and Sybase-provided PowerBuilder extensions to invoke EJBs on any J2EE compatible application server. While this approach allows interoperability with servers from multiple vendors, the deployed client files are larger due to the need for a Java Runtime Environment and additional PowerBuilder libraries.

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

Invoking EJB components from ActiveX clients

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

For a description of EAServer's ActiveX client proxy, see Chapter 20, "Creating ActiveX Clients."

Supported datatypes

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

About overloaded methods and nested IDL

Most EJB home interfaces have overloaded methods, and many imported Java components use nested IDL modules.

Overloaded methods

The Java interfaces for an EJB component may have overloaded methods; that is, several methods with the same name that differ in the number and type of parameters. For example, a home interface may contain several create methods. EAServer maps such methods to uniquely named IDL methods by appending two underscores and a suffix to the Java method name. ActiveX does not support overloaded methods, so you must use the full IDL method names for methods that are overloaded in the Java interface.

For example, if a Java home interface has these methods:

```
mypackage.MyBeanRemote Create()

mypackage.MyBeanRemote Create(String p1, long p2)

mypackage.MyBeanRemote Create(
       String p1, String p2, long p3)
```

The IDL equivalent might be:

```
mypackage::MyBean Create()

mypackage::MyBean Create__String(string p1, long p2)
```

```
mypackage::MyBean Create__StringString(
        string p1, string p2, long p3)
```

To determine the full IDL method names, view the IDL interface in EAServer Manager.

Nested IDL modules

EAServer supports nested IDL modules. IDL modules that define the interfaces for an EJB component typically follow the Java package structure of the component's Java interfaces. For example, if the Java interfaces are in the Java package com.sybase.foo, IDL interfaces are in module com::sybase::foo. When implementing ActiveX clients, you must understand how nested IDL modules are mapped to ActiveX interface PROGIDs and the type names used in Object.Narrow_ calls.

The ActiveX PROGID for an IDL type defined in a nested IDL module follows this naming pattern:

```
module1_module2_module3.typeName
```

Each nested module name is preceded by an underscore, and the IDL type name is preceded by a period (.). For example, the PROGID for IDL type com::sybase::foo::MyBeanRemote is `com_sybase_foo.MyBeanRemote`.

When specifying type names in Object.Narrow_ calls, substitute a forward slash (/) for every double-colon (::) in the IDL type name. For example, if the IDL type is com::sybase::foo::MyBeanRemote, use `com/sybase/foo/MyBeanRemote` in the call to Object.Narrow_.

Using the home interface

An EJB home interface contains methods that return proxies for the component's remote interface.

The home interface for an entity bean contains finder methods that can be used to obtain instances that represent rows already in the underlying database.

Instantiating the home interface

To instantiate a home interface, use a SessionManager::Manager instance to create a SessionManager::Session instance, then call the SessionManager::Session::lookup method, passing the bean's home interface name. Narrow the returned object to the bean's home interface.

The example below instantiates the home interface named *bookStore/customerMaintenance*. In IDL, the home interface is bookStore::custMaintenanceHome:

```
' Initialize the ORB
Dim orbRef As JaguarTypeLibrary.ORB
```

```
Set orbRef = New JaguarTypeLibrary.ORB
orbRef.Init ("")

' Get a SessionManager::Manager proxy
Dim manager_ior As String
Dim CORBAObj As Object
Dim sessManager As SessionManager.Manager
manager_ior = Format("iiop://" & host & ":" & port)
Set CORBAObj = _
    orbRef.string_to_object(manager_ior)
Set sessManager = CORBAObj.Narrow_("SessionManager/Manager")

' Get a Session proxy, passing username and password
Dim session As SessionManager.session
Set CORBAObj = sessManager.createSession( _
    userName, password)
Set session = CORBAObj.Narrow_("SessionManager/Session")

' Get a proxy for the home interface
Dim home As bookStore.custMaintenanceHome
Set CORBAObj = session.lookup("bookStore/custMaintenance")
Set home = CORBAObj.Narrow_("bookStore/custMaintenanceHome")
```

Instantiating proxies for entity beans

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

A home interface may contain several finder methods, each of which accepts parameters to 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 CtsComponents::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 CtsComponents::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 structure has two members: *custName* is a string and *creditType* is also a string. The example looks for a customer named "Morry" using the findByPrimaryKey method. If a user exception is raised, the code assumes that CtsComponents::FinderException was thrown to indicate that the requested entity does not exist. In this case, the example calls a create method to create a new entity.

```
  Dim pKey As bookStore.custCreditKey
  Dim customerName as String
  customerName = "Morry"

  Set pKey = New bookStore.custCreditKey
  pKey.creditType = "VISA"
  pKey.custName = customerName

  Dim balance As Long

  ' First try to look up the customer as an existing entity
  ' This fails with CtsComponents::FinderException if the
  ' entity does not exist.
  On Error GoTo FinderError
  Set customer = home.findByPrimaryKey(pKey)
  GoTo Instantiated

FinderError:
  ' An error 9000 means a user-defined exception was thrown.
  ' In this case, it must be CtsComponents::FinderException,
  ' which indicates the requested entity does not exist. Any
  ’ other error number is unexpected.
  If Err.Number <> 9000 Then
     ' This is an unexpected error
     inError = True
     Call MsgBox("Error calling findByPrimaryKey", "Error")
     GoTo CleanupAfterFailure
  End If

  ' Create a new entity. Create methods are not overloaded in the
  ’ IDL home interface, and we must use the full IDL method name.
  On Error GoTo CleanupAfterFailure
  balance = 3000
  Set customer = home.create__string(customerName, balance)

Instantiated:
  ' Successful instantiation. Code to call business methods goes here.
```

```
CleanupAfterFailure:
  ' Unexpected error. Code to clean up forms, display errors,
  ' and so forth.
```

Instantiating proxies for session beans

The home interface for a session bean contains only create methods.

The example below instantiates a home interface named HelloWorldHome/HelloWorldHome, then calls the create method that takes no parameters. The IDL home interface type is mde::helloworld::HelloWorldHome and the remote interface is mde::helloworld::HelloWorld.

```
Dim session as SessionManager.Session
... deleted code that instantiated a valid session ...

Dim compHome As mde_helloworld.HelloWorldHome
Set CORBAObj = session.lookup("HelloWorldHome/HelloWorldHome")

Set compHome = CORBAObj.Narrow_("mde/helloworld/HelloWorldHome")
Set comp = compHome.Create().Narrow_("mde/helloworld/HelloWorld")
```

Serializing and deserializing instance references

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

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

Invoking EJB components from CORBA Java clients

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

Chapter 12, "Creating CORBA Java Clients," describes how to use EAServer's Java client ORB. This chapter provides new information on implementing clients that call EJB component methods.

Deciding whether to use EJB or CORBA interfaces

When using the EAServer client runtime, the EJB and CORBA interfaces offer identical functionality unless EJB 2.0 intervendor interoperability features are required (see "Intervendor EJB interoperability" on page 155). In this case, you must use the EJB client interace. If intervendor EJB interoperability is not required, choose the interfaces that you are most comfortable with.

Generating CORBA stubs

You cannot generate CORBA stubs to the same Java package as EJB stubs. Therefore, you must generate CORBA stubs using either a different code base or a different Java package than used by the existing EJB stubs. For example:

- If all clients will access an EJB component using the CORBA interfaces, you can generate CORBA stubs using EAServer's *html/classes* subdirectory as a code base. The EJB component uses the EJB stubs deployed under the *java/classes* subdirectory to marshall parameters and return values.
- If both CORBA and EJB clients will access an EJB component, you can generate CORBA stubs to a different Java package than that used by the existing EJB stubs. See "Specifying Java package mappings for IDL modules" on page 87.

❖ **Generating stubs in EAServer Manager**

1. Make sure the CORBA stubs will be generated to a different package or code base than existing EJB stubs. To change the Java package for CORBA stubs, follow the steps under "Specifying Java package mappings for IDL modules" on page 87.
2. Highlight the component icon, or to generate stubs for all components in a Package, highlight the package where the EJB component is installed.
3. Choose File | Generate Stub/Skeleton.
4. In the Generate Stubs and Skeletons wizard, select both Generate Stubs and Generate Java Stubs, and choose CORBA from the drop-down list of stub types. Uncheck Generate C++ Stubs and Generate Skeletons.
5. Specify a code base for the generated files. The default is EAServer's *html/classes* subdirectory.

Using the home interface

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

Instantiating the home interface To instantiate a home interface, use a SessionManager::Manager instance to create a SessionManager::Session instance, then call the SessionManager::Session::lookup method, passing the EJB component's home interface name. Call the narrow method in the helper class for the EJB component's home interface to narrow the returned object to the home interface.

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

```
org.omg.CORBA.Orb orb;

... deleted standard Orb initialization ...

org.omg.CORBA.Object obj = orb.string_to_object(_url);
Manager manager = ManagerHelper.narrow(obj);

Session session = manager.createSession("Guest", "GuestPassword");

// Create an instance of the home interface.

custMaintenanceHome custHome = custMaintenanceHomeHelper.narrow (
session.lookup("bookStore/custMaintenance") );
```

Instantiating a session bean The example below instantiates a home interface named *bookStore/inventory*, then calls the create method that takes no parameters. The IDL home interface type is bookStore::inventoryHome and the remote interface is bookStore::inventory.

```
SessionManager.Session session;

... deleted code that instantiated session ...

inventoryHome home = inventoryHomeHelper.narrow (
     session.lookup(_compName) );

if (home == null)
{
     System.out.println("Error: home interface is null.");
     return;
}

inventory inv = home.create();
```

Instantiating an entity bean This example instantiates an entity bean that represents a customer credit account. The primary key structure has two members: *custName* is a string and *creditType* is also a string. The example looks for a customer named Morry using the findByPrimaryKey method. CtsComponents::FinderException can be thrown to indicate that the requested entity does not exist. In this case, the example calls a create method to create a new entity.

```
// Obtain an instance of the remote interface. First check
// to see if the requested customer exists. If not, create
// a new entity.
String _custName = "Morry";
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 (CtsComponents.FinderException fe)
{
    System.out.println("Not found. Creating customer " + _custName + ".");
    try
    {
        cust = custHome.create(_custName, 2000);
    }
    catch (FinderException fe)
    {
        System.out.println("Error: could not create customer " + _custName);
    }
}
```

Serializing and deserializing instance references

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

In a CORBA client, you can obtain the same functionality using the Orb.object_to_string and Orb.string_to_object methods. The same restrictions apply when deserializing EJB component proxies as apply to any other remote object. See Chapter 12, “Creating CORBA Java Clients” for details.

Invoking EJB components using the MASP interface

An EJB component can be invoked from MASP as long as:

- The home interface has a create method that takes no parameters. This method is called to create an instance for the MASP invocation.
- The EJB component is not stateful.

MASP only supports primitive types, that is, those types listed in the drop-down list in EAServer Manager's Method Properties dialog box. From MASP, you can only call methods in the remote interface that meet these qualifications:

- All parameters are primitive Java types.
- The method returns a primitive Java type or void.

CHAPTER 10

Creating Application Clients

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

Creating an application client

An application client uses JNDI to look up and gain access to EJB components, resources, and environment properties defined in an XML deployment descriptor.

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

```
InitialContext initCntxt = new InitialContext();

Object acctRef =
   initCntxt.lookup("java:comp/env/ejb/acctBean");
acctBeanHome home = (acctBeanHome)
   PortableRemoteObject.narrow(acctRef,
   acctBeanHome.class);
Account acct = home.findByPrimaryKey(new
   AcctPK(1));
String name = acct.getName();
System.out.println(name);
```

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

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

❖ **Creating an application client in EAServer Manager**

1 Expand the top-level Applications folder, then expand the icon that represents your application, and highlight Clients.

2 Choose File | New Application Client and enter a name for the application client.

3 If necessary, create the EJB client. See Chapter 8, “Creating Enterprise JavaBeans Clients.”

4 Configure the application client properties. See “Configuring application client properties” on page 176.

Configuring application client properties

You can configure an application client’s properties in EAServer Manager. If you have created an Enterprise archive (EAR) file using another tool and imported it into EAServer, most properties are automatically set during the import process.

❖ **Displaying the Application Client Properties dialog**

To set the properties described in this section, first open the Application Client Properties dialog.

1 Expand the Clients folder, then highlight the icon that represents your application client.

2 Choose File | Application Client Properties.

General properties

Enter the application client's general properties:

- **Description** An optional text description of the application client.
- **Main Class** The main Java class of the application client in dot notation; for example, `com.sybase.appclient.Myclient`.

EJB references

Add references for the EJBs that the application client accesses in its code. For example, add the EJB reference *ejb/acctBean* in EAServer Manager and use *java:comp/env/ejb/acctBean* in your application client code.

"EJB references" on page 383 describes how to add or configure EJB references.

Resource references

Resource references are used to obtain connector and database connections, and to access JMS connection factories, JavaMail sessions, and URL links.

"Resource references" on page 385 describes how to add and configure resource references.

Resource environment references

Resource environment references are logical names applied to objects administered by EAServer, such as JMS message queues and topics.

"Resource environment references" on page 386 describes how to add and configure resource environment references.

Environment properties

Environment properties allow you to specify global read-only data for use by the application client.

Application clients must use JNDI to retrieve environment properties, using the prefix `java:comp/env` in JNDI lookups.

The deployment descriptor catalogs the environment properties used by the application client, as well as each property's Java datatype and default value.

"Environment properties" on page 387 describes how to add and configure environment properties.

Java classes

The Java Classes tab allows you to specify classes that must be included in the Application Client's run-time JAR file. "Configuring an entity's custom class list" on page 562 describes how to configure this setting.

JAXP properties

The settings on the JAXP tab configure the JAXP, DOM, and XSLT parser implementations used in the application client. See Chapter 36, "Configuring Java XML Parser Support,"for more information on these properties.

Application client files

The Files tab lists all the files in the application client, which are copied to *%JAGUAR%\Repository\<Application_Name>\<Client_Name>* when you deploy the application EAR file. When you export the client application, the files listed on this tab, plus the EJB stubs for the application are added to the export JAR file, which you deploy on a client machine to run the application.

Running application clients

To run an application client on a client machine:

- Copy the application client JAR file to the client's machine and import the JAR file, as described in Chapter 9, "Importing and Exporting Application Components," in the *EAServer System Administration Guide*.
- Set up the environment – see "Setting up a client's workstation" on page 179.

- Start the application client's runtime container – see "Starting the runtime container" on page 179.

Setting up a client's workstation

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

Starting the runtime container

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

Run application clients using the script *runclient.bat* (on Windows) or *runclient.sh* (on UNIX). Application clients require JDK 1.3 or later.

Use these options to define the runtime parameters:

Option	Description
- client	Application client JAR file
- name	Client's name
- login	Displays a login dialog to authenticate the client

This example illustrates the command-line syntax to start an application client's runtime container, where *my_appclient.jar* is the name of the application client JAR file and *my_client* is the name of the client. On Windows, the command is:

```
%JAGUAR%\bin\runclient -client my_appclient.jar -name my_client -login
```

On UNIX, the command is:

```
$JAGUAR/bin/runclient.sh -client my_appclient.jar -name my_client -login
```

PART 3

CORBA-Java Components and Clients

This part explains how to build Java components and clients that use standard CORBA type mappings and run-time services.

CHAPTER 11

Creating CORBA Java Components

This chapter describes how to create and install Java components using EAServer Manager and a separate development environment.

Requirements

The following list describes the software requirements for developing Java components and the hardware requirements for running Java components. All software that is required to run Java components in EAServer is supplied with the EAServer product.

- **Development** To create Java components, you need a development tool that supports JDK 1.2 or later and access to an EAServer installation. You must have Administrator Role access to define or modify components.

  For development, you can use any compatible Java compiler in combination with EAServer Manager, or you can use a Java Integrated Develelopment Environment (IDE) such as Borland JBuilder. If using JBuilder, the Sybase EAServer plugin allows direct deployment to EAServer from the IDE.

- **Runtime** Java components require that JDK 1.2 or later be installed on the server host machine. For detailed system requirements, see the EAServer *Release Bulletin* for your platform.

Procedure for creating Java components

A Java component is composed of Java class bytecode files. To create a Java component, you:

1 Define the component interface and properties

2 Choose implementation datatypes

3 Write the Java source file

4 Deploy Java components

Define the component interface and properties

The definition of a Java component specifies the interfaces that the component implements as well as its other properties.

Defining the client interfaces

All component interfaces for EAServer components are defined in CORBA IDL modules that are stored in EAServer's IDL repository. Chapter 5, "Defining Component Interfaces" describes how to define IDL interfaces.

Java component developers typically use one of the following to define the interface or interfaces that their component implements:

- **Implement a Java source file and import the methods from it** As an alternative to IDL, you can define a Java class or interface, then use EAServer Manager to import the method definitions from the compiled Java bytecode file. EAServer creates a new component definition and an IDL interface that matches the methods defined in the Java file. For more information on this feature, see "Importing interfaces from compiled Java files" on page 81.

- **Use existing interfaces from EAServer's IDL repository** In some cases, client and server component developers may have agreed upon an existing interface or several interfaces that your component must implement. In this case, it is up to you, the component developer, to implement the specified interface. EAServer stores HTML documentation for all interfaces in the IDL repository in the *html/ir* subdirectory of your EAServer installation.
- **Define a new IDL interface or interfaces** If you are defining the interface yourself, you can use EAServer Manager's IDL editor to create a new interface for the component. "Defining modules, interfaces, and types in IDL" on page 85 describes how.

If you have an IDL interface
If you are starting with an IDL interface rather than an existing class file, you can use EAServer Manager to create a class that contains the necessary method declarations. See "Generate stub, skeleton, and implementation files" on page 191 for more information.

Choose a control interface

Optionally configure a control interface for the component. Using a control interface allows you to implement methods to respond to changes in the instance lifecycle. See "Configuring a control interface" on page 72 for more information.

Specify component properties

In EAServer Manager, the Component Properties window configures the settings that EAServer uses to load the component and invoke its methods. See "Configuring component properties" on page 52 for more information.

Choose implementation datatypes

EAServer provides two component types for Java components. These component types are functionally equivalent, except that they use different mappings between IDL datatypes and the Java datatypes that are required in your implementation class. The choices are:

- **Java with IDL datatypes** The component's method declarations use the type mappings that are specified by the CORBA document, *IDL to Java Language Mapping Specification* (formal/99-07-53). To use these type mappings, specify Java - CORBA as the component type in the EAServer Manager Component Properties dialog box.

- **Java with JDBC datatypes** Predefined EAServer datatypes and IDL base types are mapped to types in the java.lang and java.sql packages. User-defined exceptions are not supported. User-defined IDL parameter and return types are mapped to Java datatypes using the standard CORBA IDL-to-Java mappings. Components that were developed for some earlier EAServer versions may use these type mappings. Components that throw com.sybase.jaguar.util.JException must use these mappings. To use these type mappings, specify Java - JDBC as the component type in the EAServer Manager Component Properties dialog box.

Use IDL types for new development
Java-JDBC type mappings are supported to provide backward compatibility with earlier EAServer versions. For new development, use the Java-CORBA types. Components using Java-JDBC type mappings cannot raise user-defined IDL exceptions; all exceptions must be thrown as the generic jaguar.util.JException class. If you import a Java class that uses jaguar.util.JException, the importer generates a Java-JDBC component.

The sections below describe the mappings in detail.

Java - CORBA component datatype mappings

The following table lists the datatypes displayed in EAServer Manager, the equivalent CORBA IDL types, and the Java datatypes used in Java/IDL component methods.

Table 11-1: EAServer Manager, CORBA IDL, and Java datatype equivalence

EAServer Manager display datatype	CORBA IDL type	Java type (input parameter or return value)	Java type (inout or out parameter)
integer<16>	short	short	org.omg.CORBA.ShortHolder
integer<32>	long	int	org.omg.CORBA.IntHolder
integer<64>	long long	long	org.omg.CORBA.LongHolder
float	float	float	org.omg.CORBA.FloatHolder
double	double	double	org.omg.CORBA.DoubleHolder
boolean	boolean	boolean	org.omg.CORBA.BooleanHolder
char	char	char	org.omg.CORBA.CharHolder
byte	octet	byte	org.omg.CORBA.ByteHolder
string	string	java.lang.String	org.omg.CORBA.StringHolder
binary	BCD::Binary	byte[]	BCD.Binary
decimal	BCD::Decimal	BCD.Decimal	BCD.DecimalHolder

EAServer Manager display datatype	CORBA IDL type	Java type (input parameter or return value)	Java type (inout or out parameter)
money	BCD::Money	BCD.Money	BCD.MoneyHolder
date	MJD::Date	MJD.Date	MJD.DateHolder
time	MJD::Time	MJD.Time	MJD.TimeHolder
timestamp	MJD::Timestamp	MJD.Timestamp	MJD.TimestampHolder
ResultSet	TabularResults::ResultSet	TabularResults.ResultSet	TabularResults.ResultSetHolder
ResultSets	TabularResults::ResultSets	TabularResults.ResultSet[ ]	TabularResults.ResultSetsHolder

Binary, Fixed-Point, and Date/Time types The BCD and MJD IDL modules define types to represent common database column types such as binary data, fixed-point numeric data, dates, times. The BCD::Binary CORBA type maps to a Java byte array. The other BCD and MJD types map to data representations that are optimized for network transport.

To convert between the IDL-mapped datatypes and from core java.* classes, use these classes from the com.sybase.CORBA.jdbc11 package:

Class	Description
SQL	Contains methods to convert from BCD.* and MJD.* types to java.* types
IDL	Contains methods to convert from java.* types to BCD.* and MJD.* types

Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference* provides reference pages for these classes.

Result set types The TabularResults IDL module defines types used to represent tabular data. Result sets are typically used only as return types, though you can pass them as parameters.

"Return result sets" on page 199 describes how to create and return result sets.

User-defined IDL types A user-defined type is any type that is:

- Not in the set of datatypes that is not predefined by EAServer's read-only repository modules and
- Not one of the CORBA IDL base types.

If a method definition includes user-defined types, the Java component method will use the equivalent Java datatype as specified by the CORBA Java language mappings specification. See "Overview" on page 207 for more information on this document.

CORBA Any and TypeCode support
EAServer's Java ORB supports the CORBA Any and TypeCode datatypes. Refer to the OMG CORBA 2.3 specification and *IDL to Java Language Mapping Specification* (formal/99-07-53) for information on using these types.

Holder classes for IDL types All IDL-mapped Java types have an accompanying holder class that is used for passing parameters by reference. Each holder class has the following structure:

```
public class <Type>Holder {
    // Current value
    public <type> value;
    // Default constructor
    public <Type>Holder() {}
    // Constructor that sets initial value
    public <Type>Holder(<type> v) {
      this.value = v;
    }
}
```

This structure is defined by the CORBA Java-language bindings specification.

Java - JDBC component datatype mappings

Java-JDBC type mappings are supported to provide backward compatibility with earlier EAServer versions. For new development, use the Java-CORBA types. Components using Java-JDBC type mappings cannot raise user-defined IDL exceptions; all exceptions must be thrown as the generic jaguar.util.JException class.

The table below shows the datatypes displayed in the EAServer Manager, the datatypes used by Java components, and the argument modes (in, inout, out for parameter passing modes and return to indicate the type is used for method return values).

inout and out parameters for these datatypes are passed in holder classes from the com.sybase.jaguar.util and com.sybase.CORBA.jdbc11 packages. For more information on these packages, see the reference pages in Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference*.

Table 11-2: Java - JDBC component datatype mappings

EAServer Manager datatype	IDL type	Mode	Java type
boolean	boolean	input, return inout, out	boolean BooleanHolder
binary	BCD::Binary	input, return inout, out	byte[ ] BytesHolder
byte	octet	input inout, out	byte ByteHolder
date	MJD::Date	input, return inout, out	java.sql.Date com.sybase.jaguar.util.jdbc11.DateHolder
decimal	BCD::Decimal	input, return inout, out	java.math.BigDecimal com.sybase.jaguar.util.jdbc11.BigDecimalHolder
double	double	input, return inout, out	double DoubleHolder
float	float	input, return inout, out	float FloatHolder
integer<16>	short	input, return inout, out	short ShortHolder
integer<32>	long	input, return inout, out	int IntegerHolder
integer<64>	long long	input, return inout, out	long LongHolder
money	BCD::Money	input, return inout, out	java.math.BigDecimal com.sybase.jaguar.util.jdbc11.BigDecimalHolder
string	string	input, return inout, out	java.lang.String StringHolder
time	MJD::Time	input, return inout, out	java.sql.Time com.sybase.jaguar.util.jdbc11.TimeHolder

EAServer Manager datatype	IDL type	Mode	Java type
timestamp	MJD:: Timestamp	input, return inout, out	java.sql.Timestamp com.sybase.jaguar.util.jdbc11.TimestampHolder

User-defined IDL types in a method declaration are mapped to the same Java classes as for a Java/IDL component. See “User-defined IDL types” on page 187 for more information.

Methods in a Java-JDBC component do not return result sets explicitly. If the IDL method definition indicates a result set or result sets are returned, the Java method must be declared to return `void`, and the implementation must use the EAServer JServerResultSet and JServerResultSetMetaData interfaces to send result sets back to the client.

Write the Java source file

When you code the parameters for each method, make sure you use the Java datatype that corresponds to the datatype you defined in the EAServer Manager.

If you have an IDL interface

If you are starting with an IDL interface rather than an existing class file, you can use EAServer Manager to create a class that contains the necessary method declarations. See “Generate stub, skeleton, and implementation files” on page 191 for more information.

❖ **Implementing the component**

1 Generate stub, skeleton, and implementation files – Generate the files required to run the component. If you are starting development with an IDL interface, and not an existing Java class or interface, EAServer Manager will generate a sample implementation with all the required method signatures.

2 Add package import statements – Import the packages that contain the classes that you need to use in your Java class.

3 Code the constructor – Provide a default constructor to be called when EAServer loads the implementation class.

4 Implement control interface methods – Implement the control interface methods to respond to changes in the instance lifecycle.

5 Add error handling code – Add code that gracefully handles errors by logging status messages and sending meaningful messages to the client.

6 To finish up, you can use these advanced technique to polish your component implementation:

   a Manage database connections – Connect to databases through connection caches using the Connection Management API.

   b Return result sets – Return result sets using the EAServer Result Sets API.

   c Issue intercomponent calls – Instantiate a Java stub to make intercomponent calls.

Generate stub, skeleton, and implementation files

Use EAServer Manager to generates stubs and skeletons for the component. EAServer Manager will also create a sample implementation template for the class that implements the component methods.

Note Internally, EAServer's IDL-to-Java compiler is invoked by EAServer Manager to generate Java stubs and skeletons. The direct compiler interface is not intended for customer use.

What the skeleton does The skeleton class interprets component invocation requests and calls the corresponding method in your component with the parameter values supplied by the client. When a client sends an invocation request, the skeleton reads the parameter data and calls the Java method. When the method returns, the skeleton sends output parameter values, return values, and exception status to the client.

You must generate a new skeleton class if:

- You *install the component* in a different EAServer package,
- You *change the name* of the implementation class or move it to a different Java package,

- You *add a method* to the component interface,
- You *delete a method* from the component interface, *or*
- You *change the signature* of an existing method in the component interface.

Using the sample implementation EAServer Manager creates a sample source for the implementation class that is specified in the Component Properties window. The generated template file name is:

```
component Impl.java.new
```

where component is the name of the component. The *.new* extension avoids conflicts with existing source files.

The sample implementation provides a starting point for your own implementation, as it contains all the required method definitions to match the IDL interfaces that the component implements. Each method has the same name as the IDL operation it implements, and uses return and parameter datatypes that are mapped according to the type mappings that you have chosen (see "Choose implementation datatypes" on page 185).

In the Java component, component interface methods must be public and cannot be declared static. If the IDL definition of the method has a non-empty raises clause, the Java method must throw equivalent Java exceptions for the IDL exceptions listed in the raises clause.

All methods in the implementation throw the exception org.omg.CORBA.NO_IMPLEMENT. Replace this code with your own method implementation.

If you've added methods to an existing component, you can copy the additional method signatures from the *.new* file to your original source file.

Stubs may be required to compile the component If the component's definition uses user-defined types for parameters, return values, or exceptions, Java stubs are required for these types. These stubs are generated when you generate stubs for your component, as described in "Generating Java stubs" on page 209.

Compiled Java stubs for user-defined IDL types must be available when you compile your component's implementation file.

❖ **Generating skeletons**

1 Select the component or, if you want to generate skeletons for all components in a package, select the package.

2 Select File | Generate Stub/Skeleton. The Generate Stubs & Skeletons Wizard displays. Follow the instructions on each page to generate skeletons. See the online help for descriptions of any input fields that you do not understand.

EAServer Manager generates the skeleton source file into the same Java package as the component's implementation class. Skeletons are named as _sk_*Package_Comp*.java, where *Package* represents the EAServer package name and *Comp* represents the component name.

You must compile the Java class that implements the component before you compile the skeleton class. If the class file is available when you generate skeletons, you can select the Compile Skeletons option in the wizard to compile from EAServer Manager.

When you compile the skeleton class, make sure that the CLASSPATH setting contains the code base directory, as well as the following JAR files in the EAServer installation directory:

- *java/lib/easserver.jar*
- *java/lib/easclient.jar*
- *java/lib/easj2ee.jar*

Add package import statements

In addition to any Java packages that you might need, you might also need to import several Java packages. Classes coded with IDL datatypes and classes coded with SQL datatypes require different import statements.

Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference* provides reference pages for these packages.

Imports for classes implemented with SQL datatypes

The packages below are useful if your component is implemented using SQL datatypes:

Package(s)	Description
com.sybase.jaguar.server	Contains utility classes for use in server-side Java code.
com.sybase.jaguar.sql	Defines interfaces for defining and sending result sets. See "Sending result sets with Java" on page 464 for details on using these classes.

Package(s)	Description
com.sybase.jaguar.jcm	Provides the Java Connection Management (JCM) classes. See Chapter 26, “Using Connection Management” for a description of this feature.
com.sybase.jaguar.util com.sybase.jaguar.util.jdbc11	Contain the JException class and the holder classes that are used to pass inout and out parameter values.

The fragment below shows the import statements for all of these classes:

```
import com.sybase.jaguar.server.*;
import com.sybase.jaguar.util.*;
import com.sybase.jaguar.util.jdbc11.*;
import com.sybase.jaguar.sql.*;
import com.sybase.jaguar.jcm.*;
import com.sybase.jaguar.beans.enterprise.*;
```

You can also import com.sybase.jaguar.*, but you must remember to include the rest of the package name when you specify methods.

Imports for classes implemented with IDL datatypes

The packages below are useful if your component is implemented using the standard CORBA IDL-to-Java datatype mappings:

Package(s)	Description
org.omg.CORBA	Contains Java holder and helper classes for each of the core CORBA datatypes. Also defines the interfaces for a standard Java client-side Object Request Broker.
com.sybase.CORBA.jdbc11.*	Contains utility classes for converting between EAServer IDL datatypes and core Java datatypes.
com.sybase.jaguar.server	Contains utility classes for use in server-side Java code.
com.sybase.jaguar.sql	Defines interfaces for defining and sending result sets. See “Sending result sets with Java” on page 464 for details on using these classes.
com.sybase.jaguar.jcm	Provides the Java Connection Management (JCM) classes. See Chapter 26, “Using Connection Management” for a description of this feature.

Package(s)	Description
com.sybase.jaguar.util.JException	Many of the methods in the EAServer Java classes throw JException. Note that the packages com.sybase.jaguar.util and org.omg.CORBA contain identically named classes, so you can not import all classes from both packages. To avoid compilation problems, import JException explicitly or always refer to this class by its full name.

The fragment below shows the import statements for all of these classes:

```
import org.omg.CORBA.*;
import com.sybase.CORBA.jdbc11.*;
import com.sybase.jaguar.util.JException;
import com.sybase.jaguar.server.*;
import com.sybase.jaguar.sql.*;
import com.sybase.jaguar.jcm.*;
```

Code the constructor

A class constructor is normally used to initialize instance-specific data. However, if your component implements a control interface, then you should use the control interface methods to manage instance-specific data. Otherwise, instance-specific initialization must be done in the constructor.

Any uncaught exception that is thrown within the constructor aborts the creation of the new component instance.

Implement control interface methods

You can specify a control interface to be implemented by your component as described in “Configuring a control interface” on page 72. At runtime, EAServer calls the control interface methods to indicate changes in the instance lifecycle. For example, if you use CtsComponents::ObjectControl:

- The setObjectContext method provides an ObjectContext instance. Among other features, the object context allows you to:
  - Control transactions.
  - Obtain the component’s EAServer Manager properties, allowing you to read user-defined properties in EAServer Manager.

- The ctsActivate method indicates that the instance has been bound to a client session.
- The ctsDeactivate method indicates that the instance has been unbound from a client session.

You can also implement CORBA components that use the EJB session or entity design pattern using the CtsComponents::ObjectControl control interface. For more information on these methods, see the generated CtsComponents::ObjectControl HTML documentation in the *html/ir* directory of your EAServer installation.

Add error handling code

Errors occurring during component execution should be handled gracefully as follows:

1 Write detailed descriptions of the error to the log. This will help you debug the problem later. You can call any of the System.out.print methods to write to the log (the output is redirected).

2 If the error prevents completion of the current transaction, roll it back as described in "Set transactional state" on page 200.

3 Throw an exception with a brief, descriptive message that is appropriate for display to an end user of the client application.

Java components can record errors or status messages to the server's log file. Writing to the log creates a permanent record of the error, and log messages can be automatically stamped with the date and time that the message was written. Call any of the System.out.print methods to write to the log.

You can also throw an uncaught exception. Ideally, any exception thrown by your component should be a standard CORBA IDL exception or a user-defined IDL exception (the latter must be listed in the raises clause of the IDL method definition and the throws clause of the equivalent Java method declaration). All exceptions are forwarded to the client, but only exceptions that are defined in IDL can be rethrown by the client stub as a duplicate of the server-side exception. CORBA ORB and EAServer EJB clients receive forwarded exceptions differently:

- *CORBA ORB clients* rethrow any exception that is defined in IDL as a duplicate of the original exception. Other exceptions are rethrown as the standard CORBA exception UNKNOWN.

- *EAServer EJB clients* rethrow any server exception as a JException instance with the message text returned by calling toString() on the original exception.

Advanced techniques

After the basic component implementation is in place, you can add code to perform the following advanced tasks:

- "Issue intercomponent calls" on page 197
- "Manage database connections" on page 198
- "Return result sets" on page 199
- "Access SSL client certificates" on page 199
- "Set transactional state" on page 200

Issue intercomponent calls

You must use a proxy to issue intercomponent calls. If you call methods in another Java component directly, no server features are available to the called component, such as transaction control, instance lifecycle management, and security.

Using the CORBA ORB to instantiate proxies

To invoke other components, instantiate a proxy (stub) object for the second component, then use the stub to invoke methods on the component.

To invoke methods in other components, create an ORB instance to obtain proxy objects for other components, then invoke methods on the object references. You obtain object references for other components on the same server by invoking string_to_object with the IOR string specified as *Package*/*Component*. For example, the fragment below obtains a proxy object for a component *SessionInfo* that is installed in the *CtsSecurity* package.

```
    java.util.Properties props = new java.util.Propert
ies();
    props.put("org.omg.CORBA.ORBClass",
                    "com.sybase.CORBA.ORB");
    ORB orb = ORB.init((java.lang.String[])null, props
);
    SessionInfo sessInfo =
```

```
SessionInfoHelper.narrow
    (orb.string_to_object(
        "CtsSecurity/SessionInfo"));
```

When making intercomponent calls using string_to_object, the user name of the client that executed the component is automatically used for authorization checking. The exception is when instantiating the system components in the Jaguar package: the ORB automatically switches to the system user priveleges when you specify a component in the Jaguar package. To specify a user name, use this syntax:

```
orb.string_to_object("iiop://0:0:user_name:password/Package/Component"));
```

You can retrieve the system user name and password with these methods in class com.sybase.CORBA.ORB, which both return strings:

- getSystemUser() returns the system user name.
- getSystemPassword() returns the system password.

When called from components, string_to_object returns an instance running on the same server if the component is locally installed; otherwise, it attempts to resolve a remote instance using the naming server.

Connecting to third-party CORBA servers

Your component may need to invoke methods on a component hosted by another vendor's CORBA server-side ORB. Sybase recommends that Java components use the EAServer client-side ORB for all IIOP connections made from EAServer components. See "Connecting to third-party ORBs using the EAServer client ORB" on page 240 for more information.

Manage database connections

If your Java methods connect to remote data servers, you should use EAServer's connection caching feature to improve performance. See Chapter 26, "Using Connection Management" for more information.

Note EAServer's transactional model works only with connections obtained from the EAServer Connection Manager. Connections that you open yourself will not be able to participate in EAServer transactions.

Return result sets

Using the JDBC API, a Java component can retrieve result sets from a database. Using classes in the *com.sybase.jaguar.sql* package, Java components can also send these result sets to the caller. A Java component can combine the data from several result sets retrieved from databases and send that data as a single result set to a Java client. A Java component can also forward the original result set retrieved from a database.

To learn how to return result sets, see "Sending result sets with Java" on page 464.

Access SSL client certificates

Clients can connect to a secure IIOP port using an SSL client certificate. You can issue intercomponent calls to the built-in *CtsSecurity/SessionInfo* component to retrieve the client certificate data, including:

- The distinguished SSL user name
- The client certificate fingerprint (MD5 message digest)
- The client certificate data
- The chain of issuing certificates

This component implements CtsSecurity::SessionInfo IDL interface. HTML documentation is available for the interface in the *html/ir* subdirectory of your EAServer installation. You can view it by loading the main EAServer HTML page, then clicking the "Interface Repository" link.

The CtsSecurity::UserCredentials interface is deprecated

The CtsSecurity::UserCredentials interface, which is implemented by the *CtsSecurity/UserCredentials* component, has been replaced by the CtsSecurity::SessionInfo interface, which provides additional functionality such as certificate parsing. EAServer supports the CtsSecurity::UserCredentials interface for backwards compatibility. Please use the interface CtsSecurity::SessionInfo if developing new components.

Set transactional state

The transactional state of a component instance determines whether a transactional component's database updates are committed or rolled back.

In components that use the CtsComponents::ObjectControl control interface, each instance receives a CtsComponents::ObjectContext object each time that EAServer calls the setObjectContext method. The object reference is valid until unsetObjectContext is called. For more information on these methods, see the generated HTML documentation in the *html/ir* directory of your EAServer installation.

In classes that do not implement a control interface, call Jaguar.getInstanceContext() in each method that sets transactional state (do not save the object across method invocations, because it will not be valid if the component instance has been deactivated and reactivated). See the *EAServer API Reference Manual* for information on this method.

To set transaction state, choose the method that reflects the state of the work that the component is contributing to the transaction, as follows:

- If the work is complete and without error, call setComplete.
- Call setRollbackOnly if the work cannot be completed. Alternatively, throw the exception org.omg.CORBA.TRANSACTION_ROLLEDBACK. If the error indicates an internal inconsistency in the application, log a description of the error to help debug the problem as described in "Add error handling code" on page 196.

Transaction control with the ServerBean control interface

If you use the deprecated control interface JaguarEJB::ServerBean and Auto demarcation/deactivation option is disabled in the Transactions tab in the Transactions properties for your component, the transaction state specified in the method determines whether the instance is deactivated or remains bound to the client.

Deploy Java components

This section describes how to deploy a Java component to a server for the first time for development testing. Deployment to production servers is typically performed by exporting and importing EAServer packages, as described in "Deploying components" on page 11.

❖ **Deploying Java components to EAServer**

1 Determine the Java code base directory from which EAServer will load your component's classes. To allow refresh of the component, use the EAServer *java/classes* subdirectory, and add necessary classes and JAR files to the Java Classes tab in the Component Properties dialog box. See "Custom class lists for Java and EJB components" on page 556 for more information.

Note For security reasons, it is preferable to deploy Java components to the *java/classes* subdirectory or some other directory that is not accessible to HTTP downloads. See "Security considerations for deployment" on page 201 for more information. 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.

2 Under the code base directory, copy the Java component and skeleton class files. When copying class files, preserve the package subdirectory structure.

3 Copy other class files and JAR files that your component depends on to the same codebase. For example, you may need to copy utility classes that are in your component's package.

Security considerations for deployment

Your application may have a potential security hole if Java component implementation classes are deployed under the EAServer *html* directory. An unauthorized user can implement a program that connects to EAServer's HTTP port and downloads the component's implementation classes. The user can then decompile the classes and gain access to potentially sensitive information such as database passwords. To close this security hole, Sybase recommends one of the following approaches:

- Deploy Java component implementation classes under the EAServer *java/classes* subdirectory.

- Code components that retrieve connection caches to use the getCacheByName API rather than the APIs that require a database password. This approach removes database passwords from your Java class file. However, other sensitive information may still be present in the Java class file.
- Implement your Java components to retrieve potentially sensitive information from a properties file that is not located beneath the EAServer HTML directory.

Refreshing Java components

You can refresh a component's implementation classes while the server is running. You do not need to shut down and restart the server. All classes that can be refreshed must be deployed under the EAServer *java/classes* subdirectory. Classes loaded from a different code base directory will not be reloaded. EAServer only reloads the component's implementation class, the skeleton class, and any classes on the Java Classes tab in the Component Properties Dialog box. "Custom class lists for Java and EJB components" on page 556 describes how to configure the custom class list.

❖ **Refreshing a component**

1 Copy new versions of the changed class or JAR files to the EAServer *java/classes* subdirectory. If you are adding new classes or JAR files, you may need to add them to the custom class list as described in Chapter 30, "Configuring Custom Java Class Lists."

2 In EAServer Manager, select the component, or to refresh all components in a package, select the package.

3 Choose File | Refresh from the menu.

If the com.sybase.jaguar.component.refresh property is set to `false` (the default is `true`), the component cannot be refreshed. This property must be set on the Advanced tab in the Component Properties window. See "Component properties: Advanced" on page 71 for more information.

Disabling component refresh

In some cases, you may want to disable refresh for Java components. You can do so by setting the `com.sybase.jaguar.component.refresh` component property to false.

When refresh is disabled, all Java classes that your component depends on must be deployed under a Java code base that is specified in the server's CLASSPATH environment variable. If the component uses classes in a JAR file, you must add the JAR file to the server CLASSPATH variable.

If you deploy your component files to the class tree that starts in the EAServer *java/classes* or *html/classes* subdirectory, they will be in the server CLASSPATH by default. If you deploy to another location, add this location to the CLASSPATH setting for the server process.

Debug Java components

You can debug Java components that are executing in EAServer.

Before you start

Debugging Java components requires the following:

- If using the default server configuration, the debug version of the server must be running. "Starting the server" in the *EAServer System Administration Guide* describes how to start the debug server. If running JDK 1.3 or later, you must specify a port for debugger socket connections on the JDPA tab in the EAServer Manager Server Properties dialog box. Restart the server in debug mode for the change to take affect.

  To enable debugging in the production server, configure the EAServer Manager server properties. Display the Advanced tab, and modify the property com.sybase.jaguar.server.jvm.options. Append the following to the existing value, all on one line. Change *port* to and unused port number on your server machine, to specify the port number for JDPA debugger connections:

```
,-Xdebug,
'-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=port',
'-DJaguarServletDebugging=true','-Djava.compiler=NONE','-Xnoagent'
```

- You need a Java debugger that supports remote debugging and is compatible with the JDK version used by the host server, such as JBuilder or the JDK jdb tool.
- Component class files must include symbol information for fields and local variables. If you are compiling with javac, components must be compiled with the `-g` option. If you use a Java IDE, set the equivalent option to build debuggable class files.
- Source files must be available to the debugger. The required location for source files depends on your debugging tool:
  - For jdb, source files must be on the server host, in the directory from which the corresponding class file is loaded.

- JBuilder uses local source files from the project directory.
- For other IDE debuggers, consult the IDE documentation for source file requirements when you are debugging remotely.

Steps to debug an executing component

❖ **Debugging Java components**

1 Make sure that you have followed the requirements in “Before you start” on page 203.

2 Make sure no one else is debugging components using the same server. The server-side debugger supports only one remote-debugger connection at a time.

3 Run your debugger. The debugger JDK version must match the server’s JDK version. You must specify the server’s host name and JPDA port, the JDPA port is specified on the JDPA tab in the server properties window. For example, using jdb to connect to a server running on machine “myhost,” using port 11000 for JPDA connections:

```
jdb -connect com.sun.jdi.SocketAttach:hostname=myhost,port=11000
```

4 Once connected, you can specify breakpoints for the debugger to stop at in your component implementation. Set the breakpoint before running any client applications that will exercise the target line of code. For example (the following must be entered on one line):

```
stop at Sample.SVU.javaComp.
Enrollment_Java.Enrollment_JavaImpl:74
```

5 Run a client application that will invoke the target component code. For example, to debug the CreateStudentRecord method in the Java_Enrollment component, run the SVU Java example client in your Web browser and create a new student record on the login screen. In the debugger, you should see some indication that the breakpoint has been reached.

Note You may need to change the debugger’s active thread before the component breakpoint is indicated. EAServer runs each Java component instance in a separate Java thread. The component thread must be the active debugger thread in order to step through the component’s code. In jdb, use the threads command to list the executing threads. You can then use the thread command to switch to the thread that is running the component.

---

6 Once the breakpoint is tripped, you can step through the code, inspect fields and variables, and so forth. For example, in jdb:

```
Thread-21[1] where 6

  [1]
Sample.SVU.javaComp.Enrollment_Java.Enrollment_Java
Impl.
createStudentRecord (Enrollment_JavaImpl:74)

Thread-21[1] print student_id

student_id = 333-33-3333

Thread-21[1] cont
```

Debugging class loading and unloading problems

EAServer Manager provides the following server properties to log information about Java class loading and unloading. You can set these properties on the Advanced tab in the Server Properties dialog box:

Property	Description
com.sybase.jaguar.server.jvm.verbose	When this property is set to `true`, the server logs all Java classes that are loaded and the location where each class file was read.
com.sybase.jaguar.server.jvm.verboseGC	When this property is set to `true`, the server logs information when memory is freed by the Java garbage collector.
com.sybase.jaguar.server.classloader.debug	When this property is set to `true`, the server logs information about loading classes from custom class lists defined for the component, package, application, or server.

The default for all of these properties is `false`.

CHAPTER 12

Creating CORBA Java Clients

EAServer includes a Java implementation of a standard CORBA Object Request Broker (ORB). EAServer's Java ORB brings the portability of CORBA and Java to EAServer applications. This chapter describes how to code a CORBA-compatible Java client application that calls EAServer component methods.

Overview

CORBA is a distributed component architecture defined by the Object Management Group. EAServer supports the CORBA Internet Inter-ORB Protocol (IIOP). EAServer also provides a CORBA-compatible client-side interface that is implemented according to the CORBA specification for IDL-to-Java language mappings. These two items allow you to create CORBA-compliant Java applications and applets that interact with EAServer components.

About CORBA Java language bindings

For information on the CORBA architecture, see the specifications available at the Object Management Group (OMG) Web site at http://www.omg.org.

The EAServer Java ORB runtime is implemented according to the CORBA 2.3 specification (specifically, the document *IDL to Java Language Mapping Specification*, formal/99-07-53). You can download this document from the OMG Web site at http://www.omg.org.

EAServer Java ORB runtime

The Java ORB programming interface is defined by the CORBA Java-language bindings specification. The top-level class, org.omg.CORBA.ORB, is an abstract Java class. Each Java ORB vendor must provide an implementation of this class. For example, the EAServer ORB implementation class is com.sybase.CORBA.ORB. You can use the EAServer ORB or any CORBA-compatible ORB to invoke EAServer components.

In this version, EAServer's ORB implementation does not support:

- Method invocation via the Dynamic Invocation Interface (DII)
- The CORBA::Any type

Procedure for creating CORBA-compatible Java clients

A Java client establishes a session with the application server, instantiates stub (or proxy) instances for EAServer components, and executes component methods by calling like-named methods on the stub instance.

1 Generate stub classes.

  These classes act as a proxy object for a component instance that is executing on the server; there is one stub for each IDL interface that the component implements. "Generating Java stubs" on page 209 describes how to generate stubs with EAServer Manager.

2 Implement code to instantiate proxy objects.

  Your program must obtain proxy objects for the EAServer component and narrow them to the stub interface that you intend to use. EAServer supports three techniques for proxy instantiation, using different interfaces for resolving component names to server objects. "Instantiating proxy instances" on page 212 describes each technique in detail.

3 Implement code that invokes the component methods.

You execute the component's methods by calling like-named methods on the stub class and passing the necessary input data. Each stub method has a return value and parameter list that is mapped from the corresponding EAServer Manager method definition. "Executing component methods" on page 223 describes return type and parameter type mappings in detail.

4 If desired, you can serialize the component instance reference as an IOR string, then deserialize the reference later.

See "Serializing component instance references" on page 227 for details.

5 Clean up client-side resources.

When proxy objects are no longer required, set the references to null to expedite cleanup by the Java garbage collection mechanism. See "Cleaning up client resources" on page 227 for details.

Each of these steps requires appropriate exception handling. "Handling exceptions" on page 228 summarizes CORBA exceptions.

Generating Java stubs

Stub classes allow you to instantiate local Java objects that act as proxies for an instance of the EAServer component. You can generate Java stubs for components that are implemented in any of EAServer's supported component models. One stub interface is generated for each IDL interface that the component implements.

When using the EAServer ORB runtime, you must generate stubs with EAServer Manager and compile them with a Java compiler. If you are using another ORB implementation class to connect to EAServer, you must export the IDL interface definitions, then use the vendor's IDL compiler to generate stubs. See "Connecting to EAServer with a third-party client ORB" on page 238 for more information.

Stubs for different client models

If you are generating stubs for multiple client models, such as EJB and CORBA, stubs for each model must be generated to a different codebase or Java package. "Specifying Java package mappings for IDL modules" on page 87 describes how to change the Java package for stubs associated with each IDL module.

You can generate stubs in EAServer Manager or by using the command line IDL compiler (see Appendix D, “Using the Command Line IDL Compiler”).

❖ **Generating Java stubs in EAServer Manager**

1 Highlight a component, package, or module as follows:

   a Highlight a component to generate stubs for all interfaces and types required by a component,

   b Highlight a package to generate all stubs needed by components in the package, or

   c Highlight a module to generate stubs for IDL interfaces and types defined within that module.

2 Select File | Generate Stub/Skeleton. The Generate Stubs & Skeletons wizard displays. Follow the instructions on each page to generate Java/CORBA stubs. See the online help for descriptions of any input fields that you do not understand.

Avoiding name collisions with existing Java files

When you are generating Java stubs for a Java component, you must ensure that the generated stubs will not overwrite existing Java classes or interfaces. Name collision occurs if an unscoped IDL interface name matches the name of an existing class in the package to which you are generating stubs. For example, collision would occur if you generate stubs into the com.yourco package when the class com.yourco.Stock exists and the component implements the IDL interface YourCo::Stock. You can avoid name collisions using either of the following strategies:

- When *defining components*, specify an IDL interface name that is different from existing Java class names in the package where you will generate stubs.
- When *generating stubs*, specify a stub package that is different than that which contains the duplicate class.

Compiling stubs

For each IDL interface that is assigned to a component, EAServer Manager 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, EAServer Manager creates the source files listed in the following table:

Table 12-1: IIOP Java stub source files for example component calc

File Name	Purpose
Calc.java	Defines an interface with methods equivalent to the component's methods.
_st_Calc.java	Class that implements the interface.
CalcHelper.java	Contains methods that are required by the ORB and by the application; for example, the ORB calls helper-class methods to read and write object instances to the network.
CalcHolder.java	Used when interface references are passed as an inout or output parameter.

EAServer Manager 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.

EAServer Manager creates stubs in a package subdirectory below the directory specified as the code base in the Generate Stubs & Skeletons dialog. By default, the Java package directory has the same name as the IDL module in which the interface is defined. For example, if the interface is Calculator::Calc, and you specify a code base of `c:\classes`, the stubs will be created in c:*classes**Calculator*.

If a component implements interfaces from more than one module, EAServer Manager creates stubs for each module in separate packages that match each module name. You can specify a single Java package for all stubs as described in "Generating Java stubs in EAServer Manager" on page 210.

If you did not elect to compile the stubs in EAServer Manager, compile the stub classes with a compiler that is compatible with the desired Java version for the stubs. Make sure that the CLASSPATH setting contains the code base directory and the following:

- *%JAGUAR%\java\lib\easserver.jar*
- *%JAGUAR%\java\lib\easclient.jar*
- *%JAGUAR%\java\lib\easj2ee.jar*

Instantiating proxy instances

After you have compiled stub classes, you can implement code that uses the stubs to interact with EAServer components.

Your program must obtain proxy objects for the EAServer component and narrow them to the stub interface that you intend to use by following the steps below:

Step	What it does	Detailed explanation
1	Initialize the CORBA ORB classes.	"Configuring and initializing the ORB runtime" on page 212
2	Use an IOR string and the ORB.string_to_object method to obtain the Manager instance for the server.	"Creating a Manager instance" on page 217
3	Use the Manager instance to create a Session.	"Creating sessions" on page 220
4	Call the Session's lookup method to create proxy objects, then narrow them to an interface that the component supports. The lookup method uses the EAServer name service to resolve the requested name to an installed component.	"Creating stub instances" on page 221
5	Call the stub methods to remotely invoke component methods.	"Executing component methods" on page 223

Java exceptions can occur at any step. "Handling exceptions" on page 228 describes common exceptions and their cause.

You can also instantiate proxies using the CosNaming API, however, the technique described in this section is recommended. See "Instantiating proxies with the CosNaming API" on page 231.

Configuring and initializing the ORB runtime

ORB properties define the class name of the ORB driver that will be used, and configure settings required by the driver. Properties can be set externally in HTML parameters for a Java applet or in command-line arguments for a Java application. You can also set them directly in your source code in both applets and applications. Table 12-2 describes the EAServer ORB properties.

Table 12-2: EAServer Java ORB properties

Property	Specifies
org.omg.CORBA.ORBClass	The class that implements interface org.omg.ORB. Specify com.sybase.CORBA.ORB to indicate the EAServer ORB driver class. There is no default for this property.

Property	Specifies
com.sybase.CORBA. ConnectionTimeout	For applications that run in a cluster, sets a time limit to receive a server response before the connection fails over to try another server in the cluster. Setting this property ensures that failover happens without an unreasonable delay. Specify the timeout period in seconds. The default of 0 indicates no time limit.
com.sybase.CORBA.forceSSL	If set to true when using a a reverse proxy server, forces use of SSL for the connection to the reverse proxy. Set this property to true if the connection to the reverse proxy must use SSL (HTTPS) tunnelling, but the connection from the proxy to the server does not use SSL tunnelling. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information on connecting to EAServer through proxy servers.
com.sybase.CORBA.GCInterval	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.
com.sybase.CORBA.http	Specify whether the ORB should use HTTP tunnelling without trying to use plain IIOP first. The default is false. With the default setting, the ORB tries to open a connection using plain IIOP, and switches to HTTP tunnelling if the plain IIOP connection is refused. The default is appropriate when some users connect through firewalls that require tunnelling and others do not; the same application can serve both types. If you know tunnelling is required, set this property to true. This setting eliminates a slight bit of overhead that is incurred by trying plain IIOP connections before tunnelling is used.
com.sybase.CORBA. HttpExtraHeader	An optional setting to specify what extra information is appended to the header of each HTTP packet when connecting through a Web proxy. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.

Property	Specifies
com.sybase.CORBA.http. jaguar35Compatible	When set to true, specifies that HTTP tunnelling must be compatible with servers running EAServer version 3.5 or older installations. The default is false. **Compatibility with version 3.5 or older servers** The default tunnelling model is incompatible with servers older than version 3.6. If you do not set the com.sybase.CORBA.http.jaguar35Compatible property to true, clients using the EAServer 3.6 or later Java client ORB cannot connect to older-version servers using HTTP tunnelling. Note that HTTP tunnelling may happen automatically when clients connect to the server through firewalls.
com.sybase.CORBA. HttpUsePost	When using HTTP tunnelling, specifies the HTTP request type used. A value of true indicates that POST requests are to be used. A value of false (the default) specifies that GET requests are to be used. Some Web browsers cannot handle the long URLs generated when using HTTP tunnelling with GET requests. Setting this property to true can work around the issue.
com.sybase.CORBA. IdleConnectionTimeout	Specifies the time, in seconds, that a connection is allowed to sit idle. When the timeout expires, the ORB closes the connection. The default is 0, which specifies that connections can never timeout. The connection timeout does not affect the life of proxy instance references; the 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. A finite timeout also allows rebalancing of server load in an application that uses a cluster of servers. If you specify an idle connection timeout, make sure the garbage collection interval (com.sybase.CORBA.GCInterval) is set to an equal or lesser value.
com.sybase.CORBA.isApplet	Specifies whether the client is a Java applet. The default is `false` unless the ORB is initialized by calling the Orb.init method that takes a java.applet.Applet instance as a parameter. If you call another version of init from a Java applet, you must set this property to `true` in order to connect to EAServer using SSL.

Property	Specifies
com.sybase.CORBA.local	For server-side component use only. Specifies whether the ORB reference can be used to issue intercomponent calls in user-spawned threads. The default is `true`, which means that intercomponent calls are made in memory and must be issued from a thread spawned by EAServer. Set this property to `false` if your component makes intercomponent calls from user-spawned threads. **com.sybase.CORBA.local property is deprecated** This property is not needed when calling components from threads spawned by the the Thread Manager. The Thread Manager is the recommended way to spawn threads in Java components. See Chapter 32, "Using the Thread Manager" for more information.
com.sybase.CORBA.ProxyHost	Specifies the machine name or the IP address of a reverse-proxy server. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.CORBA.ProxyPort	Specifies the port number of a reverse-proxy server. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
com.sybase.CORBA.RetryCount	Specify the number of times to retry when the initial attempt to connect to the server fails. The default is 5.
com.sybase.CORBA.RetryDelay	Specify the delay, in milliseconds, between retry attempts when the initial attempt to connect to the server fails. The default is 2000.
com.sybase.CORBA. socketReuseLimit	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. The default may not be appropriate for a long-running client program that calls many methods from servers in a cluster. If sockets are reused indefinitely, the client may build an affinity for servers that it has already connected to rather than randomly distributing its server-side processing load among all the servers in the cluster. In these cases, the property should be tuned to best balance client performance against cluster load distribution. 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.
com.sybase.CORBA. WebProxyHost	The host name or IP address of an HTTP proxy server that supports generic Web tunnelling, sometimes called connect-based tunnelling. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information. There is no default for this property, and you must specify both the host name and port number properties.

Property	Specifies
com.sybase.CORBA. WebProxyPort	When generic Web tunnelling is enabled by setting com.sybase.CORBA.WebProxyHost, this property specifies the port number at which the HTTP proxy server accepts connections. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information. There is no default for this property, and you must specify both the host name and port properties.
com.sybase.CORBA. useJSSE	Use the Java Secure Sockets Extension (JSSE) classes for secure HTTP tunnelled (HTTPS protocol) connections. JSSE provides an alternative to the built-in SSL implementations when secure connections are needed from an applet running in a Web browser. Additional configuration may be required to use this option. See Chapter 5, "Using SSL in Java Clients," in the *EAServer Security Administration and Programming Guide* for more information.

Example: ORB Initialization in an Applet ORB initialization for a Java applet is demonstrated in the example below. This code constructs a java.util.Properties object and sets the required properties. The applet reference and the Properties object are passed to the org.omg.CORBA.ORB.init method.

```
import java.applet.*;
import org.omg.CORBA.*;
public class myApp extends Applet {

  public void init() {
     ...
     java.util.Properties props
         = new java.utils.Properties();
     props.put("org.omg.CORBA.ORBClass",
        "com.sybase.CORBA.ORB");
     ORB orb = ORB.init(this, props);
     ...
```

Rather than property values, you can pass properties to the ORB as parameters in the HTML APPLET tag that loads the applet, as in the example below:

```
<APPLET
codebase=....
<param name="org.omg.CORBA.ORBClass"
  value="com.sybase.CORBA.ORB">
...
</APPLET>
```

A property setting that is passed as an applet parameter supersedes any setting that is specified in the java.utils.Properties parameter to the ORB.init method. If you want to ensure that hard-coded property values are used, pass the Applet parameter as null.

Example: ORB Initialization in an Application ORB initialization for a Java application is demonstrated in the example below. This code constructs a java.util.Properties object and sets the required properties. The command-line parameters are passed to the org.omg.CORBA.ORB.init method.

```
import java.util.*;

public class myApp extends Object {

  public static void main(String[] args)
     throws Exception
{
     ...
     Properties props = new Properties();
     props.put("org.omg.CORBA.ORBClass",
        "com.sybase.CORBA.ORB");
      ORB orb = ORB.init(args, props);
     ...
```

Rather than hard-coding the property values, you can pass them to the ORB as command-line parameters, as in the example below:

```
java yourclass -org.omg.CORBA.ORBClass com.sybase.CORBA.ORB
```

Properties that are specified as command-line parameters supersede values specified in the java.utils.Properties parameter to the ORB.init method. If you want to ensure that hard-coded property values are used, pass the String[ ] parameter to init as null.

Configuring error output The client runtime writes errors to the console by default. In Java applications, you can modify this behavior by creating a logging profile and specifying the profile name in the Java system properties. For more information, see “Using log profiles in Java client applications” in the *EAServer System Administration Guide*.

Creating a Manager instance

The EAServer authentication service implements the SessionManager::Manager interface. When using CORBA naming services, you can resolve this object by using the special name AuthenticationService. Without using naming services, you must supply a CORBA Interoperable Object Reference (IOR), which is a text string that describes how to connect to the server hosting the object.

Standard CORBA IOR strings are hex-encoded and not human-readable. EAServer supports both standard format IORs and a URL form that is human-readable. For information on standard-format IORs, see "Instantiating components using a third-party ORB" on page 239.

URL format IORs The URL string format offers the benefits of being human-readable. Also, for Java applets, you can create URL strings that connect to the applet's download host by default; this feature simplifies deployment since you do not need to change hard-coded IORs when you move your application to another server. IOR strings in URL format must have the form:

```
protocol://host:iiop_port
```

where

- *protocol* is `iiops` if connecting to a secure port and `iiop` otherwise.
- *host* is the EAServer host address or machine name. In an applet, you can omit the host name to specify that the connection must go to the host from which the applet was downloaded.
- *iiop_port* is the port number for IIOP requests. Your server may accept IIOP connections at several different ports, each of which uses a diffferent security profile. For example, the default server configuration provides listeners at these ports:
  - 9000 accepts unsecure IIOP connections.
  - 9001 accepts IIOPS connections with encryption and server-side authentication.
  - 9002 accepts IIOPS connections with encryption and mutual (client and server) authentication. Mutual authentication requires that your end users have valid digital certificates, and that those certificates are issued by a certificate authority that is trusted by the server.

  The *EAServer Security Administration and Programming Guide* describes how to configure listeners and security profiles.

An example URL-format IOR is `iiop://machina:9000`, which specifies that the server runs on the machine named "machina" and listens for IIOP requests on port 9000. In an applet, you can omit the host name to specify that the connection must go to the host from which the applet was downloaded. For example, `iiop://:9000`specifies a connection to port 9000 on the applet's host.

Standard format IORs Use the standard IOR format if you must have portability to other standard Java ORB implementations. Your server generates IOR strings embedded within text files each time it starts. Several files are generated for each IIOP listener. There are files formatted as an HTML param tag; these can be used to compose HTML applet sections. There are also files that contain the IOR by itself. Additionally, there are different files generated for compatibility with different IIOP protocol versions.

For each listener, the server prints a hex-encoded IOR string with standard encoding to the following files in the EAServer *html* subdirectory:

- *<listener><iiop-version>.ior* – Contains the IOR string by itself, followed by a newline.
- *<listener>_<iiop-version>_param.ior* – Contains the IOR as part of an HTML param definition that can be inserted into an applet section.

where

<listener> is the name of the listener.

<iiop-version> is the version of IIOP and can be either 10 (which represents IIOP version 1.0) or 11 (which represents IIOP version 1.1). Use the file that matches the IIOP version that is supported by your client ORB.

For example, a server will generate the following files for a listener named iiops2. All files are created in the *html* subdirectory:

- *iiops2_10.ior*
- *iiops2_11.ior*
- *iiops2_10_param.ior*
- *iiops2_11_param.ior*

Your applet can retrieve the IOR if you supply it in applet parameters. In this case, you can copy the contents of one of the param format files to the HTML file. Alternatively, you can add code that connects to EAServer via HTTP and downloads one of the generated *.ior* files.

Note If you change a server's host name or port number, you must edit or replace IOR values that contain the host name, including hex-format IORs copied from the server-generated *.ior* files. When using the EAServer ORB, use the URL string format and omit the host name. When using another vendor's ORB, you can download the contents of a generated .ior file, or you can store server IORs in the ORB vendor's name server.

Creating the Manager instance Once the applet or application has obtained the server's IOR string or an equivalent IIOP URL string, it calls the ORB.string_to_object method to convert the IOR string into a SessionManager::Manager instance, as shown in the following example:

```
import org.omg.CORBA.*;
import java.awt.*;
import SessionManager.*;

public class myApplet extends Applet {
  String ior;
  ORB orb;
  ... deleted ORB.init() code and code that
      retrieves IOR from applet parameters ...
  Manager manager = ManagerHelper.narrow(
                    orb.string_to_object(ior));
```

Creating sessions

The SessionManager.Session interface represents an authenticated session between the client application and EAServer. The Manager.createSession method accepts a user name and password and returns a Session object, as shown in the example below:

```
import org.omg.CORBA.*;
import SessionManager.*;
import java.awt.*;

public class myApplet extends Applet {
  Manager manager;

  ... deleted code that created Manager instance
      ...
  try {
    Session session = manager.createSession(user,
                                            password);
  }
  catch (org.omg.CORBA.COMM_FAILURE cf)
  {
     // The server is likely down or has run
     // out of connections. You can retry the
     // connection if desired.
     ... report the error ...
  }
  catch (org.omg.CORBA.NO_PERMISSION np)
  {
    // Tell the user they are not authorized
    ...
```

```
}
catch (org.omg.CORBA.SystemException se)
{
  // Catch-all clause for any CORBA system
  // exception that was not explicitly caught
  // above. Report the error but don't bother
  // retrying.
  ...
}
```

Creating stub instances

A Java stub implements the Java version for one of the EAServer component's IDL interfaces. Call the Session.lookup method to obtain a factory for stub instances. The signature of Session.lookup is:

```
SessionManager.Factory lookup(String name)
```

Session.lookup takes a string that specifies the name of the component to instantiate. A component's default name is the EAServer package name and the component name, separated by a slash as in *calculator/calc*. However, a different name can be specified with the component's com.sybase.jaguar.component.naming property. For example, you can specify a logical name, such as *USA/MyCompany/FinanceServer/Payroll*. For more information on configuring the naming service, see Chapter 5, "Naming Services," in the *EAServer System Administration Guide*.

Session.lookup returns a factory for component proxies. Call the Factory.create method to obtain proxies for the component. This method returns a org.omg.CORBA.Object reference. You must call the narrow method in the IDL interface's generated helper class to convert this to an instance of the stub class for the component's IDL interface. If the component instance does not implement the requested interface, the narrow method returns a null object reference.

Session.lookup can throw these CORBA standard exceptions:

- **NO_PERMISSION** The user is not authorized to instantiate the requested component.
- **OBJECT_NOT_EXIST** The server component cannot be instantiated. Verify that:
  - The specified component is installed in the specified EAServer Manager package.
  - The specified EAServer Manager package is installed in the server.
  - The Java class, Windows DLL, or UNIX shared library that implements the component is available.

- If you are instantiating a Java component, the component's skeleton class is available.

The code to call Session.lookup and Factory.create looks like this:

```
import org.omg.CORBA.*;
import SessionManager.*;
import java.awt.*;
import Calculator.*;  // Package for Java stubs
                      // for this example, matches
                      // IDL module name for the
                      // component's interface.

public class myApplet extends Applet {

  Session session;

  ... deleted code that created Session instance
      ...

  //
  // In this example, the component is named calc
  // and is installed in the EAServer package
  // calculator. calcHelper.narrow() verifies that
  // the returned object is of the appropriate
  // type, then returns a Calculator.Calc instance
  //
  try {
    Factory fact =
      FactoryHelper.narrow(
        session.lookup("calculator/calc"));
    Calc c =
      CalcHelper.narrow(fact.create());
  }
  catch (org.omg.CORBA.OBJECT_NOT_EXIST one)
  {
    // Tell the user to contact the server
    // administrator
    ... report the error ...
  }
  catch (org.omg.CORBA.NO_PERMISSION np)
  {
    // Tell the user they are not authorized
    ... report the error ...
  }
  catch (org.omg.CORBA.SystemException se)
  {
```

```
    // Catch-all clause for any CORBA system
    // exception that was not explicitly caught
    // above.
    ... report the error ...
  }
```

Calling Session.lookup in server code

When called from server code, Session.lookup resolves the component name by calling the name service, which gives preference to a local component instance if the component is installed on the same server. However, the use of a locally installed component is not guaranteed. To ensure that a local implementation is used, specify the name as `local:`***`package`***`/`***`component`***, where *package* is the package name and *component* is the component name, for example, `local:CtsSecurity/SessionInfo`. When you specify the local: prefix, the lookup call bypasses the name service and returns a local instance if the component is installed in the same server. The call fails if the specified component is not installed in the same server..

Executing component methods

After instantiating the stub class, use the stub class instance to invoke the component's methods. Each method in the stub interface corresponds to a method in the component interface that you have narrowed the proxy object to.

Parameter and return datatypes

The following table lists the datatypes displayed in EAServer Manager, the equivalent CORBA IDL types, and the Java datatypes used in stub methods.

Table 12-3: EAServer Manager, CORBA IDL, and Java datatype equivalence

EAServer Manager display datatype	CORBA IDL type	IDL Java type (input parameter or return value)	IDL Java type (inout or out parameter)
integer<16>	short	short	org.omg.CORBA.ShortHolder
integer<32>	long	int	org.omg.CORBA.IntHolder
integer<64>	long long	long	org.omg.CORBA.LongHolder
float	float	float	org.omg.CORBA.FloatHolder
double	double	double	org.omg.CORBA.DoubleHolder
boolean	boolean	boolean	org.omg.CORBA.BooleanHolder

EAServer Manager display datatype	CORBA IDL type	IDL Java type (input parameter or return value)	IDL Java type (inout or out parameter)
string	string	java.lang.String	org.omg.CORBA.StringHolder
binary	BCD::Binary	byte[ ]	BCD.BinaryHolder
decimal	BCD::Decimal	BCD.Decimal	BCD.DecimalHolder
money	BCD::Money	BCD.Money	BCD.MoneyHolder
date	MJD::Date	MJD.Date	MJD.DateHolder
time	MJD::Time	MJD.Time	MJD.TimeHolder
timestamp	MJD::Timestamp	MJD.Timestamp	MJD.TimestampHolder
ResultSet	TabularResults::ResultSet	TabularResults.ResultSet	TabularResults.ResultSetHolder
ResultSets	TabularResults::ResultSets	TabularResults.ResultSet[ ]	TabularResults.ResultSetsHolder

Note Null parameter values are not supported for input or inout parameters. Use an output parameter instead. For input parameters that extend java.lang.Object, you must pass an initialized object of the indicated type. When using holder objects to pass inout parameters, you must set the holder object's *value* field to a valid object reference or use the holder constructor that takes an initial value.

Binary, fixed-point, date/time, and ResultSet types The BCD and MJD IDL modules define types to represent common database column types such as binary data, fixed-point numeric data, dates, and times. The BCD::Binary CORBA type maps to a Java byte array. The other BCD and MJD types map to data representations that are optimized for network transport.

To convert between the IDL-mapped datatypes and from core java.* classes, use these classes from the com.sybase.CORBA.jdbc11 package:

Class	Description
SQL	Contains methods to convert from BCD.* and MJD.* types to java.* types
IDL	Contains methods to convert from java.* types to BCD.* and MJD.* types

Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference* provides reference pages for these classes.

ResultSet types The TabularResults IDL module defines types used to represent tabular data. Result sets are typically used only as return types, though you can pass them as parameters. "Methods that return tabular results" on page 225 describes how to process result sets returned by method calls.

User-defined IDL types A user-defined type is any type that is not in the set of predefined datatypes and is not one of the CORBA IDL base types. You can define methods with user-defined types in EAServer Manager, as described in "User-defined IDL datatypes" on page 96.

If a method definition includes user-defined types, the stub method will use the equivalent Java datatype as specified by the CORBA Java language mappings specification. See "Overview" on page 207 for more information on this document.

CORBA Any and TypeCode support
EAServer's Java ORB supports the CORBA Any and TypeCode datatypes. Refer to the OMG CORBA 2.3 specification and *IDL to Java Language Mapping Specification* (formal/99-07-53) for information on using these types.

Holder classes All Java types have an accompanying holder class that is used for passing parameters by reference. Each holder class has the following structure:

```
public class <Type>Holder {
    // Current value
    public <type> value;
    // Default constructor
    public <Type>Holder() {}
    // Constructor that sets initial value
    public <Type>Holder(<type> v) {
      this.value = v;
    }
}
```

This structure is defined by the CORBA Java-language bindings specification.

Note For inout parameters, you must pass a non-null value for the parameter input value. Otherwise, method calls fail and throw an exception (NullPointerException). Use out parameters in the method definition if you do not care about the parameter's input value.

Methods that return tabular results

In EAServer Manager, a method's property sheet indicates whether the method returns zero, one, or multiple result sets. This setting determines the return code of the stub method as follows:

- A method that returns *0 result sets* returns void.
- A method that returns *a single result set* returns TabularResults.ResultSet.

- A method that returns *multiple result sets* returns an array of TabularResults.ResultSet.

The TabularResults IDL module defines the TabularResults::ResultSet CORBA IDL datatype, which maps to TabularResults.ResultSet in Java. Most applications will convert objects of this type to a java.sql.ResultSet by calling one of the following methods:

- **com.sybase.CORBA.jdbc11.SQL.getResultSet (TabularResults.ResultSet)** Accepts a TabularResults.ResultSet parameter and returns a java.sql.ResultSet object.
- **com.sybase.CORBA.jdbc102.SQL.getResultSet (TabularResults.ResultSet)** Accepts a TabularResults.ResultSet parameter and returns a jdbc.sql.ResultSet object. If your client application will run in a JDK 1.0.2 virtual machine, this method must be used instead of the previous method.

After converting the result set to java.sql.ResultSet, use standard JDBC calls to retrieve the rows and columns. Alternatively, pass the result set to a data-aware control that displays the data to the end user.

The example below calls a stub method returnsRows() that returns a single result set:

```
import com.sybase.CORBA.jdbc11.SQL;
...
    java.sql.ResultSet rs =
       SQL.getResultSet(myStub.returnsRows());
    ... code to process rows or pass result set
        to a data-aware control ...
```

The example below calls a stub method returnsResults() which returns multiple result sets:

```
import com.sybase.CORBA.jdbc11.SQL;
...
  java.sql.ResultSet rs;
  TabularResults.ResultSet[] trs_array =
      myStub.returnsResults();

  for (int i = 0; i < trs_array.length; i++)
  {
    rs = SQL.getResultSet(trs_array[i]);
    ... code to process rows or pass result set
        to a data-aware control ...
  }
```

Example method calls

See the EAServer tutorials and the examples provided with your EAServer software for example method calls. Chapter 2, "Creating CORBA Java Components and Clients," in the *EAServer Cookbook* contains tutorials with sample Java clients.

An introductory sample Java client is provided in the *html/classes/Sample/Intro* subdirectory. The file *readme.html* in that directory describes how to compile the classes, install the required component, and run the sample client.

Cleaning up client resources

The garbage collector thread of the Java virtual machine will clean up resources allocated in your client applet or application. No action is required on your part. However, when proxy object references are no longer needed, you can set them to null to expedite cleanup by the garbage collector.

The action of the client program has no direct effect on the cleanup of server-side resources. Server-side cleanup happens when the component is deactivated or destroyed. See "Component lifecycles" on page 13 for more information.

Serializing component instance references

You can call the ORB.object_to_string() and ORB.string_to_object() methods to serialize and deserialize proxy object references. Assuming that the proxy interface is Payroll, this call serializes a proxy component reference:

```
    Payroll payroll;
... deleted code that instantiates payroll ...

    String payroll_ior = orb.object_to_string(payroll);
```

This call deserializes the reference:

```
    Payroll payroll = PayrollHelper.narrow(
                      orb.string_to_object(payroll_ior));
```

The following restrictions apply when serializing and deserializing component proxy references:

- Unless the proxy is for an Enterprise Java EntityBean, the serialized reference remains valid only as long as the server has not been restarted since the time when proxy was first instantiated. When deserializing, the proxy instance will connect back to the same host and port as was used to create the original instance. An EntityBean proxy can be deserialized at any time, as long as the EntityBean is still installed on the original server.
- If the original proxy instance was created by connecting to a secure port with a client-side SSL certificate, the proxy must be deserialized in a session that connects using the same client certificate and equal or greater security constraints. For example, if you create an object with session that uses 128-bit SSL encryption, serialize the object, then later try to deserialize the object using during a session that uses 40-bit SSL encryption, the ORB will throw the CORBA::NO_PERMISSION exception. Access will be allowed when objects created using less secure session are later accessed using a more secure session.

Handling exceptions

The client-side ORB throws two kinds of exceptions:

- CORBA system exceptions – these exceptions are defined in the CORBA specification.
- User-defined exceptions – these exceptions are defined in the component's IDL definition.

CORBA system exceptions

The CORBA specification defines the list of standard system exceptions. In Java, all CORBA system exceptions extend org.omg.CORBA.SystemException. System exceptions are unchecked exceptions (they extend java.lang.RuntimeException). The Java compiler does not require that you catch CORBA system exceptions. However, some exceptions can occur in a well-behaved program. For example, the Session.loookup call throws a NO_PERMISSION exception when you request a component instance and the user lacks permission to instantiate that component. You may want to trap the exceptions shown in the code fragment below:

```
try
{
    // invoke method(s)
```

```
    ...
}
catch (org.omg.CORBA.COMM_FAILURE cf)
{
    // If this occurs when instantiating a Manager
    // instance, the server is likely down or has run
    // out of connections. You can retry the connection
    // if desired.
    //
    // If this occurs after a method call, you
    // can retry the call (or the transaction call
    // sequence for a stateful component).
    ...
}
catch (org.omg.CORBA.TRANSACTION_ROLLEDBACK tr)
{
    // A component on the server aborted the EAServer
    // transaction, or the transaction timed out.
    // Retry the method call(s) if desired.
    ...
}
catch (org.omg.CORBA.OBJECT_NOT_EXIST one)
{
    // Possibly try to create another instance. Check
    // that the package and component are installed
    // on the server.
    // Received when trying to instantiate a component
    // that does not exist. Also received when invoking
    // a method if the object reference has expired
    // (this can happen if the component is stateful
    // and is configured with a finite Instance Timeout
    // property). Create another instance if desired.
    ...
}
catch (org.omg.CORBA.NO_PERMISSSION np)
{
    // Tell the user they are not authorized
    ...
}}
catch (org.omg.CORBA.SystemException se)
{
    // Catch-all clause for any CORBA system exception
    // that was not explicitly caught above.
    // Report the error but don't bother retrying.
    ...
```

Note Not all of the possible system exceptions are shown in the example. See CORBA/IIOP 2.3 Specification for a list of all the possible exceptions.

User-defined exceptions

User-defined exceptions are defined in the component's IDL definition. For example, you might define OverdrawnException to be thrown by methods that withdraw money from a bank account. In Java, all user-defined exceptions extend org.omg.CORBA.UserException.

In Java, IDL user-defined exceptions are checked exceptions; if the IDL definition of a method contains a raises clause, the equivalent Java stub method will have a throws clause that lists the equivalent Java exceptions. For example, consider the IDL definition below:

```
module MyModule {
  exception MyException
  {
    string reason;
  };

  interface MyIntf {
    boolean throwException
    ( in boolean yes_no )
    raises (MyException);
  };
};
```

The equivalent Java throwException method is:

```
boolean throwException (boolean yes_no)
   throws MyModule.MyException;
```

Deploying and running Java clients

Run the Java client in a JDK 1.2 or later Java interpreter. If running applets, make sure your browser supports JDK 1.2. Most browsers require Sun's Java Plug-in to support JDK 1.2.

At run time, the following EAServer JAR files must be in the CLASSPATH for Java applications and included with the class files for applets:

- *java/lib/easclient.jar*
- *java/lib/easj2ee.jar*

Unlike earlier versions, EAServer 4.0 does not provide runtime class files in the *html/classes* directory. To run applets, you must include the JAR files in the applet's ARCHIVE tag, or expand these JAR files to the *html/classes* directory.

Chapter 2, "Creating CORBA Java Components and Clients," in the *EAServer Cookbook* provides a Java-CORBA tutorial that describes how to deploy Java client applications and applets.

Instantiating proxies with the CosNaming API

EAServer allows you to use the CosNaming API to instantiate proxies in your client applications. This technique is not recommended, because:

- It requires use of deprecated SessionManager::Factory methods.
- When run in standalone clients, the CosNaming classes are incompatible with 1.2 or later JDK classes. You can use the CosNaming API in server components running in a server that uses JDK 1.2 or a later JDK version.

You do not need to use the CosNaming API in clients to realize the benefits incurred by using logical component names. When you use the technique described in "Instantiating proxy instances" on page 212, EAServer uses the CosNaming API to resolve component names in the implementation of the Session.lookup and Session.create methods.

The steps for resolving objects with CosNaming are as follows:

Step	What it does	Detailed explanation
1	Configure ORB properties, including the ORB runtime driver class and the EAServer naming server URL, then initialize the ORB runtime.	"Configuring and initializing the ORB runtime" on page 212
2	Instantiate the CORBA CosNaming naming service and obtain the initial naming context.	"Obtaining an initial naming context" on page 233
3	Resolve component names to proxy objects and narrow them to the stub interface.	"Instantiating proxy objects for EAServer components" on page 236

Initializing the ORB

Before you can call any other ORB methods, you must configure ORB properties and call the org.omg.CORBA.ORB.init method. “Configuring and initializing the ORB runtime” on page 212 describes how to do this. In addtion, you must set the the com.sybase.CORBA.NameServiceURL property.

com.sybase.CORBA.NameServiceURL specifies the list of URLs with the host and port number for IIOP connectivity to the EAServer name servers for your application. Each URL takes the the form:

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

where

- *protocol* is `iiop` or `iiops`. Use `iiops` if connecting to a secure IIOP port, and `iiop` otherwise.
- *hostname* is the host machine name for the server that serves as the name server for your application. If omitted, the ORB uses a default host name. In Java applets, the default host name is the applet’s download host. In Java applications, the default is `localhost`.
- *iiop-port* is the IIOP port number for the server.
- *initial-context* is the initial naming context. This can be used to set a default prefix for name resolution. For example, if you specify `USA/Sybase/`, all names that you resolve with the context are assumed to be relative to this location in the name hierarchy. When specifying the initial context, the trailing slash is optional; it is added automatically if you do not specify an initial context that ends with a slash.

If your application uses a cluster of servers, the cluster may use multiple name servers. In this case, specify the URLs for each name server in a list separated by semicolons and no white space. Include the cluster’s initial naming context only with the last URL. For example:

```
iiop://host1:9000;iiop://host2:9000/USA/Sybase/
```

If you do not set the com.sybase.CORBA.NameServiceURL, property, the default is assumed. Different defaults are used depending whether your client is a Java application or a Java applet. The applet default is:

```
iiop://download-host:9000/
```

which indicates that the EAServer ORB expects the name server to be available at port 9000 on the host from which the applet was downloaded, and that the initial naming context is the root context (/).

The default for applications is:

```
iiop://localhost:9000/
```

Obtaining an initial naming context

After initializing the ORB, call the ORB.resolve_initial_references method to obtain the initial naming context. The naming context is an object that implements the CosNaming::NamingContext IDL interface; it is used to resolve EAServer component and service names to server-side objects.

Obtaining the initial context The example below shows how the initial naming context is retrieved:

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
public class myApplet extends Applet {

  ... deleted ORB initialization code ...
  NamingContext nc = null;
  org.omg.CORBA.Object objRef = null;
  try {
    objRef = orb.resolve_initial_references(
              "NameService");
    nc = NamingContextHelper.narrow(objRef);
  } catch (org.omg.CORBA.ORBPackage.InvalidName ine) {
    nc = null;
  }
  if (nc == null) {
    System.out.println("Error: Could not "
        + "instantiate CORBA naming context.");
    return;
  }
```

Introduction to CosNaming name resolution The initial NamingContext will have the name context that was specified in the com.sybase.CORBA.NameServiceURL ORB initialization property. Your client program invokes the NamingContext::resolve operation to obtain an instance of the EAServer authentication service as well as component instances.

Note EAServer's CosNaming implementation currently lacks support for the BindingIterator interface, which is used to browse the name hierarchy.

The NamingContext::resolve operation takes a CosNaming::Name parameter, which is a sequence of CosNaming::NameComponent structures. The Java definitions of these types and the NamingContext::resolve operation follow:

```
package org.omg.CosNaming;

class NameComponent {
```

```
  public String id;    // Represents a node in a name
  public String kind;  // Unused, can contain comment
info

  // Construct a NameComponent instance with the
  // specified initial values for id and kind fields
  public NameComponent(String id, String kind);
}

interface NamingContext {
  ... other methods not shown ...
  public org.omg.CORBA.Object resolve
   (NameComponent[] n)
   throws
   org.omg.CosNaming.NamingContextPackage.NotFound,
   org.omg.CosNaming.NamingContextPackage.CannotProcee
d,
   org.omg.CosNaming.NamingContextPackage.InvalidName;
}
```

In Java, a name is represented by an array of NameComponent instances, with the *id* field of each instance set to a node of the name. For example, the name

```
USA/Sybase/Jaguar/TestPackage/TestComponent
```

can be represented by the array *theName* which is created in this code fragment:

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
public class myApplet extends Applet {

  NamingContext nc;
  ... deleted code that retrieves initial NamingContext
...

  NameComponent theName[] = {
    new NameComponent("USA", ""),
    new NameComponent("Sybase", ""),
    new NameComponent("Jaguar", ""),
    new NameComponent("TestPackage", ""),
    new NameComponent("TestComponent", "")
  } ;
```

To simplify your source code, the EAServer naming service allows you to specify multiple nodes of a name in one NameComponent instance, using a forward slash (/) to separate nodes. The name from the example above can be represented in a one-element array as shown below:

```
    NameComponent theName[] = {
      new NameComponent(
        "USA/Sybase/Jaguar/TestPackage/TestComponent",
""")
      };
```

NamingContext::resolve resolves a name to an object; this method either returns an org.omg.CORBA.Object instance or throws one of the exceptions described below:

- NotFound indicates that the name is not bound to an object, the name does not exist, or some node in the indicated hierarchy does not exist; the why field contains an enumeration that encodes the reason why the name was not found.
- InvalidName indicates that the name is malformed.
- CannotProceed or a CORBA SystemException indicates that an error has occurred. “Handling exceptions” on page 228 describes CORBA system exceptions.

The code fragment below illustrates a typical call with exception handling:

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
public class myApplet extends Applet {

  try {
    NamingContext nc;
    ... deleted code that retrieves initial
NamingContext ...

    NameComponent theName[] = {
      new NameComponent(
        "USA/Sybase/Jaguar/TestPackage/TestComponent",
""));

    org.omg.CORBA.Object obj = nc.resolve(theName);

    ... deleted code that narrows the object to a
supported          interface ...

  } catch (NotFoundException nfe) {
```

```
    ... report the error ...
  } catch (InvalidName ine ) {
    ... report the error ...
  } catch (CannotProceed cpe) {
    ... report the error ...
  }
```

Instantiating proxy objects for EAServer components

Proxy objects are instantiated as follows:

1 Create a NameComponent array that names the component. Component names are composed as follows:

   server-context/*package*/*component*

   where

   - *server-context* is the root naming context for the server where the component is installed. You can view and edit this setting in the Naming Services tab of the Server Properties window. The default for a new server is "/". If you specified an initial name context when initializing the ORB properties, then resolved names are assumed to be relative to the initial name context. For example, if your client program specifies an initial context of */USA/Sybase*, and your server's root context is *USA/Sybase/Engineering*, then you can resolve component names as *Engineering/package/component*.
   - *package* is the EAServer package name in which the component is installed, as displayed in EAServer Manager.
   - *component* is the component name, as displayed in EAServer Manager.

2 Call the NamingContext.resolve method. It returns a factory object for the component. You can use the factory to create proxy objects.

3 Narrow the CORBA Object reference to a SessionManager::Factory instance.

4 Call the factory's create method and narrow the return value by calling the narrow method in the generated helper class for the interface. The create method requires a username and password to authenticate the end user.

The example below instantiates a component *MyComponent*, installed in package *MyPackage*, hosted on a server with initial context *USA/Sybase/Jaguar*. The username and password are *Guest* and *GuestPassword*, respectively. The component implements the IDL interface MyPackage::MyInterface, and the code narrows the proxy object to that interface.

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.CosNamingPackage.*;
import SessionManager.*;

public class myApplet extends Applet {

  NamingContext nc;

  ... deleted code that created initial naming context
       ...

  // Create a NameComponent array for
  // USA/Sybase/Jaguar/MyPackage/MyComponent
  //
  NameComponent compName[] = {
    new NameComponent("USA", ""),
    new NameComponent("Sybase", ""),
    new NameComponent("Jaguar", ""),
    new NameComponent("MyPackage", ""),
    new NameComponent("MyComponent", "")

  try {
    // Resolve the name to obtain the proxy object
    org.omg.CORBA.Object obj = nc.resolve(compName);

    // Narrow to a factory instance
    Factory compFactory = FactoryHelper.narrow(obj);

    // Get the proxy object and narrow it to
MyInterface.
    obj = compFactory.create("Guest", "GuestPassword");
    MyPackage.MyInterface comp =
        MyPackage.MyInterfaceHelper.narrow(obj);
  }
  catch (NotFoundException nfe) {
     ... report the error ...
  }
  catch (CannotProceed cpe) {
     ... report the error ...
  }
  catch (InvalidName ine) {
     ... report the error ...
  }
}
```

Using other CORBA ORB implementations

EAServer's IIOP implementation allows you to use any CORBA-compliant client ORB to invoke EAServer components. You can also use the EAServer client ORB to execute components that are hosted by another vendor's server ORB.

Connecting to EAServer with a third-party client ORB

In some cases, you may wish to use another vendor's ORB in your client applications. For example, you may have an existing installation of the ORB on client workstations.

Clients that use another ORB can use the same code as for the EAServer ORB, except for the following differences:

- You must use stub classes generated by the vendor's IDL-to-Java compiler rather than stubs generated by EAServer Manager.
- Your code to connect to EAServer and instantiate components may differ.

When executing methods, you may wish to use the EAServer conversion classes to create and interpret the predefined EAServer datatypes (see "Binary, fixed-point, date/time, and ResultSet types" on page 224). These conversion classes, in packages com.sybase.CORBA.jdbc102 and com.sybase.CORBA.jdbc11, are documented in Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference*. The classes are compatible with any Java ORB.

Generating compatible Java stubs

You should generate stubs for your third-party ORB using the IDL-to-Java or IDL-to-C++ compiler provided by the vendor. Stubs created by EAServer Manager are not guaranteed to work with another ORB.

Each component's IDL interfaces are specified in the Component Properties window, under the General tab. See "Configuring component properties" on page 52 for more information. All interfaces are defined in IDL modules that are stored as plain text files in the EAServer *Repository* subdirectory. For example, if the component implements the Module1::I1 and Module2::I2 interfaces, you will need to copy the files *Module1.idl* and *Module2.idl* into a working directory for generating stubs for your third-party ORB software. You must also copy any files that are included by these modules, including those listed in "Predefined EAServer IDL files" on page 239.

As an alternative to copying files, you can open modules in the EAServer IDL editor and use File | Save As to save them to your working directory. See "Creating and editing IDL modules, interfaces, and types" on page 86 for more information.

Predefined EAServer IDL files lists the names of the predefined EAServer IDL modules that are needed by all client applications.

Predefined EAServer IDL files

Filename	Description
SessionManager.idl	Defines interfaces for session-based creation of EAServer component instances.
BCD.idl	Defines the CORBA datatypes for EAServer's binary and fixed-point numeric datatypes.
MJD.idl	Defines the CORBA datatypes for EAServer's date and time datatypes.
TabularResults.idl	Defines the CORBA datatypes that represent result sets returned by a method invocation.

Warning! When creating stubs for another ORB, do not overwrite the EAServer Java stubs in the EAServer *html/classes* subdirectory. Use different package names when creating stubs for third-party ORBs or create the 3rd-party ORB stubs under a different code base.

Instantiating components using a third-party ORB

EAServer's naming service cannot be used with other client ORBs, so you must use the EAServer SessionManager::Manager interface to instantiate components from another ORB, as described in "Instantiating proxy instances" on page 212. Set the org.omg.CORBA.ORBClass property to the name of the class provided by your ORB vendor.

Also, you must use standard format IORs, not the URL format, as described in "Standard format IORs" on page 219.

To simplify applet deployment, you can use one of the following techniques to avoid coding IORs into deployed HTML or Java class files:

- Code your applets to open an HTTP connection to the server, then retrieve the contents of the server-generated *.ior* file that contains the IOR. (See "Standard format IORs" on page 219 for more information on the generated *.ior* files.)
- If your third-party ORB provides a name service, store the IOR for EAServer in the third-party name service.

Connecting to third-party ORBs using the EAServer client ORB

You can use the EAServer client-side ORB to execute components hosted by another vendor's server-side ORB, as long as the server-side ORB accepts IIOP connections and the required interfaces are defined in standard CORBA IDL.

❖ **Implement your client as follows:**

1 Import all the required IDL modules into EAServer Manager, as described in "Importing existing IDL modules" on page 100.

2 Generate stubs for each imported module using EAServer Manager, as described in "Generating Java stubs in EAServer Manager" on page 210. You must generate stubs for each module individually.

3 Implement code to connect to the third-party server and instantiate components, following the vendor' s documentation.

PART 4

CORBA-C++ Components and Clients

This part explains how to build C++ components and clients that use standard CORBA type mappings and run-time services.

CHAPTER 13

CORBA C++ Overview

This chapter provides an overview of things to consider when developing CORBA C++ clients and components for EAServer.

Overview

CORBA is a distributed component architecture defined by the Object Management Group (OMG). EAServer supports the CORBA Internet Inter-ORB Protocol (IIOP). EAServer also provides a CORBA-compatible C++ client-side interface. These two items allow you to create CORBA EAServer C++ applications. C++ components and clients are also interoperable with clients and components using other technologies.

The dynamic invocation interface (DII) is not supported.

For information on the CORBA architecture, see the specifications available at the OMG Web site at http://www.omg.org.

Requirements

To develop C++ components, you need a C++ development tool. All software that is required to run C++ components in EAServer is supplied with the EAServer product.

To develop C++ clients, you need a C++ development tool. To deploy and run C++ clients on end-user workstations, you must install the EAServer C++ client runtime on each workstation.

For detailed system requirements, see the *EAServer Installation Guide* for your platform.

Supported datatypes

EAServer follows the OMG standard for translating CORBA IDL to C++, more specifically, refer to *C++ Language Mapping Specification* (formal/99-07-41). You can download this document from the OMG Web site at http://www.omg.org.

The standard supports all the C++ features in the *Annotated C++ Reference Manual* by Ellis and Stroustrup as implemented by the ANSI/ISO C++ standardization committees. In addition, the namespace construct is supported. Templates are not required but can be used.

IDL modules are mapped to C++ namespaces and IDL interfaces are mapped to C++ classes. All OMG IDL constructs scoped to an interface are accessed through C++-scoped-names. For example, the IDL interface CtsComponents::ThreadManager maps to the C++ class CtsComponents::ThreadManager. If your C++ compiler supports namespaces, you can use the namespace directive and refer to the interface name by itself, as in:

```
using namespace CtsComponents;
...
  ThreadManager threadMan;
```

Mapping for predefined EAServer Manager datatypes

Table 13-1 lists the datatypes in EAServer Manager, the equivalent CORBA IDL types, and the C++ datatypes used in stub methods. You can also define additional types in IDL; when you generate stubs and skeletons, these are translated to C++ types using the standard CORBA IDL to C++ type mappings. For example, The BCD and MJD CORBA IDL modules define types to represent binary data, fixed-point numeric data, dates, and times. For details, see the generated Interface Repository documentation for these IDL modules.

Table 13-1: EAServer Manager, CORBA IDL, and C++ datatype mappings

EAServer Manager	CORBA IDL type	Argument mode	IDL C++ type
integer<16>	short	in inout out return	CORBA::Short CORBA::Short& CORBA::Short_out CORBA::Short
integer<32>	long	in inout out return	CORBA::Long CORBA::Long& CORBA::Long_out CORBA::Long
integer<64>	long long	in inout out return	CORBA::LongLong CORBA::LongLong& CORBA::LongLong_out CORBA::LongLong **Define JAG_LONGLONG** Because there is no standard C++ type for an signed 64-bit integer, you must define the JAG_LONGLONG macro as your compiler's type for a signed 64-bit integer.
float	float	in inout out return	CORBA::Float CORBA::Float& CORBA::Float_out CORBA::Float
double	double	in inout out return	CORBA::Double CORBA::Double& CORBA::Double_out CORBA::Double
boolean	boolean	in inout out return	CORBA::Boolean CORBA::Boolean& CORBA::Boolean_out CORBA::Boolean
string	string	in inout out return	char* char*& CORBA::String_out char*

EAServer Manager	CORBA IDL type	Argument mode	IDL C++ type
binary	BCD::Binary	in inout out return	BCD::Binary& BCD::Binary& BCD::Binary_out BCD::Binary*
decimal	BCD::Decimal	in inout out return	BCD::Decimal& BCD::Decimal& BCD::Decimal_out BCD::Decimal*
money	BCD::Money	in inout out return	BCD::Money& BCD::Money& BCD::Money_out BCD::Money*
date	MJD::Date	in inout out return	MJD::Date& MJD::Date& MJD::Date_out MJD::Date
time	MJD::Time	in inout out return	MJD::Time& MJD::Time& MJD::Time_out MJD::Time
timestamp	MJD::Timestamp	in inout out return	MJD::Timestamp& MJD::Timestamp& MJD::Timestamp_out MJD::Timestamp
ResultSet	TabularResults:: ResultSet	return	TabularResults::ResultSet*
ResultSets	TabularResults:: ResultSets	return	TabularResults::ResultSets*

Using mapped IDL types

All EAServer component interfaces are defined in standard CORBA IDL, and C++ stubs and skeletons use the standard CORBA IDL-to-C++ type mappings.

For local variables that map to constructed C++ types and do not represent an IDL interface, use the C++ datatype that is appended with _var. _var variables are automatically freed when they are out of scope. If you do not use the _var type, references must be freed with the C++ delete operator. In Table 13-1, string, binary, decimal, money, date, time, timestamp, ResultSet, and ResultSets have _var types. Other types listed in Table 13-1 map to fixed-length C++ types. For fixed-length types, use the base C++ type.

IDL interfaces map to C++ classes that extend the CORBA::Object class. These object reference types have a _var form for references with automatic memory management, and a _ptr form for references that must remain valid after the reference variable goes out of scope. _ptr references must be freed by calling CORBA::release.

You must pass values in a _var type as follows:

```
MyType_var v;
....
v.in()              // Passes v as an in
                      // parameter.
v.inout()           // Passes v as an inout
                      // parameter.
v.out()             // Passes v as an out
                      // parameter.
return v._retn()    // Passes v as a return value.
```

Note Do not use the C++ _out types for local variables; these types are reserved for method signatures.

For out and inout parameters of IDL type string, use CORBA::string_alloc or CORBA::string_dup to allocate memory for them. For example:

```
ItemName = CORBA::string_dup("Dummy Item Name");
ItemData = CORBA::string_dup("Dummy Item Data");
```

In C++, if you declare string variables as type CORBA::String_var, memory allocated by CORBA::string_dup or CORBA::string_alloc is freed automatically. Otherwise, declare as char * and free the memory explicitly by calling CORBA::string_free.

You can pass a null value as a parameter type only with the object reference type Module::Interface::_nil().

Overloaded methods

Overloading methods is supported for C++ components. When you overload a method, you use the same name for several methods that specify different parameters. When you call an overloaded method, the method with the corresponding parameters is executed. See “Operation declarations” on page 93 for more information.

CHAPTER 14

Creating CORBA C++ Components

This chapter describes how to code CORBA EAServer C++ components.

Procedure for creating C++ components

This section contains an overview of the steps involved in creating C++ components; the remainder of this chapter includes detailed information for each step. You use EAServer Manager to define basic information (such as the component name and methods) about a C++ component, and generate files that are required to write the component's class implementation and to compile the class into a dynamic link library (on Windows) or shared library (on UNIX).

You write your component as a C++ class; the generated files include a class implementation template in which you can write your method logic. In addition, EAServer supplies an application programming interface that contains classes and methods that you can use to perform EAServer-specific tasks. You can use the EAServer API to write code to handle errors, cache connections to third-tier database servers, return result sets, manage transactions, share data between instances of the same component, retrieve a client's SSL certificate information, and make intercomponent calls.

After writing the method logic in the class implementation template, you compile the component to build a dynamic link library (DLL) or shared library, then deploy the librar to your EAServer installation.

Detailed information for creating components is in these sections:

1 Defining C++ components – use EAServer Manager to specify the component's name, DLL name, C++ class, method prototypes, and how transactions and instances are managed. This information is used to automatically generate the files necessary to compile the C++ component (including source files, makefiles and a Microsoft Visual C++ module definition file) into a DLL or shared library.

2 Generating required C++ files – use EAServer Manager to generated the source files and the makefiles for UNIX and Windows.

3 Writing the class implementation – in the class implementation template, write the logic for each method.

4 Compiling source files – Compile and link source files to create a DLL or shared library.

5 Installing the Component DLL or Shared Library – copy the DLL or shared library to the *cpplib* directory of the EAServer installation.

Defining C++ components

To define a C++ component, use EAServer Manager to create an Interface Definition Language (IDL) module and interface, assign the interface to the component, define the properties for the component, and then define the methods in the component. Define each method's return type and parameters. For each parameter, define its datatype and argument mode.

Chapter 4, "Defining Components" describes how to define and configure new components in EAServer Manager.

Transaction property

The component's transaction property determines how it participates in transactions. You can view and change this property using the Transactions tab of the component's property sheet. For a description of each option on the Transactions tab, see "Component properties: Transactions" on page 58. A transaction consists of a number of database updates (which can be performed by multiple components) that are grouped into a single atomic unit of work.

For a full description of how EAServer handles transactions, see Chapter 2, “Understanding Transactions and Component Lifecycles.”

Instance properties

The threading property imposes constraints on the concurrent execution of the component in different threads. You can view and change these properties using the Threading tab of the component’s property sheet.

In single threading, multiple instances can exist simultaneously, but only one can be active at any one time. EAServer synchronizes instantiations, method invocations, and the destruction of all instances. Use single threading if your component shares volatile global data or stateful resources between instances.

This setting determines the constraints that are placed on the concurrent execution of different instances of the component. The following settings specify the constraints that are placed on concurrent execution of different instances of the component. The choices are:

- *Concurrency* – Multiple invocations can be processed concurrently; that is, multiple instances can be simultaneously active on different threads. The component must be thread-safe. Use this setting if the component code uses no volatile global data and does not maintain data in resources (such as files) that are shared among instances. For example, you could not use this setting if every component instance opened the same file and wrote text to it. An example of volatile global data could be a counter that is stored in a global variable. This threading model offers the highest performance.

  You can use the EAServer shared properties feature in your component code to share data safely among instances of a multiple-threaded component. See “Share data between C or C++ components” on page 688 for more information.

- *Bind Thread* – Instances are bound to the creating thread. The component uses thread-local storage.

  This setting determines whether a component instance is always invoked in the same thread or can be invoked on any thread. By default, this check box is not selected, which indicates that EAServer can invoke the component’s methods with any thread.

Use the default setting unless your component uses thread-local storage. EAServer provides no APIs for thread-local storage, but you can issue thread system calls from the C++ component code. Do not use thread-local storage if you are implementing new components. Instead, use instance variables to associate data with a specific component. If you incorporate existing code that uses thread-local storage into a C++ component, select this setting.

- *Pooling* – Instances are pooled after a commit or rollback.
- *Sharing* – A single shared instance services all client requests. This model offers the worst performance. Use this model only if the logic in your code requires that only one component instance exist at one time. Attempts to create new instances when one already exists will fail.

Client interfaces

You define methods by specifying each method's return type and parameters. For each parameter, you define its datatype and argument mode. You use a method's property sheet to define its return type and parameters.

Chapter 5, "Defining Component Interfaces" describes how to define IDL methods in the component interface. "Supported datatypes" on page 244 describes the IDL to C++ type mappings.

Instead of defining methods using EAServer Manager, you can code a Java interface that defines your component's methods and import it into EAServer Manager. See "Importing interfaces from compiled Java files" on page 81 for more information.

Control interface

Optionally configure a control interface for the component. Using a control interface allows you to implement methods to respond to changes in the instance lifecycle. See "Configuring a control interface" on page 72 for more information.

Generating required C++ files

You use EAServer Manager to generate the C++ files that you need to compile into a DLL or UNIX shared library as well as a class implementation template in which to write method logic. These C++ files include:

- Method skeletons file – Contains method routines that read the parameters from the network and call the method. The method skeletons also send the return status and output parameter data back to the client.

- Class header file – Contains the method declarations only. This file is an included file in the method skeletons file and the class implementation template.
- Class implementation template – Contains the class, method, and parameter declarations, as well as empty method definitions. You enter any business logic into the empty method definitions.
- Stub interface files – Contain the interface definition for all components in a package, as well as definitions for user-defined types and exceptions used in your component's interface. EAServer Manager creates these files when you generate C++ stubs for your component.
- UNIX makefile – You use a makefile to compile the C++ source files into a UNIX shared library.
- Windows makefile and Microsoft Visual C++ module definition file – You use the makefile and a module definition file to compile the C++ source files into a DLL.

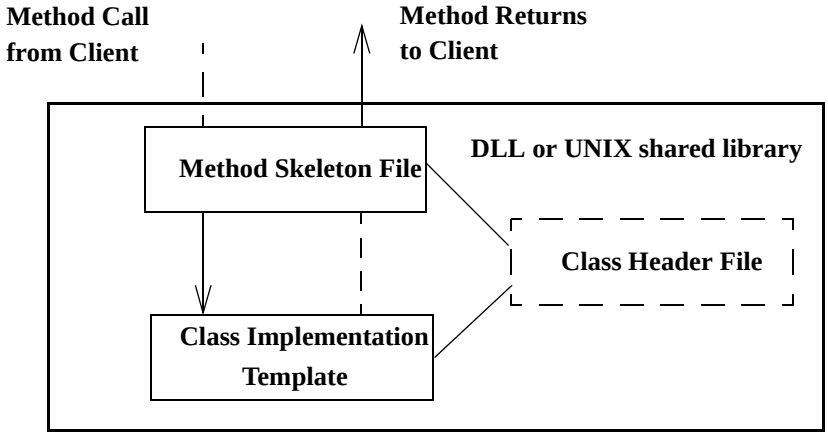
"Method call to a C++ component DLL or UNIX shared library" on page 253 shows the flow of a client method call to a C++ component DLL or UNIX shared library.

1 The client invokes a method using the proxy or stub appropriate to the type of client. The stub or proxy sends the invocation information over the network to the server.
2 The method skeleton in the method skeletons file unmarshals the call and makes a call to the method implementation in the class implementation template.
3 After the method executes, the method implementation returns the call to the method skeleton.
4 The method skeleton marshals the call and sends the call to the client.

Method call to a C++ component DLL or UNIX shared library

The following figure shows the flow of a client method call to a C++ component DLL or UNIX shared library.

Figure 14-1: How C++ component methods are called



❖ Generating required C++ files in EAServer Manager

To generate the required C++ files from a package or component, start EAServer Manager and:

1 Select the component or, if you want to generate files for all components in a package, select the package.

2 Select File | Generate Stub/Skeleton. The Generate Stubs & Skeletons Wizard displays. Follow the instructions on each page to generates C++ stubs and skeletons. See the online help for descriptions of any input fields that you do not understand.

File naming conventions

EAServer Manager generates the following files:

File type	File name
method skeletons file	*package-name_component-name.cpp*
class header file	*class-name.hpp.new*
class implementation template	*class-name.cpp.new*
stub interface file	*package-name.hpp*

where:

component-name is the name of the component that you defined in EAServer Manager.

class-name is the class name that you specified when you created the component.

EAServer Manager creates the directory structure based on the code base that you specify and the component name, as follows:

```
code_base/package_name/component_name
```

where:

code_base is the directory name in the Code Base field in EAServer Manager. If the Code Base field does not contain a full path name, the directory will be located under the EAServer installation directory, relative to the *html/classes* subdirectory.

package-name is the name of the package that contains the component.

component_name is the component name as displayed in EAServer Manager.

Regenerating changed C++ component methods

When you add or delete methods or modify component method prototypes, you must regenerate the method skeletons and class header files. You must manually add, delete, or modify the methods in the class implementation template. Before you regenerate the method skeletons and class header files, make sure that you have moved your modified class implementation template to another directory or renamed it so the generated class implementation template does not overwrite your existing class implementation template.

Writing the class implementation

After you generate the method skeleton file, class header file, and class implementation template, write the code for each method in the class implementation template (you can also write your class implementation from scratch and replace the generated class implementation template).

You must use scoped names to specify the CORBA IDL module, the EAServer SessionManager IDL module, and any component IDL modules that you want to execute methods on. To make using scoped names easier, you can use the C++ using statement for the IDL module namespaces as in the following example:

```
using namespace CORBA;
using namespace SessionManager;
```

If your C++ compiler does not support namespaces, define a compiler macro JAG_NO_NAMESPACE when compiling your source files.

CORBA::is_nil(Object) can be used to verify that a specific interface is implemented by a component.

As with any C++ class, you use the constructor and destructor to initialize and perform any cleanup of objects.

Constructors of class variables in file scope not called
If you declare a class variable in file scope and compile it into a shared object, such as a component, the Solaris C++ compiler doesn't call the constructor of the class variable. If the variables need to be in scope only for a particular function, procedure or module, then declare these variables in the appropriate function, procedure, module; otherwise declare these variables in the class definition.

You can also include EAServer C routines to:

- Cache connections to third-tier database servers
- Return result sets
- Set transaction states
- Share data between C++ components

Coding these C routines is described in "Write methods" on page 256.

Write methods

This section describes how to write methods for EAServer-specific APIs, including C routines, accessing SSL client certificates, and issuing intercomponent calls. A C++ method signature must use the return types and parameter datatypes described in "Supported datatypes" on page 244. To implement any of the features that require EAServer C routines, you must include *jagpublic.h* and implement the methods for each feature as follows:

- Handling Errors

  Use user-defined or CORBA system exceptions to handle errors. See "Error handling" on page 257 for more information about system and user-defined exceptions.

- Caching Connections to Third-Tier Database Servers

  You can use a connection cache to improve performance when connecting to database servers. See "Using Connection Manager routines in C, C++, and ActiveX components" on page 490 for more information.

- Returning Result Sets

  A component method can return row results to the client. See “Returning result sets” on page 257 for more information.

- Managing explicit OTS transactions

  You can explicitly to manage OTS transactions from your component.

- Setting Transaction State

  Methods in a transactional component should call one of the transaction primitive routines to set the transaction state before returning. See “Methods that set transactional state” on page 695 for more information.

- Sharing Data Between C++ Components

  EAServer provides C routines that allow components within the same package to share data with each other. See “Share data between C or C++ components” on page 688 for more information.

Returning result sets

You can return result sets by:

- Using the C API as described in “Sending result sets from a C or C++ component” on page 475. The component method that returns a result set or result sets must return a null pointer in place of the TabularResults::ResultSet or TabularResults::ResultSets pointer. For example:

  ```
  return NULL;
  ```

  See “Sending result sets from a C or C++ component” on page 475 for more information.

- Returning a pointer to an initialized TabularResults::ResultSet or TabularResults::ResultSets object.

Error handling

Handle errors by:

1 Writing detailed error descriptions to the server log file using JagLog.

2 Coding one of these tasks:

a If the component is transactional, call JagDisallowCommit or JagRollbackWork (or you can throw the CORBA::TRANSACTION_ROLLEDBACK exception instead of calling JagRollbackWork).

b Throw a CORBA system or user-defined IDL exception to be raised by the client stub. See “Handling exceptions” on page 292 for more information.

For more information about these methods, see Chapter 5, “C Routines Reference,” in the *EAServer API Reference*.

Managing explicit OTS transactions

You can code components (and clients) to initiate and complete transactions using the OTS (Object Transaction Service) CosTransactions::Current or CosTransactions::TransactionFactory interfaces.

Note In order to use OTS, you must enable EAServer to use the OTS/XA transaction coordinator. See Chapter 3, “Creating and Configuring Servers,” in the *EAServer System Administration Guide* for more information.

To use the functionality of these interfaces, include CosTransactions.hpp in your source file.

To explicitly use transactions in a component or client, use the CosTransactions::Current interface to perform these tasks.

Task	Call this method	Catch these exceptions
Start a transaction.	begin	SubtransactionsUnavailable
Temporarily stop a transaction.	suspend	None
Resume a suspended transaction.	resume	InvalidControl
Commit a transaction.	commit	NoTransaction, HeuristicMixed, HeuristicHazard
Roll back a transaction.	rollback	NoTransaction
Make the only possible outcome of the transaction a rollback.	rollback_only	NoTransaction
Roll back a transaction after a specified amount of time has elapsed without any response.	set_timeout	None
Retrieve a transaction’s status.	get_status	None

Task	Call this method	Catch these exceptions
Retrieve a transaction's name. Use this method when you need to debug transactions.	get_transaction_name	None

Using factories

The TransactionFactory interface is included in EAServer only to maintain compatibility with the CORBA OTS specification—Sybase recommends that you use the CosTransactions::Current interface to create explicit transactions.

Note Sybase recommends that you use suspend with caution so as not to conflict with the EAServer component model. For example, do not use suspend to take control of a transaction that it does not control.

Initializing the ORB

To initialize the ORB and retrieve a reference to the CosTransactions::Current interface, specify the TransactionCurrent *ObjectId*, which identifies the CosTransactions::Current interface, to the resolve_initial_references method, and narrow it (using the _narrow method) to the CosTransactions::Current interface. Use the is_nil method to verify that the reference to the CosTransactions::Current interface is valid.

For clients

The following code fragment shows how to initialize the ORB from a client. ORB_init must take the *argumentList* array that specifies the ORBNameServiceURL parameter. You can also set the ORBNameServiceURL using the JAG_NAMESERVICEURL environment variable.

```
int  argumentCnt = 1;
char *argumentList[] = {
   { "-ORBNameServiceURL iiop://<hostnamehere>:9000" },
   { "" }
   };


try {

  CORBA::ORB_var  orb = CORBA::ORB_init(argumentCnt,
    argumentList, 0);
  cerr << "Orb init" << endl;

  CORBA::Object_var  crntObj =
    orb->resolve_initial_references
      ("TransactionCurrent");
  CosTransactions::Current_var CurrentIntf =
```

```
    CosTransactions::Current::_narrow(crntObj);
  if( CORBA::is_nil(CurrentIntf) )
  {
    cerr << "Error getting Current" << endl;
    exit(-1);
  }
cerr << "Got Current" << endl;
```

For components

The following code fragment shows how to initialize the ORB from a component. ORB_init does not need to take any parameters.

```
orb = CORBA::ORB_init(argumentCnt, NULL, 0);
cerr << "Orb init" << endl;

CORBA::Object_var  crntObj =
  orb->resolve_initial_references
    ("TransactionCurrent");
CurrentIntf =
  CosTransactions::Current::_narrow(crntObj);
if( CORBA::is_nil(CurrentIntf) )
{
  cerr << "Error getting Current" << endl;
  /* could be due to:
  **   1. Component not BeanManaged/OTS Style
  **   2. Already in a Txn
  ** 3. not running under OTS
  */
  return CS_FAIL;
}
cerr << "Got Current" << endl;
```

Calling CosTransactions::Current interface methods

After retrieving a reference to the CosTransactions::Current interface, you can call any of the CosTransactions::Current methods on the CosTransactions::Current reference. After executing the begin method, execute the database operations you want to include in the transaction. Depending on whether the database operations succeed or fail, you can execute other appropriate methods, such as commit, rollback, or rollback_only. This code fragment shows how to begin a transaction and commit or roll it back depending on the return codes received from the databases.

```
CurrentIntf->begin();
ret = JagCmGetConnection( &cache,
    (SQLCHAR *) USERID, (SQLCHAR *) PASSWD,
    (SQLCHAR *) xaresource, (SQLCHAR *) "CTLIB_110",
    (void*) &conn, JAG_CM_UNUSED );
```

```
if (ret != CS_SUCCEED) {
  cerr << "Error getting connection" << endl;
  CurrentInt->rollback();
}

CurrentIntf->commit(CS_FALSE);
```

Executing tasks outside of a transaction

To execute a method outside of a transaction, you can write the code to perform either:

- Execute the method before beginning a transaction, or
- Temporarily stop and start execution of the transaction.

❖ **Execute tasks outside of a transaction using the *suspend* and *resume* methods**

1 Execute suspend to temporarily stop execution of the transaction.

2 Execute the tasks.

3 Execute resume to restart the execution of the transaction from where it stopped.

This code fragment shows how to execute tasks outside of a transaction. The suspend method returns the control context. You specify the control context when you use the resume method to restart the transaction. Catch the InvalidControl exception, which may be raised when a control context is out of scope (and not null).

```
sus_ctrl = CurrentIntf->suspend();

/* The following method is not in the transaction */
component1->method2();

CurrentIntf->resume(sus_ctrl);
/* The following methods are invoked
in the transaction */
    component2->method1();

CurrentIntf->commit(CS_FALSE);

}
catch(CosTransactions::SubtransactionsUnavailable
  &ex )
{
```

```
    cerr << "Exception: SubTxnUnavailable " <<
      ex._jagExceptionCode << endl;
}
catch(CosTransactions::NoTransaction &ex )
{
    cerr << "Exception: NoTransaction  " <<
      ex._jagExceptionCode << endl;
}
catch(CosTransactions::InvalidControl &ex )
{
    cerr << "Exception: InvalidCtrol  " <<
      ex._jagExceptionCode << endl;
}
catch(...)
{
    cerr << "Caught Unexpected exception" << endl;
    exit(-1);
}
```

Exceptions

The CosTransactions module includes these exceptions:

- SubtransactionsUnavailable – raised when the client thread already has an associated transaction and the transaction coordinator does not support nested transactions.
- NoTransaction – raised when there is no transaction associated with the client thread.
- InvalidControl – raised when the specified control is not null and not within the scope of the client thread.
- Inactive – raised when a method such as rollback_only is executed on a transaction has already been prepared.
- InvalidTransaction – raised when a request carries an invalid transaction context, such as if an error occurred when registering a resource.
- TransactionRequired – raised when a request carries a null transaction context but required an active transaction. For example, this could occur when a component specifies the Mandatory attribute.
- Unavailable – raised when the requested object cannot be returned because OTS/XA transaction coordinator restricts the availability of the object.
- TransactionRolledBack – raised when a transaction is marked to roll back or has already been rolled back.

Heuristic exceptions

A heuristic decision is a decision to commit or roll back updates that one or more participants in a transaction make without waiting for the consensus decision from the transaction coordinator. These types of commits and rollbacks are also called heuristic commits and heuristic rollbacks. When a heuristic commit or rollback is made, the transaction can become inconsistent. Therefore, a heuristic commit or rollback is made only in unusual circumstances such as communication failures. When the System Administrator issues a heuristic commit or rollback from EAServer Manager, a heuristic exception is raised.

- HeuristicMixed – Raised when a heuristic decision is made and some relevant updates are committed and others are rolled back.
- HeuristicHazard – Raised when a heuristic decision may have been made, when not all of the conditions of all relevant updates is known, and for those updates whose condition is known, either all of them were committed or rolled back.
- HeuristicRollback – Raised when a heuristic decision to roll back all of a transaction's relevant updates has been made.
- HeuristicCommit – Raised when a heuristic decision to commit all of a transaction's relevant updates has been made.

Accessing SSL client certificates

Clients can connect to a secure IIOP port using an SSL client certificate. You can issue intercomponent calls to the built-in *CtsSecurity/SessionInfo* component to retrieve the client certificate data. See Chapter 6, "Using SSL in C++ Clients," in the *EAServer Security Administration and Programming* guide for more information about retrieving SSL information and issuing intercomponent calls using SSL.

Issuing intercomponent calls

To invoke other components, instantiate a stub for the second component, then use the stub to invoke methods on the component.

You must use a stub to issue intercomponent calls. If you call methods in another C++ component directly, EAServer features such as transactions and security will not work.

To invoke methods in other components, create an ORB instance to obtain object references to other components and invoke methods on the object references. You obtain object references for other components on the same server by invoking string_to_object with the IOR string specified as *Package*/*Component*. For example:

```
CORBA::Object_var obj =
    orb->string_to_object("MyPackage/MyComponent");
MyModule::MyInterface_var i =
   MyModule::MyInterface::_narrow(obj);
```

When making intercomponent calls using string_to_object, the user name of the client that executed the component is automatically used for authorization checking. string_to_object returns an instance running on the same server if the component is locally installed; otherwise, it attempts to resolve a remote instance using the naming server.

To components on a non-EAServer ORB

Your component may need to invoke methods on a component hosted by another vendor's CORBA server-side ORB. Sybase recommends that C++ components use the EAServer client-side ORB for all IIOP connections made from EAServer components. See "Connecting to third-party ORBs using the EAServer client ORB" on page 301 for more information.

Compiling source files

This section describes how to compile and link a component DLL or UNIX shared library that contains EAServer methods. Your code must be built as a DLL or UNIX shared library in order to be installed into the EAServer runtime environment. When you generate source files for your component, EAServer Manager creates an example makefile that builds the component library. You may have to edit this file to match your environment, as described in the following sections:

- "Compiling on UNIX platforms" on page 265
- "Compiling on Windows" on page 266

Compiling on UNIX platforms

EAServer Manager generates a *make.unix* file when you generate the component skeleton as described in “Generating required C++ files” on page 252. To build your shared library, run the following command:

```
make -f make.unix
```

On Solaris, when linking component shared libraries or client binaries, you must link with the EAServer libraries that match your compiler version. Choose the appropriate directory from those listed below:

- *lib* contains libraries that are compatible with the 6.*x* compiler, stripped of symbol information for production use.
- *devlib* contains libraries that are compatible with the 6.*x* compiler, for debugging use.
- *lib_sol4x* contains libraries that are compatible with the 4.*x* compiler, for production use.
- *devlib_sol4x* contains libraries that are compatible with the 4.*x* compiler, for debugging use.

The generated Solaris make files link with 6.*x* libraries by default. To use 4.*x* libraries, edit the definition of the LIB macro in the make file, and change the paths to the library directories. The library and binary format is different between version 6.*x* and version 4.*x* compilers. Use the compiler version that the server is running with. By default, the server runs with version 6.*x* compatibility, but you can override this when starting the server. For more information, see “Starting the server” in the *EAServer System Administration Guide*.

The generated UNIX make file for C++ components works on other platforms without changes. Platform-specific information is defined in the file *make.include.platform*, where *platform* is the name returned by the command:

```
uname -s
```

The *make.include.platform* includes the necessary settignss to run the compiler and linker in the component make file. You may need to edit these settings if your compiler and linker are not installed in the standard location, or you use different software.

If you generate stub and skeleton files at the same time, EAServer Manager automatically adds the location of the component stub files to the makefile. If you move the component source files to another machine, make sure that you copy the stub files as well and specify their location in the makefile. You specify the component stub files location by adding `/I`*stub_location* to the `.ccp.obj` rule in the makefile. *stub_location* is the directory in which the component stub files reside.

After building the shared library, copy it to the *cpplib* directory of your EAServer installation.

Note If you do not place the component shared library in the EAServer *cpplib* subdirectory, the directory containing the shared library must be specified in the shared library search path environment variable for your platform (for example, LD_LIBRARY_PATH for Solaris).

Compiling on Windows

For components that run on Windows, you must build a DLL that contains your C++ component methods. After building the DLL, copy it to the *cpplib* directory of your EAServer installation.

Note If you do not place the component DLL in the EAServer *cpplib* subdirectory, the directory containing the DLL must be specified in the PATH environment variable.

You can use EAServer Manager to generate a makefile and module definition (*.def)* file. See “Generating required C++ files” on page 252 for instructions on generating a makefile and *.def* file with EAServer Manager.

Before compiling your C++ component using nmake with the generated makefile, verify that the makefile can find the directory containing the ODBC header files and libraries. You must set the ODBCHOME environment variable to the directory containing the ODBC header files and libraries. If you have Microsoft Visual C++ and ODBCHOME is not set, the makefile looks in *C:\msdev* (which is the default installation directory for Microsoft Visual C++) for these files.

If you generate stub and skeleton files at the same time, EAServer Manager automatically adds the location of the component stub files to the makefile. If you move the component source files to another machine, make sure that you copy the stub files as well and specify their location in the makefile. You specify the component stub files location by adding `/I`*stub_location* to the `.ccp.obj` rule in the makefile. *stub_location* is the directory in which the component stub files reside.

To build your DLL, run this command from a command window in your component's source directory:

```
nmake -f make.nt
```

If you make changes to the makefile, rename it so it won't be overwritten when you regenerate the required files.

Visual C++

Visual C++ requires a module definition file that specifies which functions are exported from a DLL and some options that control how the DLL is loaded into memory. Module definition files end with the extension *.def*.

For most projects, you can use the generated module definition file as is. In some cases, you may want to edit settings other than those in the EXPORTS section. For example, your component may perform better with a smaller or larger HEAPSIZE setting.

Note Do not edit the generated function names in the EXPORTS section of the *.def* file for a C++ component. If you do, the EAServer dispatcher will not be able to call your methods.

Debugging C++ components

To debug a component you must run the debug version of the server, and use a debugger running on the same host as EAServer. Chapter 3, "Creating and Configuring Servers," in the *EAServer System Administration Guide* describes how to start the debug server.

To debug a component from Microsoft Visual C++, you must set the component's com.sybase.jaguar.component.cpp.debug property under the Advanced tab to `true`.

Follow these steps to attach to the server and step into your component code:

1 Change to the *bin* subdirectory in your EAServer installation, and start the debugger with the executable.

On Solaris:

a Edit the *user_setenv.sh* file, and set the WORKSHOP_DIR environment variable to the location of the Workshop debugger; for example:

```
WORKSHOP_DIR=/OPT/SUNWspro6.2/
export WORKSHOP_DIR
```

b On a command line, enter:

```
serverstart.sh -servername ServerName -workshop
```

On Windows:

a Edit *user_setenv.bat*, and set the VC variable to the Visual C++ installation, where *vcvars32.bat* is located in *vc_path\bin*; for example:

```
set vc=c:\vc_path
```

b On a command line, for Visual C++ version 5 or 6 compilers, enter:

```
serverstart.bat -servername ServerName -msdev
```

For Visual C++ version 7 compilers, enter:

```
serverstart.bat -servername ServerName -devenv
```

ServerName is the name of the server. If you are using the preconfigured server rather than one that you created yourself, use “Jaguar”.

2 Set a breakpoint on the function jag_dbg_stop. This function executes every time the server loads a component DLL. The jag_dbg_stop prototype is:

```
void jag_dbg_stop(char *compName)
```

The *compName* parameter specifies the name of the library or shared library that was just started. Several components may be started before yours. In the debugger, display the *compName* value when the jag_dbg_stop breakpoint is tripped, and monitor the value to determine when your component is started. Breakpoints on jag_dbg_stop are triggered before the server calls the component's create method.

Note Make sure the jag_dbg_stop breakpoint is set before running your client application.

3 When your component's DLL is started, you can specify the component's C++ function names as breakpoints and step into the method's code when it is invoked.

Running C++ components externally

EAServer's C++ component model allows you incorporate legacy C and C++ business logic code into a component. However, if legacy code is unstable, it can cause the server to crash.

Beginning in version 4.0, you can configure C++ components to execute within a dedicated external process. EAServer spawns a subprocess to execute the component, and issues component invocations using interprocess communication.

Limitations

Because external components execute in a different process than the host server, they cannot use the following features:

- **Sharing, Concurrency, or Bind Thread properties** The Sharing, Concurrency, and Bind Thread component properties have no effect when components execute externally, because each component instance runs in a separate process. You can get a form of instance reuse by enabling the Pooling property. With Pooling enabled, the server reuses component processes for multiple invocations.

- **Transactions or connection caches** Server managed transactions and connection caching are not supported in components that execute externally.
- **C and C++ API routines** None of the Jag* C routines or server-side C++ classes documented in Chapter 5, “C Routines Reference,” in the *EAServer API Reference* are available to components that execute externally. These routines and classes can only be called by code that executes within the host server process.

Input, output, and logging
You cannot read from standard input in C++ components (whether they execute in-process or in an external process). C++ components that execute externally cannot call the JagLog C routine, but any text written to standard output is recorded in the server log file.

- **Stateful components** Components that execute externally must be stateless, and no control interface methods are called on the component implementation class. The Auto Demarcation/Deactivation property must be enabled for components that execute externally.

Configuring a component to run externally

To run your C++ component externally, configure the following component properties:

- **General / C++ Executable** Specifies the name of the executable that the server launches as a subprocess. Specify a plain filename, with no path information or platform extensions such as *.exe* for Windows. The executable must exist in the EAServer *cpplib* subdirectory. When you generate a component skeleton, EAServer Manager generates a makefile to build the executable.
- **Resources / Maximum Wait** Specifies the maximum time, in seconds, that the server waits for method execution to complete. A value of 0 indicates infinity, which is the default. If the method does not complete in time, the server returns a CORBA::NO_RESOURCE_EXCEPTION to the caller.

- **Resources / Maximum Active Instances** Specifies the maximum number of external component processes that run simultaneously. A value of 0 indicates no limit, which is the default. There is one process per component instance, and one component instance per client session. When the limit has been reached, client requests for new instances block until an existing instance is destroyed. The maximum blocking time is limited by the Maximum Wait setting.

Building and deploying the external component executable

Before you can build an external component executable, you must generate a skeleton. The skeleton for an external component is different than for a component that runs in-process, so regenerate skeletons if you have changed the component properties to run externally. “Generating required C++ files” on page 252 describes how to generate the C++ code.

The executable indicated by the component properties General / C++ Executable must be deployed in the EAServer *cpplib* directory, as well as the library specified by the DLL Name field. The generated Makefile builds the library and executable and copies both to the *cpplib* directory when you run the “all” make target.

Creating C++ components for multiplatform clusters

If you run C++ components in multiplatform clusters, you must configure the additional settings described here.

To deploy C++ components in a multiplatform cluster, specify `${JAGUAR_PLATFORM}` in the component library name, and do not include the platform-specific file extension such as *.dll* or *.so*. EAServer replaces this macro with the platform identifier when loading the component. This feature allows you to deploy libraries for multiple platforms in the same directory.

If running the component externally, specify `${JAGUAR_PLATFORM}` in the C++ Executable name.

In EAServer Manager, the Use Platform Independent Library Naming option on the General Tab in the Component Properties dialog box strips the library extension from the library name and appends `${JAGUAR_PLATFORM}` to the existing name.

CHAPTER 15

Creating CORBA C++ Clients

This chapter describes how to code a CORBA-compatible EAServer C++ client application.

For information about establishing secure C++ client sessions, see Chapter 6, “Using SSL in C++ Clients,” in the *EAServer Security Administration and Programming* guide.

Procedure for creating CORBA C++ clients

To create a CORBA C++ client, you write and compile a C++ program that establishes a connection and session with the EAServer ORB, that instantiates a proxy object for the component, and that calls methods in the proxy object. You use EAServer Manager to define the component methods and generate stubs for the components. When the client calls the methods in the proxy objects, the proxy object methods communicate across the network and execute the corresponding methods in the components.

To create CORBA EAServer C++ clients:

1 Use EAServer Manager to generate stubs (C++ header files). See “Generating stubs” on page 274.

2 Write the C++ source files and include the stubs you created with EAServer Manager. See “Writing CORBA C++ clients” on page 275.

3 Compile the C++ source files. See “Compiling C++ clients” on page 293 for EAServer-specific requirements for compiling EAServer C++ clients. To learn how to compile your C++ client into an executable in your development environment, see the development environment’s documentation.

Generating stubs

The EAServer ORB implementation class requires stub header files in order to invoke component methods. You generate the stub header files with EAServer Manager and include them in your client source files. The stub header files contain as inline all the component functions, which make calls to the C functions in *libjcc.dll*. Inline functions allow EAServer to support multiple C++ compilers without having to include separate link libraries for each compiler.

If you are using another ORB implementation class to connect to EAServer, you must export IDL and use the vendor’s IDL compiler to generate stubs that are compatible with that ORB implementation. “Using CORBA ORB implementations other than EAServer” on page 299 describes how to export IDL files for EAServer components.

❖ Generating stubs in EAServer Manager

You can generate stub header files from EAServer Manager as follows:

1 Highlight a component, package, or module as follows:

- Highlight a component to generate stubs for all interfaces and types required by a component,
- Highlight a package to generate all stubs needed by components in the package, or
- Highlight a module to generate stubs for IDL interfaces and types defined within that module.

2 Select File | Generate Stub/Skeleton. The Generate Stubs & Skeletons wizard displays. Follow the instructions on each page to generate C++ stubs. See the online help for descriptions of any input fields that you do not understand.

Writing CORBA C++ clients

These section describes how to code a CORBA C++ client that invokes component methods:

Adding required include and namespace declarations

Stub header files are generated for all IDL modules that include interfaces that the component implements—you must include all these stub header files. In addition to the stub header files, you must also include *SessionManager.hpp* (which contains the classes and functions that allow a C++ client to create and destroy sessions) in the client source file.

You can also include these optional header files:

- *TabularResults.hpp* – contains the classes and functions that allow C++ clients to receive result sets from components.
- *CosNaming.hpp* – contains the classes and functions that allow C++ clients to use the EAServer's name service feature to bind a component to a name that must be unique within a naming context.
- *BCD.hpp* – contains the mappings for binary and arbitrary precision floating point-decimal datatypes.
- *MJD.hpp* – contains the datatype mappings from CORBA to C++ for Modified Astronomical Julian Date (M.J.D.) dates and times.

Note *TabularResults.hpp* already includes *BCD.hpp* and *MJD.hpp*; if you include *TabularResults.hpp*, you do not have to include *BCD.hpp* and *MJD.hpp*.

You must use scoped names to the CORBA IDL module, the EAServer SessionManager IDL module, and any component IDL modules that you want to execute methods on. To make using scoped names easier, you can use the C++ using statement for the IDL module namespaces as in the following example:

```
using namespace CORBA;
using namespace SessionManager;
```

If your C++ compiler does not support namespaces, define the compiler macro JAG_NO_NAMESPACE when compiling your source files.

When you create an object, identify the object reference by appending *_var* to the object name. The *ObjectName_var* reference will be automatically released when it is deallocated or assigned a new object reference.

CORBA::is_nil(Object) can be used to verify that a specific interface is implemented by a component. For an example, see “Creating a Manager instance” on page 280.

If you are returning result sets from components, you should also specify the TabularResults EAServer IDL module with the using statement.

Instantiating stub instances

Before invoking methods on component instances, the client must connect to a server and instantiate the components. Your code must perform these steps to create proxy instances:

Step	What it does	Detailed explanation
1	Initialize the CORBA ORB and create an ORB reference.	“Configure and initialize the ORB runtime” on page 277
2	Use the ORB reference to create a Manager instance.	“Creating a Manager instance” on page 280
3	Use the Manager instance to create a Session.	“Creating sessions” on page 282
4	Use the Session instance to create stub component instances.	“Creating stub instances” on page 282
5	Call the stub methods to remotely invoke component methods.	“Invoking methods” on page 283

Note Except for the example in “Processing result sets” on page 284, the same client source code is used as an example throughout this section. Only the parts relevant to each step are used.

Configure and initialize the ORB runtime

Before you can use any ORB classes, you must call the ORB_init method, which:

- Returns an object reference to the ORB.
- Allows you to pass initialization parameters to the driver class in the form of a string array. You can also set an environment variable (in the System Properties for your machine) for each initialization parameter. If the environment variable and initialization parameter are set, the value of the initialization parameter is used. You can set any initialization parameter to a value of *none*, which overrides the value of the environment variable and sets the value to the default, if any.

You can pass the following initialization parameters to the driver class:

- ORBHttp – this specifies whether the ORB should use HTTP-tunnelling to connect to the server. A setting of of "true" specifies HTTP tunnelling. The default is "false". This parameter can also be set in an environment variable, JAG_HTTP. Some firewalls may not allow IIOP packets through, but most all allow HTTP packets through. When connecting through such firewalls, set this property to "true".
- ORBHttpExtraHeader – An optional setting to specify what extra information is appended to the header of each HTTP packet when connecting through a Web proxy. See Chapter 11, “Deploying Applications Around Proxies and Firewalls,” in the *EAServer Security Administration and Programming Guide* for more information.
- ORBHttpUsePost – when using HTTP tunnelling, specifies the HTTP request type used. A value of true indicates that POST requests are to be used. A value of false (the default) specifies that GET requests are to be used. This parameter can also be set in an environment variable, JAG_HTTPUSEPOST.
- ORBLogIIOP – this specifies whether the ORB should log IIOP protocol trace information. A setting of "true" enables logging. The default is "false". This parameter can also be set in an environment variable, JAG_LOGIIOP. When this parameter is enabled, you must set the ORBLogFile option (or the corresponding environment variable) to specify the file where protocol log information is written.
- ORBLogFile – this sets the path and name of the file to which to log client execution status and error messages. This parameter can also be set in an environment variable, JAG_LOGFILE. The default setting is *no log*.

- ORBCodeSet – this sets the code set that the client uses. This parameter can also be set in an environment variable, JAG_CODESET. The default setting is *iso_1*.
- ORBRetryCount – specify the number of times to retry when the initial attempt to connect to the server fails. This parameter can also be set in an environment variable, JAG_RETRYCOUNT. The default is 5.
- ORBRetryDelay – specify the delay, in milliseconds, between retry attempts when the initial attempt to connect to the server fails.This parameter can also be set in an environment variable, JAG_RETRYDELAY. The default is 2000.
- ORBProxyHost – specifies the machine name or the IP address of an reverse proxy server. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
- ORBProxyPort – specifies the port number of a reverse proxy server.
- ORBforceSSL – force an SSL connection to a reverse proxy server (indicated by the ORBProxyHost and ORBProxyPort properties). Set this property to true if the connection to the reverse proxy must use SSL (HTTPS) tunnelling, but the connection from the proxy to the server does not use SSL tunnelling.
- ORBsocketReuseLimit – specifies 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. The default may not be appropriate for a long-running client program that calls many methods from servers in a cluster. If sockets are reused indefinitely, the client may build an affinity for servers that it has already connected to rather than randomly distributing its server-side processing load among all the servers in the cluster. In these cases, the property should be tuned to best balance client performance against cluster load distribution. In Sybase testing, a setting of 10 to 30 proved to be a good starting point. If the reuse limit is too low, client performance degrades.

- ORBIdleConnectionTimeout – 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. A finite timeout also allows rebalancing of server load in an application that uses a cluster of servers.
- ORBWebProxyHost – the host name or IP address of an HTTP proxy server that supports generic Web tunnelling, sometimes called connect-based tunnelling. There is no default for this property, and you must specify both the host name and port number properties. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information. You can also specify the property by setting the environment variable JAG_WEBPROXYHOST.
- ORBWebProxyPort – when generic Web tunnelling is enabled by setting `ORBWebProxyHost`, this property specifies the port number at which the HTTP proxy server accepts connections. There is no default for this property, and you must specify both a host name and port. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information. You can also specify the property by setting the environment variable JAG_WEBPROXYPORT.
- ORBHttpExtraHeader – an optional setting to specify what extra information is appended to the header of each HTTP packet sent to a proxy server (specified with the `ORBWebProxyHost` parameter). You can also specify the property by setting the property JAG_HTTPEXTRAHEADER. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.

You can pass additional properties to configure secure (IIOPS) connections. See Chapter 6, "Using SSL in C++ Clients," in the *EAServer Security Administration and Programming* guide for more information.

Example: ORB initialization

ORB initialization is demonstrated in this example. You can specify the ORB options as a command line parameters to be passed to the ORB_init method.

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <SessionManager.hpp>
#include <CosNaming.hpp>
#include <Jaguar.hpp>
#include <Tutorial.hpp>    // Stubs for interfaces in
Tutorial IDL
                           // module.

int main(int argc, char** argv)
{
const char *usage =
  "Usage:\n\tarith -ORBNameServiceURL iiop://
  <host>:<iiop-port>/<initial-context>\n";
const char *tutorial_help =
  "Check EAServer Manager and verify that the"
  "Tutorial/CPPArithmetic component exists "
  "and that it implements the "
  "Tutorial::CPPArithmetic IDL interface.";

const char *ior_prefix = "iiop://";
const char *component_name = "Tutorial/CPPArithmetic";
char *ior = NULL;

try {

cout << "Creating Jaguar session\n\n";

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

Creating a Manager instance

The SessionManager::Manager interface is used for client authentication for EAServer connections. To create a Manager instance, you must identify the server by using:

- The Interoperable Object Reference (IOR) for the server, or
- The URL for the server.

The IOR string encodes the server's host address and the port at which the server accepts IIOP requests. Each time EAServer is started, for each listener the server prints a hex-encoded IOR string with standard encoding to the following files in the EAServer *html* subdirectory:

- *<listener><iiop-version>.ior* – Contains the IOR string by itself.
- *<listener>_<iiop-version>_param.ior* – Contains the IOR as part of an HTML PARAM definition that can be inserted into an APPLET tag.

<listener> is the name of the listener.

<iiop-version> is the version of IIOP and can be either 10, which represents IIOP version 1.0, or 11, which represents IIOP version 1.1.

For example, a server will generate the following files for a listener, iiops2:

- *iiops2_10.ior*
- *iiops2_11.ior*
- *iiops2_10_param.ior*
- *iiops2_11_param.ior*

You can code your C++ client to retrieve the IOR string from one of the *<listener><iiop-version>.ior* files.

The server's IIOP port is configured in EAServer Manager using listeners. In the default configuration, the IIOP port number is 9000.

Once the client has obtained the server's IOR or URL string, it calls the ORB::string_to_object method to convert the IOR or URL string into a Manager instance, as shown in the following example. You use the Manager::_narrow method to return a new object reference for the existing object, which is the IOR object.

```
  ...
  Object_var object = orb->string_to_object
("iiop://myhost:9000");
  Manager_var manager = Manager::_narrow (object);
  if (is_nil(manager)) {
    cout << "Error: Null SessionManager::Manager
instance. Exiting.";
    return -1;
  }...
```

string_to_object returns an object reference to the URL, `iiop://jagpc3:9000`, as object. For each reference, the _var form is used because the object will be automatically released when it is deallocated or assigned a new object reference. _narrow converts *object* into object reference for Manager.

_narrow returns a nil object reference if the component does not implement the interface. is_nil(manager) verifies that the SessionManager::Manager interface is implemented and returns an error if the interface is not implemented.

Creating sessions

The SessionManager::Session interface represents an authenticated session between the client application and a server. The Manager::createSession method accepts a user name and password and returns a Session_var object, session, as shown in the example below:

```
...
Session_var session =
  manager.createSession("jagadmin", "");
...
```

Creating stub instances

You call the Session::lookup method to return a factory for proxy object references. The signature of Session::lookup is:

```
SessionManager::Factory_var lookup("name")
```

Session::lookup takes a string that specifies the name of the component to instantiate. A component's default name is the EAServer package name and the component name, separated by a slash as in *calculator/calc*. However, a different name can be specified with the component's `com.sybase.jaguar.component.naming` property. For example, you can specify a logical name, such as *USA/MyCompany/FinanceServer/Payroll*. For more information on configuring the naming service, see Chapter 5, "Naming Services," in the *EAServer System Administration Guide*.

Session::lookup returns a factory for component proxies. Call the Factory::create method to obtain proxies for the component. This method returns a org.omg.CORBA.Object reference. Call _narrow to convert the object reference into an instance of the stub class for the component.

The code to call Session::factory and Factory::create looks like this:

```
...
```

```
  // In this example, the component is named
  // Repository and is installed in
  // the EAServer package.

Object_var obj = session->lookup("Jaguar/Repository");
SessionManager::Factory_var repoFactory =
SessionManager::Factory::_narrow(obj);

obj = repoFactory->create();
Jaguar::Repository_var repository =
  Jaguar::Repository::_narrow(obj);

// Verify that we really have an instance.
if (CORBA::is_nil(repository))
{
  cout << "ERROR: Null instance for component.";
}
```

Calling Session.lookup in server code

When called from server code, Session::lookup resolves the component name by calling the name service, which gives preference to a local component instance if the component is installed on the same server. However, the use of a locally installed component is not guaranteed. To ensure that a local implementation is used, specify the name as `local:`***`package`***`/`***`component`***, where *package* is the package name and *component* is the component name, for example, `local:CtsSecurity/SessionInfo`. When you specify the local: prefix, the lookup call bypasses the name service and returns a local instance if the component is installed in the same server. The call fails if the specified component is not installed in the same server..

Invoking methods

After instantiating the stub class, use the stub class instance to invoke the component's methods. The stub class has methods that correspond to each method in the component. Parameter datatypes are mapped as described in Table 13-1 on page 245. Any parameter datatype can be used as a return type; in addition, user-defined IDL datatypes can be used as return, in, inout, or out parameters.

You can overload methods in C++ and Java, but not in ActiveX components. See "Operation declarations" on page 93 and "Supported datatypes" on page 244.

In addition to the tasks described in this section, you can also explicitly manage OTS transactions from your client. See “Managing explicit OTS transactions” on page 258 for more information.

Processing result sets

To retrieve and process a single result set from a component:

1 Call the component method on the stub instance that returns a result set.

2 Iterate through each row and then each column in a row by using nested for loops.

3 Use the discriminator method (_d) to retrieve the datatype of the column in a row and switch/case syntax to process the column values (such as printing the column values).

To retrieve and process multiple result sets returned from a component method as a TabularResults::ResultSets object:

1 Call the component method on the component reference that returns the result sets.

2 Retrieve the length or number of result sets.

3 Iterate through the result sets using a for loop.

   For each result set, iterate through each row and then each column in a row by using nested for loops.

   You can treat a ResultSets object as an array of ResultSet objects. On each iteration, retrieve a reference to each ResultSet object by using the subscript [ ] operator.

4 Use the discriminator method (_d) to retrieve the datatype of the column in a row and switch/case syntax to process the column values (such as printing the column values).

Example of processing result sets

This example retrieves a single result set. The following code shows the C++ client in its entirety. For detailed explanations, see the sections that explain each result-set processing step.

All of the required header files are included. The IDL module namespaces are specified with the C++ using statement. The printResultSet() method contains the logic for processing a result set. main() contains the logic to initialize and connect to the EAServer ORB, instantiate the stub, call the component method to retrieve the result set object, and call printResultSet() to process the result set.

After the result set has been processed, execution of printResultSet() ends and control is returned to main(). In main(), the screen is kept open with the fprintf statement. Once you press Return, execution ends.

```
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include <SessionManager.hpp>
#include <TabularResults.hpp>
#include <Test.hpp>
using namespace CORBA;
using namespace SessionManager;
using namespace TabularResults;
using namespace Test;
void printResultSet(const ResultSet& rs)
{
  ULong nc = rs.columns.length();
  cout << rs.rows << " rows, " << nc << " columns" << endl;
  for (ULong row = 0; row < rs.rows; row++)
  {
    cout << "row " << row << ": ";
    for (ULong column = 0; column < nc; column++)
    {
      if (column > 0)
      {
        cout << ", ";
      }
      BooleanSeq& nulls = ((ColumnSeq&)rs.columns)[column].nulls;
      if (row + 1 <= nulls.length() && nulls[row])
      {
        cout << "null";
        continue;
      }
      Data& values = ((ColumnSeq&)rs.columns)[column].values;
      switch (values._d())
      {
         case TYPE_BIT:
         {
            BooleanSeq& booleanValues = values.booleanValues();
            cout << (booleanValues[row] ? "true" : "false");
            break;
```

```
}
case TYPE_TINYINT:
{
   OctetSeq octetValues = values.octetValues();
   cout << octetValues[row];
   break;
}
case TYPE_SMALLINT:
{
   ShortSeq& shortValues = values.shortValues();
   cout << shortValues[row];
   break;
}
case TYPE_INTEGER:
    {
   LongSeq& longValues = values.longValues();
   cout << longValues[row];
   break;
}
case TYPE_REAL:
{
   FloatSeq& floatValues = values.floatValues();
   cout << floatValues[row];
   break;
}
case TYPE_DOUBLE:
case TYPE_FLOAT:
{
   DoubleSeq& doubleValues = values.doubleValues();
   cout << doubleValues[row];
   break;
}
case TYPE_CHAR:
case TYPE_LONGVARCHAR:
case TYPE_VARCHAR:
{
   StringSeq& stringValues = values.stringValues();
   cout << stringValues[row];
   break;
}
case TYPE_BINARY:
case TYPE_LONGVARBINARY:
case TYPE_VARBINARY:
{
   BinarySeq& binaryValues = values.binaryValues();
   cout << "(binary)";
```

```
            break;
         }
         case TYPE_BIGINT:
         case TYPE_DECIMAL:
         case TYPE_NUMERIC:
         {
            DecimalSeq& decimalValues = values.decimalValues();
            cout << "(decimal)";
            break;
         }
         case TYPE_DATE:
         {
            DateSeq& dateValues = values.dateValues();
            // Assumption: time_t is seconds from Jan 1, 1970
            time_t t = (time_t)((dateValues[row].dateValue - 40222.0) *
               86400);
            cout << ctime(&t);
            break;
         }
         case TYPE_TIME:
         {
            TimeSeq& timeValues = values.timeValues();
            cout << "time: " << timeValues[row].timeValue;
            break;
         }
         case TYPE_TIMESTAMP:
         {
            TimestampSeq& timestampValues = values.timestampValues();
            time_t t = (time_t)((timestampValues[row].dateValue +
            timestampValues[row].timeValue - 40222.0) * 86400);
            cout << ctime(&t);
            break;
         }
       }
     }
     cout << endl;
   }
}
int main(int argc, char** argv)
{
   ORB_var orb = ORB_init(argc, argv, "");
   Manager_var manager = Manager::
       _narrow(Object_var(orb->string_to_object("iiop://myhost:9000")));
   Session_var session = manager->createSession("jagadmin", "");
   Ping_var p = Ping::_narrow(Object_var(session->create("Test/Java")));
   ResultSet_var rs = p->results();
```

```
   printResultSet(rs.in());
   {
      char c;
      fprintf(stderr, "Press Return to continue...");
      c = getchar();
   }
   return 0;
}
```

Retrieving the result set

To retrieve the result set, you must instantiate the stub and call the component method that returns a result set to the client. This example instantiates the stub from the Java component in the Test package in a session as an object p of type Ping_var using the _narrow method. The component method, results() is called on p which returns the result set rs.

```
 Ping_var p = Ping::_narrow(Object_var(session-
>create("Test/Java")));
   ResultSet_var rs = p->results();
```

Iterating through the rows and columns

You must process each column value of each row one at a time. In this example, the processing is contained in a method (which you can reuse in other applications) called printResultSet(). printResultSet() takes the result set rs as an input parameter.

```
printResultSet(rs.in());
```

The method uses the length() method to determine how many columns, nc, are in the result set, rs, and displays the number of columns and rows; the number of rows is represented by the variable rows. The method uses a for loop to iterate through each row, row, in the result set; and a nested for loop to iterate through each column, column, in the current row. The method must check for null values before it can process and print the values in each of the columns of the current row. After checking for and printing out null values, the method continues to the next column in the current row.

```
void printResultSet(const ResultSet& rs)
{
  ULong nc = rs.columns.length();
  cout << rs.rows << " rows, " << nc << " columns" <<
endl;
  for (ULong row = 0; row < rs.rows; row++)
  {
```

```
cout << "row " << row << ": ";
for (ULong column = 0; column < nc; column++)
{
  if (column > 0)
  {
    cout << ", ";
  }
  BooleanSeq& nulls =
       ((ColumnSeq&)rs.columns)[column].nulls;

  if (row + 1 <= nulls.length() && nulls[row])
  {
    cout << "null";
    continue;
  }
```

Retrieving the column datatype and processing values

In the body of printResultSet(), the _d() method (the discriminator method) is used to retrieve the datatype of the column and switch/case processing is used to process the column value in the current row. values is a reference to a Data object that represents the column value. _d() returns the datatype of the referenced value to the switch statement and the body of the case statement that matches the datatype is executed. In each case, the current row's column value that corresponds to the case's datatype is printed.

For the Date, Time, Timestamp datatypes, some conversion is required to print a value in a standard format (such as "January 5, 1998").

```
 Data& values =
((ColumnSeq&)rs.columns)[column].values;
      switch (values._d())
      {
         case TYPE_BIT:
         {
            BooleanSeq& booleanValues =
values.booleanValues();
            cout << (booleanValues[row] ? "true" :
"false");
            break;
         }
         case TYPE_TINYINT:
         {
            OctetSeq octetValues =
values.octetValues();
            cout << octetValues[row];
```

```
                break;
            }
            case TYPE_SMALLINT:
            {
                ShortSeq& shortValues =
values.shortValues();
                cout << shortValues[row];
                break;
            }
            case TYPE_INTEGER:
                {
                LongSeq& longValues = values.longValues();
                cout << longValues[row];
                break;
            }
            case TYPE_REAL:
            {
                FloatSeq& floatValues =
values.floatValues();
                cout << floatValues[row];
                break;
            }
            case TYPE_DOUBLE:
            case TYPE_FLOAT:
            {
                DoubleSeq& doubleValues =
values.doubleValues();
                cout << doubleValues[row];
                break;
            }
            case TYPE_CHAR:
            case TYPE_LONGVARCHAR:
            case TYPE_VARCHAR:
            {
                StringSeq& stringValues =
values.stringValues();
                cout << stringValues[row];
                break;
            }
            case TYPE_BINARY:
            case TYPE_LONGVARBINARY:
            case TYPE_VARBINARY:
            {
                BinarySeq& binaryValues =
values.binaryValues();
                cout << "(binary)";
```

```
            break;
         }
         case TYPE_BIGINT:
         case TYPE_DECIMAL:
         case TYPE_NUMERIC:
         {
            DecimalSeq& decimalValues =
values.decimalValues();
            cout << "(decimal)";
            break;
         }
         case TYPE_DATE:
         {
            DateSeq& dateValues = values.dateValues();
            // Assumption: time_t is seconds from Jan
1, 1970
            time_t t =
(time_t)((dateValues[row].dateValue - 40222.0) *
               86400);
            cout << ctime(&t);
            break;
         }
         case TYPE_TIME:
         {
            TimeSeq& timeValues = values.timeValues();
            cout << "time: " <<
timeValues[row].timeValue;
            break;
         }
         case TYPE_TIMESTAMP:
         {
            TimestampSeq& timestampValues =
values.timestampValues();
            time_t t =
(time_t)((timestampValues[row].dateValue +
            timestampValues[row].timeValue - 40222.0) *
86400);
            cout << ctime(&t);
            break;
         }
       }
     }
     cout << endl;
   }
}
```

Handling exceptions

The client-side ORB throws two kinds of exceptions:

- CORBA system exceptions – These exceptions are defined in the CORBA specification.
- User-defined exceptions – These exceptions must be defined in the component's IDL definition.

CORBA system exceptions

The CORBA specification defines the list of standard system exceptions. In C++, all CORBA system exceptions are mapped to a C++ class that is derived from the standard SystemException class defined in the CORBA module. You may want to trap the exceptions shown in this code fragment:

```
try
{
... // invoke methods
}
catch (CORBA::COMM_FAILURE& cf)
{
... // A component aborted the EAServer transaction,
    // or the transaction timed out. Retry the
    // transaction if desired.
}
catch (CORBA::TRANSACTION_ROLLEDBACK& tr)
{
... // possibly retry the transaction
}
catch (CORBA::OBJECT_NOT_EXIST& one)
{
... // Received when trying to instantiate
    // a component that does not exist. Also
    // received when invoking a method if the
    // object reference has expired
    // (this can happen if the component
    // is stateful and is configured with
    // a finite Instance Timeout property).
    // Create a new proxy instance if desired.}
}
catch (CORBA::NO_PERMISSSION& np)
{
... // tell the user they are not authorized
}
catch (CORBA::SystemException& se)
```

```
{
... // report the error but don't bother retrying
}
```

Note Not all of the possible system exceptions are shown in the example. See the CORBA/IIOP 2.2 Specification (formal/98-02-01) for a list of all the possible exceptions.

User-defined exceptions

In C++, all CORBA user-defined exceptions are mapped to a C++ class that is derived from the standard UserException class defined in the CORBA module. For more information, see "User-defined IDL datatypes" on page 96 and "User-defined exceptions" on page 98.

Note User-defined types must exist in the EAServer IDL repository before you can use them in interface declarations.

Compiling C++ clients

For example C++ client compilation commands, see the C++ tutorial in Chapter 3, "Creating C++ Components and Clients," in the *EAServer Cookbook*.

If the client uses SSL, the following files must also reside on the client machine in a directory specified in the libary search environment variable. In the UNIX column, replace *ext* with the platform extension for shared library files:

Windows	UNIX
libjctssec.dll	*libjctssec.ext*
libjsybscl.dll	*libjsybscl.ext*
libjspks.dll	*libjspks.ext*
libjsentpks.dll	*libjsentpks.ext*
libjintl.dll	*libjintl.ext*

If your C++ compiler does not support namespaces, add this in your makefile's compile line:

```
-DJAG_NO_NAMESPACE
```

On Solaris, the installation includes libraries in two formats for compatibility with different versions of the Solaris CC compiler. Choose the appropriate directory from those listed below:

- *lib* contains libraries that are compatible with the 6.*x* compiler, stripped of symbol information for production use.
- *devlib* contains libraries that are compatible with the 6.*x* compiler, for debugging use.
- *lib_sol4x* contains libraries that are compatible with the 4.*x* compiler, for production use.
- *devlib_sol4x* contains libraries that are compatible with the 4.*x* compiler, for debugging use.

Deploying C++ clients

To deploy a C++ client on another machine:

1 Install the EAServer client runtime if not done already, including C++ libraries. If the client uses SSL, make sure the SSL client runtime support is installed.

2 Copy the client's executable to the machine.

3 Configure the environment as described in "Verify your environment" in Chapter 3, "Creating C++ Components and Clients," in the *EAServer Cookbook*.

Using the CosNaming interface

EAServer allows you to use the CORBA CosNaming interface to instantiate proxies in your client applications. This technique of instantiating proxies is not recommended, because it requires use of deprecated SessionManager::Factory methods. "Instantiating stub instances" on page 276 describes the recommended technique for stub instantiation.

You do not need to use the CosNaming API in clients to realize the benefits incurred by using logical component names. EAServer uses the CosNaming API to resolve component names in the implementation of the Session::lookup and Session::create methods.

To use CosNaming, follow these steps:

Step	What it does	Detailed explanation
1	Configure ORB properties, including the ORB runtime driver class and the EAServer naming server URL, then initialize the ORB runtime.	"Configure and initialize the ORB for CosNaming use" on page 295
2	Instantiate the CORBA CosNaming name service and obtain the initial naming context.	"Obtain an initial naming context" on page 296
3	Instantiate proxy objects and narrow them to the stub interface.	"Resolving component proxies" on page 297
4	Call the proxy objects to remotely invoke component methods.	"Invoking methods" on page 283

Note All examples in this section are taken from the *arith.cpp* file for the C++ client tutorial, describe in Chapter 3, "Creating C++ Components and Clients," in the *EAServer Cookbook*.

Configure and initialize the ORB for CosNaming use

"Configure and initialize the ORB runtime" on page 277 describes how to initialize the ORB and configure run-time properties. One additional property is required in applications that use the CosNaming API.

You must set ORBNameServiceURL property to specify the IIOP URL to the EAServer name service. This parameter can also be set in an environment variable, JAG_NAMESERVICEURL. Use the following syntax for values:

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

where:

hostname is the host machine name for the server that serves as the name server for your client. If omitted, the default host name applies.

iiop-port is the IIOP port number for the server.

initial-context is the initial naming context, which you set in the server property Initial Context. This can be used to set a default prefix for name resolution. For example, if you specify `USA/Sybase/`, all names that you resolve with the context are assumed to be relative to this location in the name hierarchy. When specifying the initial context, a trailing slash is optional; it is added automatically if you do not specify an initial context that ends with a slash.

If your application uses a cluster of servers, the cluster may use multiple name servers. In this case, specify the URL (host machine name and IIOP port number) for each name server in a list separated by semicolons and no white space. Include the cluster's initial naming context only with the last URL. For example:

```
iiop://host1:9000;iiop://host2:9000/USA/Sybase/
```

Obtain an initial naming context

After initializing the ORB, call the resolve_initial_references method to obtain the initial naming context. The naming context is an object that implements the CosNaming::NamingContext IDL interface; it is used to resolve EAServer component and service names to server-side objects.

Obtaining the initial context

The example below shows how the initial naming context is retrieved:

```
// Obtain the CORBA CosNaming initial naming context
that
// we will use to resolve objects by name. The ORB
retrieves
// the naming server address from command line arguments
or
// the environment.

CORBA::Object_var obj =
  orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var nc =
  CosNaming::NamingContext::_narrow(obj);
if (CORBA::is_nil(nc)) {
    cout << "Error: Null NamingContext instance.
Exiting.";
    return -1;
    }
```

Introduction to CosNaming name resolution

The initial NamingContext will have the name context that was specified in the `NameServiceURL` ORB initialization property. The client invokes the NamingContext::resolve operation to obtain an instance of the EAServer authentication service as well as component instances.

The NamingContext::resolve operation takes a CosNaming::Name parameter, which is a sequence of CosNaming::NameComponent structures.

A name is represented by a sequence of NameComponent instances, with the *id* field of each instance set to a node of the name.

As a convenience, the EAServer name service allows you to specify multiple nodes of a name in one NameComponent instance, using a forward slash (/) to separate nodes.

NamingContext::resolve resolves a name to an object; this method either returns a CORBA::Object instance or throws one of the exceptions described below:

- NotFound indicates that the name is not bound to an object, the name does not exist, or some node in the indicated hierarchy does not exist; the why field contains an enumeration that encodes the reason why the name was not found.
- InvalidName indicates that the name is malformed.
- CannotProceed or a CORBA SystemException indicates that an error has occurred. “Handling exceptions” on page 292 describes CORBA system exceptions.

Resolving component proxies

Proxy objects are instantiated as follows:

1 Create a NameComponent array that names the component. Component names are composed as follows:

```
server-context/package/component
```

where

- *server-context* is the root naming context for the server where the component is installed. You can view and edit this setting in the Naming Services tab of the Server Properties window. The default for a new server is "/". If you specify an initial name context when initializing the ORB properties, then resolved names are assumed to be relative to the initial name context.
- *package* is the EAServer package name in which the component is installed, as displayed in EAServer Manager.
- *component* is the component name, as displayed in EAServer Manager.

2 Call the NamingContext::resolve method to instantiate a factory object for the component.

3 Narrow the CORBA Object reference to a SessionManager::Factory instance.

4 Call the factory's create method and narrow the return value by calling the _narrow method in the class for the interface. The create method requires a username and password to authenticate the end user.

The example below instantiates a component "CPPArithmetic," installed in package "Tutorial," hosted on a server with a null root context. The username and password are *Guest* and *GuestPassword*, respectively. The component implements the IDL interface Tutorial::CPPArithmetic, and the code narrows the proxy object to that interface.

```
// Build a CosNaming::Name object that contains the
// name of the tutorial component,
Tutorial/CPPArithmetic

name[0].id = CORBA::string_dup( component_name );
name[0].kind = CORBA::string_dup( "" );

// Obtain a factory for component instances by
// resolving the component name
cout << "Creating component instance for "
  << component_name << "\n\n";
obj = nc->resolve(name);
SessionManager::Factory_var arithFactory =
  SessionManager::Factory::_narrow(obj);

if (CORBA::is_nil(arithFactory)) {
  cout << "ERROR: Null component factory. " <<
tutorial_help ;
  return -1;
```

```
}

// Use the factory to create an instance, passing the
// username and password for authorization
Tutorial::CPPArithmetic_var arith =
  Tutorial::CPPArithmetic::_narrow
    ( arithFactory->create("Guest", "GuestPassword"));

// Verify that we really have an instance.
if (CORBA::is_nil(arith)) {
  cout << "ERROR: Null component instance. " <<
tutorial_help ;
  return -1;
}
```

Using CORBA ORB implementations other than EAServer

EAServer's IIOP implementation allows you to use any CORBA client ORB to invoke EAServer components. You can also use the EAServer client ORB to execute components that are hosted by another vendor's server ORB.

Connecting to EAServer with a third-party client ORB

In some cases, you may wish to use another vendor's ORB in your client applications. For example, you may have an existing installation of the ORB on client workstations.

Clients that use another ORB can use the same code as the EAServer ORB, except for the following differences:

- You must use stub classes generated by the vendor's IDL-to-C++ compiler rather than stubs generated by EAServer Manager.
- Your code to connect to EAServer and instantiate components may differ.

Generating compatible C++ stubs

CORBA Interface Definition Language (IDL) files are required in order to use another vendor's ORB implementation class. EAServer Manager generates IDL files for components when you create or import them using EAServer Manager. Use the IDL-to-C++ compiler that comes with your ORB software to generate compatible stubs.

For information about which component IDL files and EAServer IDL files you need to use to generate stubs for other ORBs, see "Generating compatible Java stubs" on page 238 (although this section refers to Java clients, it also applies to C++ clients).

EAServer IDL modules

Use the ORB vendor's IDL-to-C++ compiler to generate stubs for the files in the table, "EAServer IDL files" on page 300. All IDL files are installed in the EAServer *include* subdirectory. "Writing CORBA C++ clients" on page 275 describes how these interfaces are used to instantiate EAServer components and call component methods. For additional information, see the comments in each IDL file.

EAServer IDL files

File name	Description
SessionManager.idl	Defines interfaces for session-based creation of EAServer component instances.
BCD.idl	Defines the CORBA datatypes for EAServer's binary and fixed-point numeric datatypes.
MJD.idl	Defines the CORBA datatypes for EAServer's date and time datatypes.
TabularResults.idl	Defines the CORBA datatypes that represent result sets returned by a method invocation.

Performing datatype conversion

EAServer provides C++ header files to convert from the EAServer CORBA datatypes to those commonly used in C++. "Supported datatypes" on page 244 lists the datatypes displayed in EAServer Manager, the equivalent CORBA IDL types, and the C++ datatypes used in stub methods. If you are using another vendor's ORB, use the EAServer header files in your application. For languages other than C++, see the comments in the IDL files for details on how the data is interpreted.

Instantiating components using a third-party ORB

EAServer's naming service cannot be used with other client ORBs, so you must use the EAServer SessionManager::Manager interface to instantiate components from another ORB, as described in "Instantiating stub instances" on page 276.

Also, you must use standard format IORs, not the URL format, as described in "Creating a Manager instance" on page 280.

Connecting to third-party ORBs using the EAServer client ORB

You can use the EAServer client-side ORB to execute components hosted by another vendor's server-side ORB, as long as the server-side ORB accepts IIOP connections and the required interfaces are defined in standard CORBA IDL. Implement your client as follows:

1 Import all the required IDL modules into EAServer Manager, as described in "Importing existing IDL modules" on page 100.

2 Generate stubs for each imported module using EAServer Manager, as described in "Generating stubs" on page 274. You must generate stubs for each module individually.

PART 5

PowerBuilder Components and Clients

While PowerBuilder is not included with EAServer, the products are fully integrated and work well together. A PowerBuilder application can act as a client to any EAServer component. In addition, EAServer can contain PowerBuilder custom class (nonvisual) user objects that execute as middle-tier components.

Using PowerBuilder 7.0 or later, you can create nonvisual objects (NVOs) that run natively in EAServer as EAServer components. You can also create NVO proxies for EAServer components, then use the proxies in PowerBuilder client applications.

CHAPTER 16

Creating PowerBuilder Components

While PowerBuilder is not included with EAServer, the products are fully integrated and work well together. EAServer hosts the PowerBuilder virtual machine natively. This means that EAServer can communicate directly with PowerBuilder nonvisual user objects, and vice versa. EAServer components developed in PowerBuilder can take full advantage of the ease of use and flexibility of PowerScript® and the richness of PowerBuilder's system objects.

The PowerBuilder IDE runs on Windows platforms, but you can deploy PowerBuilder components to EAServer on any platform for which a compatible PBVM is available, including most UNIX platforms. For more information, see the EAServer *Release Bulletin* for your platform.

The PowerBuilder IDE includes wizards to create EAServer components and deployment projects. While you can edit PowerBuilder component properties in EAServer Manager, you should use the PowerBuilder IDE instead. Changes you make outside of the IDE can be overwritten when you redeploy your project to EAServer from the IDE. For information on using the wizards, see the *Application Techniques* manual in the PowerBuilder documentation. If you must set additional component properties that cannot be set from the PowerBuilder IDE, consider creating a script or batch file that uses the jagtool set_props command to configure these additional settings. This allows you to maintain an automated deployment mechanism. For more information, see Chapter 12, "Using jagtool and jagant," in the *EAServer System Administration Guide*.

PowerBuilder provides full-fledged support for EAServer component technologies, including:

- Instance pooling, by configuring the Pooling setting in the wizards and optionally implementing lifecycle methods to control whether specific instances are pooled.
- Server-managed transactions, by configuring the Transactions settings in the wizards and by calling the methods in the TransactionServer context object.

- Connection caching, by using PowerScript DataStore objects in your implementation code.
- Result sets, by using the PowerScript DataStore, ResultSet, and ResultSets objects. You can use the DataStore object to return result sets that are presented in the client using DataWindow controls. You can also use the ResultSet and ResultSets objects to return tabular results to clients of other types.
- Intercomponent calls, using the CreateInstance method in the TransactionServer object to obtain proxies for components.

  Note By default, the TransactionServer CreateInstance method invokes the EAServer name service to create proxies. Proxies for remote components may be returned by the name service rather than an instance that is running locally. To guarantee that a locally installed instance is used, specify the component as `local:`***`package`***`/`***`component`***, where *package* is the package name and *component* is the component name, for example, `local:CtsSecurity/SessionInfo`. The call fails if the component is not installed in the same server.

- Logging, using the ErrorLogging object to write error or status messages to the server log file.
- Running independent of client interaction, using the EAServer thread manager or service component model.

For details on implementing components that use these features, see the *Application Techniques* manual in the PowerBuilder documentation.

CHAPTER 17

Creating PowerBuilder Clients

While PowerBuilder is not included with EAServer, the products are fully integrated and work well together. PowerBuilder 7.0 or later allows you to generate NVOs that act as proxies for EAServer components. Using a proxy, you can call component methods as if they were implemented as local NVO methods. You can call any type of component from a PowerBuilder client, not just PowerBuilder NVO components.

To create a PowerBuilder client, use the PowerBuilder IDE wizards to generate proxies for the EAServer components that the client calls. Use the PowerScript Connection or JaguarORB object to connect to the server and instantiate proxies for the components. You can invoke the proxy methods to call the component's business methods.

To create clients that call EJB components, you can use the same proxy wizard that you use for any other component. You can also use the EJB Client Proxy wizard to create EJB proxies. The proxies generated by this wizard use the EJB client PowerBuilder extension. This extension is a wrapper for Java, and therefore provides more flexibility in communicating with EJBs. For example, an EJB client can manipulate a Java class returned from an EJB method call through its proxy. The PowerBuilder Connection object has a smaller footprint (and thus is easier to deploy) because it does not require a JRE to be installed on the computer where the client resides. Connectivity to the server is also faster with the connection object, because there is no delay while a JRE loads.

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

PART 6

ActiveX Components and Clients

This part explains how to build ActiveX components and clients.

CHAPTER 18

ActiveX Overview

Overview

ActiveX/COM is a Microsoft component technology. Many IDE tools such as Visual Basic allow you to create ActiveX components and write code to call methods in registered ActiveX components.

Any nonvisual ActiveX component can be installed as an EAServer component (though you may need to define an "adaptor," or wrapper class to handle methods that use unsupported parameter datatypes). EAServer uses COM and ActiveX automation support to execute ActiveX component methods. Consequently, all EAServer ActiveX components must support COM's automation interface (the IDispatch interface). Many application development tools, such as Microsoft Visual Basic, can be used to create ActiveX components that are compatible with EAServer. Once installed in EAServer, ActiveX components can be called by clients of any type.

To support ActiveX clients, EAServer provides an ActiveX automation server that interacts with the server using the C++ CORBA ORB and standard CORBA IIOP. Because ActiveX clients use IIOP rather than the DCOM network protocol, they can call EAServer components of any type and interact with servers running on platforms that do not support ActiveX.

No client managed transactions
This release does not provide an ActiveX client interface to manage transactions. Consequently, ActiveX clients cannot call component methods that have the Mandatory transaction attribute.

Requirements

ActiveX component requirements

The following list describes software and hardware requirements for developing ActiveX components.

All software that is required to run ActiveX components in EAServer is supplied with the EAServer product.

- **Operating system** ActiveX components require Windows NT 4.0 or Windows 2000.
- **Development tools** To create ActiveX components, you need an ActiveX-enabled IDE. The following list shows some of the many IDEs you can use:
  - Visual C++ 4.0 or later
  - Visual Basic 5.0 or later

Note EAServer provides native, built-in support for PowerBuilder libraries. Though PowerBuilder supports ActiveX development, Sybase recommends that you use EAServer's native PowerBuilder support to run PowerBuilder objects in EAServer.

ActiveX client requirements

EAServer's ActiveX client model allows you to invoke EAServer components from ActiveX enabled IDEs such as Visual Basic.

To develop and run ActiveX clients, you need:

- A supported client operating system such as Windows NT 4.0 or Windows 2000. The ActiveX client can call server components executing on any any platform. See the *EAServer Installation Guide* for Windows for more information on operating system requirements.
- C++ client and ActiveX runtime files, which includes *jagproxy.dll*, the required *.tlb* and *.reg* files, and the C++ DLLs required for use with an ActiveX CORBA client. See the *EAServer Installation Guide* for Windows for instructions on installing the client runtime.
- An ActiveX-enabled IDE. The following list includes some of the ActiveX-enabled IDEs you can use:
  - Visual Basic 4.0 or later
  - Visual C++ 4.0 or later
- Type libraries (you generate these files using EAServer Manager).
- Registry files (you generate these files using EAServer Manager).
- The Microsoft tool *uuidgen.exe*, which is provided with EAServer on Windows, must be installed on the server. The Microsoft tool *midl.exe* must be installed on your client machine and specified in the path. *midl.exe* must support the /tlb command-line option. You can issue this command to see the options that your *midl.exe* version supports:

  ```
  midl /?
  ```

  midl.exe is distributed by Microsoft as part of their development tools and as part of the Win32 SDK. Many tools from other vendors include a redistribution of the Win32 SDK. Visual C++ 4.2 or higher contain the correct version of *midl.exe*. You can also download *midl.exe* from the Microsoft Developer Network web site at http://msdn.microsoft.com.

Note Make sure that an older version of *midl.exe* is not located in a directory that is specified prior to the current *midl.exe* directory in the PATH environment variable.

- EAServer Manager on Windows NT or Windows 2000.

Note See the *EAServer Installation Guide* for Windows for additional system requirements.

ActiveX datatype support

Table 18-1 on page 315 the datatypes and argument modes supported by EAServer Manager, and their corresponding CORBA IDL and ActiveX types. Each IDE script language uses a different syntax to represent these types. See your tool's documentation for more information.

For client development, most IDEs provide some way to view the interfaces exposed by registered automation servers. If your IDE provides an object browser, you can look up the interface for the proxy object and see it displayed in the IDE's specific syntax. For example, in Visual Basic, you can browse registered ActiveX interfaces using the Object Browser window.

For component development, you can code your component methods to use supported ActiveX datatypes, then import the DLL or type library into EAServer Manager to define the component's IDL interface.

Argument modes specify how an argument is passed:

- in – read-only; arguments are passed by value
- inout – read/write; arguments are passed by reference
- out – write only
- return – the method returns a value of this datatype

Use in when the parameter is used to pass a value without changing the value. All parameters specified as in arguments are passed by value except for the string datatype (BSTR*). Use inout when the parameter is used to pass a value and change it. In the inout and out argument modes, the datatype of the parameter being passed must be identical to the datatype used for the same parameter in the server component method. Otherwise, no coercion is performed and an exception is thrown.

Table 18-1 on page 315 the datatypes and argument modes supported by EAServer Manager, and their corresponding CORBA IDL and ActiveX types. The ActiveX column contains ActiveX datatype specification as it is defined at runtime in the ActiveX VARIANTARG structure. The ActiveX datatypes are used by OLE automation to pass data. For information on how your IDE's datatypes correspond to these types, see your IDE's documentation.

Table 18-1: ActiveX datatypes

EAServer Manager	CORBA IDL type	Argument mode	ActiveX (automation) type
boolean	boolean	in inout out return	VT_BOOL VT_BOOL | VT_BYREF VT_BOOL | VT_BYREF VT_BOOL
integer<16>	short	in inout out return	VT_I2 VT_I2 | VT_BYREF VT_I2 | VT_BYREF VT_I2
integer<32>	long	in inout out return	VT_I4 VT_I4 | VT_BYREF VT_I4 | VT_BYREF VT_I4
float	float	in inout out return	VT_R4 VT_R4 | VT_BYREF VT_R4 | VT_BYREF VT_R4
double	double	in inout out return	VT_R8 VT_R8 | VT_BYREF VT_R8 | VT_BYREF VT_R8
string	string	in inout out return	VT_BSTR VT_BSTR | VT_BYREF
octet	octet	in inout out return	VT_UI1 VT_UI1 | VT_BYREF VT_UI1 | VT_BYREF VT_UI1
binary	BCD::Binary	in inout out return	VT_ARRAY | VT_UI1 (safe array) VT_ARRAY | VT_UI1 | VT_BYREF VT_ARRAY | VT_UI1 | VT_BYREF VT_ARRAY | VT_UI1 (safe array)

EAServer Manager	CORBA IDL type	Argument mode	ActiveX (automation) type
decimal	BCD::Decimal	in inout out return	VT_R8 VT_R8 | VT_BYREF VT_R8 | VT_BYREF VT_R8
money	BCD::Money	in inout out return	VT_CY VT_CY | VT_BYREF VT_CY | VT_BYREF VT_CY
date	MJD::Date	in inout out return	VT_DATE VT_DATE | VT_BYREF VT_DATE | VT_BYREF VT_DATE
time	MJD::Time	in inout out return	VT_DATE VT_DATE | VT_BYREF VT_DATE | VT_BYREF VT_DATE
timestamp	MJD::Timestamp	in inout out return	VT_DATE VT_DATE | VT_BYREF VT_DATE | VT_BYREF VT_DATE
ResultSet	TabularResults:: ResultSet	out return	VT_DISPATCH VT_DISPATCH | VT_BYREF
ResultSets	TabularResults:: ResultSets	out return	VT_DISPATCH VT_DISPATCH | VT_BYREF
interface (any IDL interface)	*interface*	in inout out return	VT_DISPATCH VT_DISPATCH | VT_BYREF VT_DISPATCH | VT_BYREF VT_DISPATCH
structure (any IDL structure)	See “Structure support” on page 317.	in inout out return	VT_DISPATCH VT_DISPATCH | VT_BYREF VT_DISPATCH | VT_BYREF VT_DISPATCH
enum (any IDL enum)	See “IDL enumeration support” on page 322.	in inout out return	VT_DISPATCH VT_DISPATCH | VT_BYREF VT_DISPATCH | VT_BYREF VT_DISPATCH
typedef (any IDL type alias)	See “IDL typedef support” on page 321.	in inout out return	VT_DISPATCH VT_DISPATCH | VT_BYREF VT_DISPATCH | VT_BYREF VT_DISPATCH

EAServer Manager	CORBA IDL type	Argument mode	ActiveX (automation) type
union (any IDL union)	See "Union support" on page 318.	in inout out return	VT_DISPATCH VT_DISPATCH | VT_BYREF VT_DISPATCH | VT_BYREF VT_DISPATCH
sequence (any IDL sequence)	JCollection (See "Sequence support" on page 321.)	in inout out return	VT_DISPATCH VT_DISPATCH | VT_BYREF VT_DISPATCH | VT_BYREF VT_DISPATCH

IDL attributes are not supported by ActiveX clients.

Because the Timestamp datatype maps to the OLE date datatype, which does not represent thousandths of a second, you could lose precision when accessing a timestamp parameter from a Java component.

ResultSet or ResultSets datatypes can only be specified as a return type.

Only byte arrays are supported—no other type of array is supported. You can also use IDL sequences in lieu of arrays. The EAServer JCollection interface is used by both clients and components to represent IDL sequences. See "Sequence support" on page 321 for more information.

If you develop your component using Visual C++ and the parameter type is BSTR *, then the IDL signature for the parameter argument mode must be inout. In the IDL file, change the parameter mode to inout, regenerate and reregister the *.tlb* and *.reg* files, and modify the ActiveX client to pass the string argument by reference.

You can define get and set methods that control runtime properties for your component. However, using get and set methods to access properties over a network decreases performance.

Structure support

EAServer component interfaces can use IDL structures as parameter types in method definitions. For example, in the home interface for an EJB entity bean component, findByPrimaryKey method typically accepts a structure that represents the primary key for a database row.

ActiveX mapping for IDL structures

An IDL structure is mapped to an IDispatch interface, with every data member in the structure being mapped to a property in the corresponding IDispatch interface. All types supported by the ActiveX proxy are supported as members inside a structure, including structures and sequences.

If an EAServer component interface uses IDL structures, EAServer Manager generates IDispatch interfaces for each IDL structure when generating type libraries for the component interface definitions.

Example: using a structure in Visual Basic

bookStore::custCreditKey is an IDL structure defined in the IDL fragment below:

```
module bookStore
{
    struct custCreditKey
    {
        string custName;
        string creditType;
    };
};
```

To use this type in Visual Basic, you must first create a reference to the type library generated by EAServer Manager to represent the bookStore IDL module. In Visual Basic code, you can create an instance of the structure and set the fields like this:

```
Set pKey = New bookStore.custCreditKey
pKey.creditType = "VISA"
pKey.custName = "Morry"
```

Explicit version implicit field initialization
Structure fields that use complex types such as struct, union, object, date, time, or timestamp must be initialized explicitly. If you do not initialize these fields before passing the union as an EAServer method parameter or return value, the ActiveX dispatcher throws a marshalling exception. Fields of other types are implicitly set to a default value.

Union support

EAServer maps IDL unions to an interface with properties to set and retrieve union member values.

About IDL unions

An IDL union is similar to an IDL structure, except that an instance of a union can contain a value for one field only. IDL union declarations look like this:

```
module TMod
  {
    union TUn switch (long)
    {
      case 5: long lVal;
      case 9, 7: short sVal;
      default: double dVal;
  };
};
```

The declaration has a **discriminator** of the datatype specified by `switch (typename)`. Each field declaration must have a case specifier that describes the matching discriminator value. There can be one default field that applies when the discriminator matches no other value. At runtime, the union's discriminator tells which field contains the current value. In the example above, a discriminator value of seven indicates that the *sVal* field contains the current value.

Supported discriminator and field types

The following discriminator types are supported by the ActiveX proxy:

- signed or unsigned 2, 4 byte integers (including enumerations)
- boolean
- a typedef that aliases one of the above types
- An enumeration that switches on the IDL char type.

All IDL types presently supported by the ActiveX proxy can be used as union fields, including other unions.

IDL unions cannot have a field named *discriminator*, as it will conflict with the name of the discriminator variable in the generated ActiveX type.

ActiveX mapping for unions

An IDL union maps to an ActiveX interface named with the following properties and methods:

- One get/set property for each field, with the same name.

- A get/set discriminator property that represents the discriminator value. The discriminator has the ActiveX datatype that corresponds to the IDL discriminator type.

Setting and getting member values

To set a union member, simply set the property for that member. The discriminator value changes automatically to match the member you set.

To access the value of a member, first verify that the discriminator value is in the set of allowable cases for that member, then reference the matching property. The ActiveX proxy throws an exception if you attempt to access a member while the discriminator value is not in the set of case values for that member.

Explicit version implicit field initialization
Union fields that use complex types such as struct, union, object, date, time, or timestamp must be initialized explicitly. If you do not initialize these fields before passing the union as an EAServer method parameter or return value, the ActiveX dispatcher throws a marshalling exception. Fields of other types are implicitly set to a default value.

Example

As an example, consider the following IDL union:

```
module TMod
  {
    union TUnion switch (long)
    {
      case 5: long lVal;
      case 9, 7: short sVal;
      default: double dVal;
  };
};
```

The following Visual Basic code sets each member:

```
dim myUnion as TUnion
set myUnion = new TUnion
myUnion.lVal = 43000
myUnion.sVal = 43
myUnion.dVal = 43.43
```

The following code checks the discriminator and accesses the value if the lVal member is set:

```
if (myUnion.discriminator = 5) then
  print "Current value is " & myUnion.lVal
endif
```

Sequence support

EAServer component interfaces can use IDL sequences as parameter or return types in method definitions. For example, in the home interface for an EJB entity bean component, finder methods may return a sequence of remote interface proxies for the entity bean.

In ActiveX clients and components, sequences are represented by the JCollection IDispatch interface. This interface is implemented by the EAServer ActiveX proxy.

The JCollection interface is documented in Chapter 4, "ActiveX Client Interfaces," in the *EAServer API Reference*.

IDL typedef support

In IDL, the typedef construct defines an alias for an existing type. For example:

```
typedef short TShort;
```

short is the existing type and TShort is an alias for the same. Aliases to any type that is supported by the ActiveX proxy are supported. Nested IDL typedef declarations are supported, such as:

```
typedef short TShort;
typedef TShort MyKeyType;
```

Because ActiveX does not support type aliases, EAServer translates each use of an IDL typedef with the equivalent ActiveX base type declaration. Alias names are not preserved in the ActiveX representation, but you can use IDL interfaces that use type aliasing.

IDL enumeration support

Enumerations represent a set of symbolic values. In IDL, an enumeration defines a set of constants that are represented by symbolic names, for example:

```
enum ShirtSize { xl, l, m, s }
```

The ActiveX proxy maps this IDL enumeration to an Microsoft IDL (MIDL) enumeration as:

```
typedef enum {
    xl = 0,
    1 = 1,
    m = 2,
    s = 3
} ShirtSize;
```

You can declare IDL enumerations globally, within a module, or within an interface. The ActiveX proxy supports only enumerations that are declared in a module or interface. Enumerations map to MIDL enumerations as follows:

- The IDL enumeration *e* declared in module *m* translates to MIDL enumeration *e* in type library *m*.
- Enumeration *e* declared in interface *i* in module *m* translates to MIDL enumeration *i_e* in type library *m*. MIDL does not allow enumerations to be declared inside an interface, so the interface is declared at the type-library level, and the interface and enumerations names are concatenated to avoid name collisions between like-named IDL enumerations declared in different interfaces.

In Visual Basic, you can refer to an enumeration's members as:

```
enum.member
```

Where *enum* is the enumeration name and *member* is the member name.

If your ActiveX development tool supports MIDL enumerations, you should use the symbolic member names rather than the hard coded constants. Doing so isolates your code from changes to the IDL enumeration definition.

In tools that do not support enumerations, you must use the enumeration's integer constants rather than the symbolic names. However, the constants associated with an enumeration are subject to change if the IDL definition changes. To minimize the effect of changes to your source code, declare variables and assign the constant value to them. For example:

```
dim shirtsize_xl as integer
shirtsize_l = 0
```

Result-set support

A ResultSet or ResultSets datatype can be specified only as a return type or a parameter type with an out argument mode. You cannot define a method as having both a ResultSet or ResultSets return type and a ResultSet or ResultSets parameter type.

A result set is returned as a RecordSet object. After retrieving the result set, you can process it using the methods in the interfaces below. These interfaces are documented in Chapter 4, "ActiveX Client Interfaces," in the *EAServer API Reference*:

- RecordSet Interface – provides methods to iterate through the rows in each result set.
- Fields Collection – contains the Field objects that represent the columns in a row.
- Field Interface – represents one column in a row.

Note For compatibility with previous releases, you can still use GetRecordSet method to retrieve an ActiveX RecordSet interface pointer that can be used to retrieve the row results.

Algorithm to retrieve result sets

The pseudocode below illustrates a typical algorithm for retrieving result sets using a RecordSet object. getEmployeeDetails() is a method in the component that returns a single result set as a RecordSet object. The algorithm executes three nested loops, as follows:

- The outermost loop iterates through RecordSet objects. Each object contains rows from one result set. After rows have been retrieved, the example calls the RecordSet.NextRecordSet method. This method returns the next RecordSet object. The outermost loop terminates when RecordSet.NextRecordSet has set the RecordSet.EOF property to true.
- The middle loop iterates through the rows in a result set, calling the RecordSet.MoveNext method until the RecordSet.EOF property tests as true. Inside the loop, the RecordSet.Fields property provides a Fields object that allows access to the row's columns.
- The innermost loop iterates through the columns in a row, using the Fields.Item property to retrieve the Field object that represents each column.

Here is the algorithm:

```
Integer employee_id
RecordSet =
proxycomponent.getEmployeeDetails(employee_id)

DO
    // Position the row pointer before the first row.
    RecordSet.MoveFirst()

    // Iterate through all the rows.
    WHILE RecordSet.EOF = FALSE

       // Fields object represents the current row.
       Fields = RecordSet.Fields

       // Iterate through columns.
       FOR i = 0 TO i = (Fields.Count - 1)

           Field = Fields.Item(i)

           ... retrieve Field  properties to process
column
               data as desired ...

       END FOR

       // Move to the next row.
       RecordSet.MoveNext()

    END WHILE

    // Move to the next result set, if any.
    RecordSet = RecordSet.NextRecordSet()

WHILE RecordSet.EOF = FALSE
```

The logic in this example executes correctly if a method has not returned result sets. In this case, the RecordSet.EOF property is always false.

Some scripting languages may allow or require variations on this algorithm, for example:

- You can replace the WHILE loop logic that iterates through rows with a FOR loop that indexes from 1 to RecordSet.Count.

- Some scripting languages provide a FOR EACH loop syntax that allows iterations over an ActiveX collection. You can use this construct to iterate through the Field objects in a Fields collection. For example, in Microsoft Visual Basic, you can use code similar to this:

```
'Get the collection of fields from record set

Set flds = recset.Fields
For Each fld in flds
    'Process each Field as desired
Next fld
```

CHAPTER 19

Creating ActiveX Components

EAServer can load and execute a nonvisual ActiveX programmable object (also called an automation server) as a component. Hereafter, an ActiveX programmable object will be called an ActiveX component.

Procedure for creating ActiveX components

Begin by writing an ActiveX component in an ActiveX-enabled IDE. After you have defined the method prototypes (ActiveX type definitions), use EAServer Manager to import the component, which includes method prototypes and basic component information (such as the component's name); specify additional component properties in EAServer Manager. This allows another developer to create a client that calls the component's methods. You can also use EAServer Manager to define a component, but you will still have to use an ActiveX-enabled IDE to create the component—you cannot use EAServer Manager to export the component to an ActiveX-enabled IDE.

In the ActiveX-enabled IDE, write the method logic for the ActiveX component. The ActiveX component must support the IDispatch interface and cannot contain a user interface. In addition to writing code for standard ActiveX features, you can also write code to implement EAServer-specific features such as error handling, database connection caches, result sets, transactions, intercomponent calls, and data sharing.

After you finish writing the ActiveX component, compile the component into a dynamic link library (DLL) and install (copy and register) it onto the server. See your ActiveX IDE documentation for compilation instructions.

Defining ActiveX components

Defining an ActiveX component means defining the interfaces, transaction properties, and instance properties. If you define an ActiveX component in EAServer before implementing the ActiveX component, the client developer can build the client at the same time you are building the ActiveX component. You can write the entire component in your IDE first and then import it into EAServer, where you set the properties, but in this case, the client cannot be developed concurrently with the component.

You can also individually define each method and parameter using EAServer Manager. After defining the interfaces, you can use EAServer Manager to define the transaction and instance properties.

Chapter 4, "Defining Components" describes how to define and configure new components in EAServer Manager. Chapter 5, "Defining Component Interfaces" describes how to define methods in the component interface.

Warning! When you define an ActiveX component, you must enter the ProgID. Do not use underscores in the ProgID for the component—use a period instead.

Importing ActiveX components

EAServer Manager can import ActiveX components from the component's type library (*.tlb* file) file or DLL. When you import the ActiveX component, it is automatically added to the EAServer IDL repository. All CoClass classes (component object model classes) in the ActiveX component are imported.

Imported interfaces must conform to all EAServer requirements for ActiveX components. (The requirements are listed in "Defining methods" on page 329.)

Importing components does not import TabularResults::ResultSet and TabularResults::ResultSets return types. If a method in the imported component returns a result set, then you must use EAServer Manager to change the return value to TabularResults::ResultSet or TabularResults::ResultSets in EAServer Manager.

Procedure

Before you can use the ActiveX import feature, make sure that you register the *.dll* file using the Windows utility, regsvr32.exe:

```
regsvr32 <fully qualified path name to dll>
```

If you change the location of the *.dll* file after registration, you must reregister the file with the new location.

To import an ActiveX file:

1 Double-click the Packages folder to expand it.

2 Highlight the package to which the component will be added.

If installing to the Components folder, highlight the Components folder.

3 Select File | Install Component from the menu.

4 In the Component wizard dialog, select Import from ActiveX File, then click OK.

Note If the Import from ActiveX File option is not displayed, you are running EAServer Manager or the server on UNIX. You cannot create ActiveX components when either EAServer Manager or the server is running on UNIX.

5 Enter the fully qualified path name of the *.tlb* or *.dll* file from which you are importing. Some development environments do not automatically generate *.tlb* files. Enter the name of the *.dll* file if this is the case. You can use the browse feature to locate either file.

6 Click OK. The component is imported; the component name is the ProgID without the version number and with periods replaced by underscores. You can view the new component along with its methods and parameters from the EAServer Manager.

Defining methods

To define methods, you must specify each method's return type and the number, datatypes, and modes of the method's parameters. See "ActiveX datatype support" on page 314 for more information.

Do not use two consecutive underscores in method names—the underscores and the text following the underscores are deleted when stubs and skeletons are generated. This issue is related to function overloading, which is allowed in Java and C++ but not in ActiveX components. See “Operation declarations” on page 93 for more information.

Warning! You cannot define methods with names that differ only in case—IDL does not support this.

Defining return and parameter datatypes

ActiveX component methods can return any valid datatype. Methods can take zero or more parameters. For each parameter you add, you must specify a name, a datatype, and the argument mode. Datatypes are limited to those supported by EAServer Manager. “ActiveX datatype support” on page 314 describes the supported types.

Defining the transaction property

The transaction property specifies how a component participates in transactions. You can view and change the transaction property using the Transactions tab of the component’s property sheet. For a description of each option on the Transactions tab, see “Transactional component attribute” on page 22. A transaction consists of a number of database updates (which can be performed by multiple components) that are grouped into a single atomic unit of work.

This information is not stored in the EAServer repository, so if you import a component, you must configure this property manually after importing it.

For a full description of how EAServer handles transactions, Chapter 2, “Understanding Transactions and Component Lifecycles”

Defining instance properties

Instance properties impose constraints on concurrent execution of the different component instances. You can view and change instance properties using the Instances tab of the component's property sheet.

Note If you import a component interface, you must configure this property manually after importing the component. This information is not stored in the EAServer repository.

EAServer supports only the ActiveX single-threaded apartment model. In the single-threaded apartment model, each component instance is bound to the same thread for the lifetime of the instance. A thread is serviced by the same connection. Multiple instances may be simultaneously active on different threads. Shared stateful resources and global data should not be used.

To implement the single-threaded apartment model for an ActiveX component, enable only the Bind Thread option in the component properties Instances tab. Although most ActiveX-enabled IDEs use the single-threaded apartment model, if a component uses the ActiveX free-threaded model (in which a single method invocation can run on different threads), the component defaults to using the ActiveX single-threaded apartment model. ActiveX components developed with Power++ support the single-threaded apartment model.

Because Visual C++ 4.2 ActiveX components use nonapartment single-threading (in which multiple instances cannot be simultaneously active) by default, you must change them to use the single-threaded apartment model by:

- Not using global data, and
- Marking the component's Registry entry to indicate that the component supports the single-threaded apartment model.

The following settings specify the constraints that are placed on concurrent execution of different instances of the component. The choices are:

- *Concurrency* – Multiple invocations can be processed concurrently; that is, multiple instances can be simultaneously active on different threads. The component must be thread-safe. Use this setting if the component code uses no volatile global data and does not share stateful resources (such as a file) among instances. This threading model offers the highest performance.
- *Bind Thread* – Instances are bound to the creating thread. The component uses thread-local storage.

- *Pooling* – Instances are pooled after a commit or rollback. Instance pooling allows EAServer to recycle component instances, avoiding the overhead incurred when a new instance is created each time a component is activated.

  When deciding whether to support instance pooling, consider the following factors:

  - Instance pooling increases the efficiency of your component the most when more resources are used to initialize an instance than to clean it up. Complex structures that incur a large overhead to create are prime candidates for instance pooling. If the component does not perform a lot of initialization, it may not be more efficient for a component to use instance pooling.
  - Transactional components can benefit from instance pooling. Each time an EAServer transaction is committed or rolled back, EAServer deactivates the component instances that are involved. If your component does not support instance pooling, a new instance is required for each EAServer transaction that the component participates in.

  You might also want to implement the IObjectControl interface in place of or in conjunction with the pooling option. Implement the IObjectControl interface if you:

  - Want to determine, at runtime, whether a specific instance should be pooled (do not select the pooling option—otherwise, the CanBePooled method in the IObjectControl interface will not be called), or
  - Need to reset the component's state after deactivation.

  Only if you are coding the component in C++ can you directly implement IObjectControl.

- *Sharing* – A single shared instance services all client requests. Only one instance of the component can exist at any time. Attempts to create new instances when one already exists will fail.

Writing ActiveX components

This section describes how to write the code for ActiveX components that run in EAServer.

When you code the parameters for each method, make sure you use the ActiveX datatypes that are supported by EAServer (see "ActiveX datatype support" on page 314).

Note IDL attributes are not supported by ActiveX components.

To write code for ActiveX components:

1 Implement the IDispatch interface – ActiveX components running on a server are nonvisual, that is, they do not display text or graphics. Consequently, many commonly used ActiveX interfaces are not required for creating ActiveX components in EAServer. ActiveX components running on EAServer need to support only the IDispatch interface. If you develop your component with an automation controller such as Visual Basic, the IDispatch interface is implemented transparently.

2 Implement the constructor and destructor. See "Implementing a constructor and destructor" on page 334.

3 Optionally, implement the IObjectControl interface – You can use this interface to determine, at runtime, whether to pool instances.

4 Implement methods to perform the following optional tasks:

- Sharing data between components – Enable components to share properties between the same class's instances.
- Issuing intercomponent calls – Execute methods in other components.
- Managing database connections – Connect to databases through connection caches by using the Connection Management API.
- Sending result sets from an ActiveX component – Return result sets using the EAServer Result Sets API.
- Setting transactional state – If your component is transactional, call IObjectContext methods to set the transaction state before returning.
- Accessing SSL client certificates – If the client connected using SSL with mutual (client and server) authentication, you can retrieve the client certificate information in your component. See Chapter 8, "Using SSL in ActiveX Clients," in the *EAServer Security Administration and Programming Guide* for more information.

- Adding error-handling code – If errors occur in a method, raise an ActiveX automation exception. Add code that responds to errors by recording error details to the server log file and sending an exception to the client.

Implementing a constructor and destructor

A constructor is called when a new instance is created. A destructor is called when the instance is destroyed.

Normally, a constructor sets the object's fields to their initial value and allocates any other objects that are used by the component, and a destructor frees any objects that were allocated in the constructor.

However, if the component implements the IObjectControl interface, instance-specific initialization must be performed in the Activate method. See the IObjectControl interface reference page in the *EAServer API Reference* for more information.

Sharing data between components

EAServer allows components in the same package to share data. Shared data is organized within groups. Properties within a group can be referred to by either a string name or a numeric index. Property values are represented by an ActiveX VARIANT structure.

Note You cannot use shared variables in components that are configured for automatic failover, because these components cannot use local shared resources. See "Component properties: Transactions" on page 58 for more information. If you need to share data, you can store shared data in a remote database.

Using shared data in C++

For components implemented in C++, EAServer provides the interfaces below for sharing data between components. See Chapter 2, "ActiveX C++ Interface Reference," in the *EAServer API Reference* for descriptions of these interfaces:

- ISharedPropertyGroupManager Interface – Contains methods to create shared property groups or retrieve a handle for access to an existing group.

- ISharedPropertyGroup Interface – Represents a shared property group. Contains methods to create new properties and access existing properties.
- ISharedProperty Interface – Represents a shared property. Contains methods to get and set the property value.

Using shared data IDispatch interfaces

For components implemented using automation controllers such as Visual Basic, EAServer provides the IDispatch interfaces below. See Chapter 3, "ActiveX IDispatch Interface Reference," in the *EAServer API Reference* for descriptions of these interfaces:

- SharedPropertyGroupManager Interface – Contains methods to create shared property groups or retrieve a handle for access to an existing group.
- JagSharedPropertyGroup Interface – Represents a shared property group. Contains methods to create new properties and access existing properties.
- JagSharedProperty Interface – Represents a shared property. Contains a Value property that allows the shared property value to be retrieved and updated.

Issuing intercomponent calls

To invoke another component, use the ActiveX proxy automation server to create a proxy for the second component. See Chapter 20, "Creating ActiveX Clients" for instructions.

You must use a proxy to issue intercomponent calls. If you instantiate another component directly, EAServer transactions will not work. Also, many EAServer features such as shared objects will not work correctly in the called component. In addition, you must define the Host property as "localhost."

Managing database connections

If your ActiveX methods connect to remote data servers, you should use EAServer's connection caching feature to realize improved performance. See Chapter 26, "Using Connection Management" for more information.

Note EAServer's transactional model works only with connections obtained from the EAServer Connection Manager. Connections that you open yourself will not be affected by EAServer transactions.

Sending result sets from an ActiveX component

ActiveX methods use the IJagServerResults interface to return rows to the client. For details, see "Sending result sets from an ActiveX component" on page 470.

Setting transactional state

Transaction state is set using an IObjectContext interface pointer. The IObjectContext interface can be directly accessed only if you are coding the component in C++.

In C++, call GetObjectContext to obtain a reference to an IObjectContext object. Call the appropriate IObjectContext method to set transactional state before returning from the method:

- Call SetComplete, if the instance has completed its work without error.
- Call EnableCommit, if the work is not necessarily finished but not in error.
- Call DisableCommit, if the work is still in progress and has errors.
- Call SetAbort if the work cannot be completed.

For nontransactional components, either SetComplete or SetAbort deactivates the component instance. To keep the instance active, call DisableCommit or EnableCommit.

If a method does not explicitly set transaction state before returning, the default behavior is EnableCommit.

Adding error-handling code

Errors occurring during a method call should be handled as follows:

1 Call the IJagServer::WriteLog method to write a description of the error to the log file.

Note *JagAxWrap.dll* must be registered on your machine. If you are developing on a machine that already has EAServer installed on it, *JagAxWrap.dll* is already registered.

2 You can also generate an ActiveX automation exception with text that describes the error. EAServer returns the text of the exception to the client. Java clients receive the message as a Java exception (class com.sybase.jaguar.util.JException) and ActiveX clients receive the message as an ActiveX automation exception.

In general, if an error prevents completion of a desired task (such as database updates that represent a new sales order), you should generate an ActiveX automation exception to send a concise description of the problem to the client. Messages sent to the client should be concise and contain language suitable for display to the end user. You can record more detailed messages in the log file.

Note IDL user-defined exceptions are not supported.

Note Never write your component to send error messages to the console to display dialog boxes. Servers run unattended; showing a dialog box will do nothing but hang the thread that executes your component.

Deploying ActiveX components

To deploy an ActiveX component to EAServer:

1 Copy the ActiveX component and any other required DLLs to any directory on the EAServer machine.

2 Register the ActiveX component into the Windows Registry by entering this command from the MS-DOS Command Prompt window:

```
regsvr32 path\MyActiveXComponent
```

where:

path is the full path name to the directory where the ActiveX component resides.

MyActiveXComponent is the file name of the ActiveX component.

3 If the component interface has not been defined in EAServer Manager, import the DLL or type library into EAServer Manager. See “Importing ActiveX components” on page 328 for more information.

Most ActiveX development environments register component DLLs when they are built. If your server runs on the machine where you developed the component, you can skip steps 1 and 2.

CHAPTER 20

Creating ActiveX Clients

This chapter describes how to create ActiveX clients that execute methods on components deployed on EAServer.

Procedure for creating ActiveX clients

To create a new ActiveX client:

- Generate .tlb and .reg files for components – use EAServer Manager to generate *.tlb* and *.reg* files to use with your ActiveX-enabled IDE.
- Develop and test the ActiveX client – import the type libraries into your ActiveX-enabled IDE and use the drag-and-drop technique to add component methods into your ActiveX client code.
- Deploy the ActiveX client – install the ActiveX client, EAServer client runtime files, and component type libraries and registry files on every machine where you want to run the ActiveX client.

Generate .tlb and .reg files for components

To generate *.tlb* and *.reg* files from components in a package on a remote server, connect EAServer Manager to the remote server. EAServer Manager must be running on Windows.

Type libraries contain proxy interface metadata, which can be used to perform drag-and-drop development of the ActiveX client. Also, the APAS uses a type library to perform type checking prior to invoking methods at runtime.

Registry files contain entries for proxy objects so that they can be instantiated using the standard mechanisms in ActiveX-enabled IDEs for creating an OLE automation object. These registry entries also specify the location of the APAS, which is responsible for processing all operations invoked by an ActiveX client on the proxy object. When generating type libraries and registry files for a package, EAServer Manager can automatically register the type libraries in the local registry.

An ActiveX client can be deployed on many machines. If you deploy an ActiveX client on another machine, make sure that the *.tlb* and *.reg* files also deployed on that machine. Because the location of these files and the APAS might be different from their location on the original machine, you must use the jagreg tool to automatically update and register the *.reg* files. (See "Deploy the ActiveX client" on page 365 for details on jagreg.)

Before you start

You must have the Microsoft *uuidgen.exe* (which is provided with EAServer) and *midl.exe* tools to generate type libraries and registry import files for an ActiveX proxy. Make sure that the directory or directories containing these tools are in your path before you run EAServer Manager. See "ActiveX client requirements" on page 312 for more information about how to get *midl.exe*.

A type library cannot be updated while in use
EAServer Manager will fail to generate *.tlb*/*.reg* information if you attempt to overwrite a type library file that is in use. Exit or shut down any ActiveX client applications that use the type library before you attempt to generate an updated version. Alternatively, generate the type library to a different directory than the one that is in use, and reregister the library in the new location.

Check the ProgID for each interface

If you do not want to use the default ProgID, then you can specify your own ProgID by setting `com.sybase.jaguar.interface.com.progid` property in the Advanced tab for the interface before generating *.tlb* and *.reg* files.

The default ProgID for the proxy object follows this pattern:

```
module1_module2_module3.innerModule.
```

Each nested module name is preceded by an underscore, and the IDL type name is preceded by a period (.). For example, the ProgID for IDL type com::sybase::foo::MyBeanRemote is "com_sybase_foo.MyBeanRemote", and the ProgID for the IDL type CtsSecurity::SSLSessionInfo is "CtsSecurity.SSLSessionInfo".

If a component implements multiple interfaces, you must change the ProgID for each interface individually.

Generating TLB/REG files

To generate *.tlb* and *.reg* files:

1 Select the package from which the *.tlb* and *.reg* files will be generated.

2 Select File | Generate TLB/REG.

3 Enter the name of the output directory to store the generated *.tlb* and *.reg* files. The default is the root directory of the drive on which EAServer is installed.

4 In the Proxy Server Location, you must enter the path to the InProcServer corresponding to the Generic ActiveX Proxy DLL (*jagproxy.dll*).

Warning! Do not leave this field blank. If you do, an empty string will be inserted into the InProcServer32 entry in the Windows Registry and the ActiveX proxy will not run.

---

5 Click the Register box if you plan to run ActiveX EAServer clients on the same machine where EAServer Manager is running. This will register the ActiveX proxy interfaces. The ActiveX proxy interfaces must be registered before applications can use them.

If you do not click the Register box, the *.reg* file can be manually registered by using the system regedit tool to load it into the machine's registry. To run ActiveX clients on another machine, copy the generated *.reg* file to that machine, then use the jagreg tool to load it into the machine's registry.

After the registry file is registered, do not move the type libraries or the APAS. The registry file maintains absolute paths to these files. If you move any of these files, use the jagreg tool to register the new locations of these files.

Note The directory containing *jagproxy.dll* does not need to be in the path. The directory containing EAServer's C++ client DLLs does need to be in the path.

6 Click the Save MIDL File box if you want to retain the generated Microsoft interface definition language (MIDL) file. If *.tlb*/*.reg* generation fails, the MIDL file may be used to turn the MIDL compiler from the command line to determine the cause of the failure.

7 Click Generate.

Note If unsupported constructs or datatypes are present in your file, they are ignored. The generation succeeds and a dialog is displayed. The constructs or datatypes that were not generated to the *.tlb*/*.reg* files are displayed in *srv.log* in the EAServer installation *bin* directory. For more information about unsupported constructs and datatypes, see Chapter 19, "Creating ActiveX Components".

Files generated

Clicking Generate creates the files *PACK.TLB*, *PACK.REG*, and *PACK.IDL* (if Save MIDL File is checked), where *PACK* is the name of the selected package. Every component in the package is processed and its information stored in these files.

Develop and test the ActiveX client

To write and test code for your ActiveX client, you must be connected to a server (or have the server running on your machine) and have the ActiveX runtime files installed on your machine. To install the ActiveX runtime files, see "Deploy the ActiveX client" on page 365; if you install EAServer on your machine, you have the option to install the ActiveX runtime files as well. For more information, see the *EAServer Installation Guide*.

Before invoking methods on component instances, the client must connect to a server and instantiate the components. There are two techniques for proxy instantiation:

- Instantiating proxies using CORBA-style interfaces – This technique follows the CORBA client model. This technique is recommended for new development.
- Instantiating stub instances using the EAServer 1.1 interface – This technique uses interfaces that introduced in EAServer version 1.1. These interfaces are provided for backward compatibility with existing clients.

If you currently have ActiveX proxy automation server clients, Sybase recommends that you migrate you current ActiveX clients to use the CORBA-style so that you can take advantage of the new benefits. The following features are available to CORBA style clients and not to EAServer 1.1 style clients:

- Use the ORB, Session, Factory, and Manager objects and methods.
- Configure ORB level properties.
- Use high availability/load balancing features.
- Use most SSL features.
- Invoke Enterprise JavaBean components.

The ORB, SessionManager, and other CORBA-style interfaces are documented in Chapter 4, "ActiveX Client Interfaces," in the *EAServer API Reference*.

Instantiating proxies using CORBA-style interfaces

Proxies are local objects that allow you to call EAServer component methods as if the component were a local object in your program. Instantiate proxies using the EAServer ORB and SessionManager::Manager interfaces, as follows:

Step	What it does	Detailed explanation
1	Initialize the CORBA ORB and create an ORB reference.	“Initializing the ORB” on page 344
2	Use the ORB reference to create a Manager instance for the server.	“Creating a Manager instance” on page 347
3	Use the Manager instance to create a Session.	“Creating sessions” on page 348
4	Use the Session instance to create stub component instances.	“Creating stub instances” on page 348
5	Call the stub methods to remotely invoke component methods.	“Invoke component methods” on page 352

Note If you are using Visual Basic, before using the ORB, Session, Factory, and Manager objects in your client, create references to *JaguarORB.tlb*, *SessionManager.tlb* and *CtsSecurity.tlb* in your Visual Basic project using the standard Visual Basic mechanism.

Initializing the ORB

Before any ORB classes can be used, you must call the init method, which:

- Returns an object reference to the ORB.
- Allows you to pass initialization parameters to control the operation of the ORB. For example, you can specify the password for access to the Sybase SSL certificate database.

Initialization parameters

The ORB.init() method acceps a formatted string that can contain settings for multiple initialization parameters. Pass initialization parameters as shown in this example, which configures the `-ORBlogFile` property and the `-ORBpin` property, to specify a file name for logging errors and the Sybase SSL-certificate-database password, respectively:

```
orb.init("-ORBlogFile=d:\jagorb.log,-ORBpin=sybase")
```

As shown in the example, parameter names and values must be separated by an equals sign, ‘=’, and each name/value pair must be separated from the next with a comma and no white space.

For each initialization parameter, there is an equivalent environment variable. If the environment variable and initialization parameter are set, the value of the initialization parameter is used. Parameter and environment variable names are the same as for the C++ client ORB (see Chapter 15, “Creating CORBA C++ Clients”).

You can set any initialization parameter to a value of *none*, which overrides the value of the environment variable and sets the value to the default, if any.

You can pass the following initialization parameters to the driver class:

- ORBNameServiceURL – This parameter sets the IIOP URL to the EAServer name service. This parameter can also be set in an environment variable, JAG_NAMESERVICEURL. This parameter is used in conjunction with the EAServer name service and is specified according to the following syntax:

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

  where:

  hostname is the host machine name for the server that serves as the name server for your client. If omitted, the default host name applies.

  iiop-port is the IIOP port number for the server.

  initial-context is the initial naming context, which you set in the server property Initial Context. This can be used to set a default prefix for name resolution. For example, if you specify `USA/Sybase/`, all names that you resolve with the context are assumed to be relative to this location in the name hierarchy. When specifying the initial context, a trailing slash is optional; it is added automatically if you do not specify an initial context that ends with a slash.

  If your application uses a cluster of servers, the cluster may use multiple name servers. In this case, specify the URL (host machine name and IIOP port number) for each name server in a list separated by semicolons and no white space. Include the cluster’s initial naming context only with the last URL. For example:

  ```
  iiop://host1:9000;iiop://host2:9000/USA/Sybase/
  ```

- ORBHttp – This specifies whether the ORB should use HTTP-tunnelling to connect to the server. A setting of "true" specifies HTTP tunnelling. The default is "false". This parameter can also be set in an environment variable, JAG_HTTP. Some firewalls may not allow IIOP packets through, but most all allow HTTP packets through. When connecting through such firewalls, set this property to "true".

- ORBLogIIOP – This specifies whether the ORB should log IIOP protocol trace information. A setting of "true" enables logging. The default is "false". This parameter can also be set in an environment variable, JAG_LOGIIOP. When this parameter is enabled, you must set the ORBLogFile option (or the corresponding environment variable) to specify the file where protocol log information is written.
- ORBLogFile – This sets the path and name of the file to which to log client execution status and error messages. This parameter can also be set in an environment variable, JAG_LOGFILE. The default setting is *no log*.
- ORBCodeSet – This sets the code set that the client uses. This parameter can also be set in an environment variable, JAG_CODESET. The default setting is *iso_1*.
- ORBRetryCount – Specify the number of times to retry when the initial attempt to connect to the server fails. This parameter can also be set in an environment variable, JAG_RETRYCOUNT. The default is 5.
- ORBRetryDelay – Specify the delay, in milliseconds, between retry attempts when the initial attempt to connect to the server fails.This parameter can also be set in an environment variable, JAG_RETRYDELAY. The default is 2000.
- ORBProxyHost – Specifies the machine name or the IP address of an SSL proxy. See Chapter 11, "Deploying Applications Around Proxies and Firewalls," in the *EAServer Security Administration and Programming Guide* for more information.
- ORBProxyPort – Specifies the port number of the SSL proxy.
- ORBsocketReuseLimit – Specifies 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. The default may not be appropriate for a long-running client program that calls many methods from servers in a cluster. If sockets are reused indefinitely, the client may build an affinity for servers that it has already connected to rather than randomly distributing its server-side processing load among all the servers in the cluster. In these cases, the property should be tuned to best balance client performance against cluster load distribution. In Sybase testing, a setting of 10 to 30 proved to be a good starting point. If the reuse limit is too low, client performance degrades.

- ORBIdleConnectionTimeout – 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. A finite timeout also allows rebalancing of server load in an application that uses a cluster of servers.

Example: ORB initialization

ORB initialization is demonstrated in the following example.

```
Dim orb As ORB
Dim Manager As Manager
Dim Session As Session
Dim Factory As Factory

' Create a new ORB object
Set orb = New ORB

' Initialize the ORB instance
orb.init ("")
```

init returns an object reference to the EAServer ORB. When *orb* is deallocated or assigned a new object reference, it will be automatically released.

Creating a Manager instance

The SessionManager::Manager interface is used for interacting with a server. To create a Manager instance, you must identify a server listener using a URL of the format:

```
protocol://host:port
```

where:

- *protocol* is `iiop` or `iiops`. Use `iiops` for connections to secure `iiop` listeners.
- *host* is the server's host machine name or IP address.
- *port* is the listener's port number.

Pass the URL to the string_to_object method to convert the URL string into a Manager instance, as shown in the following example. The object returned by string_to_object must be narrowed to the SessionManager/Manager interface.

```
Dim orb As ORB
Dim Manager As Manager
Dim obj as Object
... deleted orb initialization ...
Set obj = orb.string_to_object(
     "iiop://puddle:9000")
Set Manager = obj.Narrow_("SessionManager/Manager")
...
```

Creating sessions

The SessionManager::Session interface represents an authenticated session between the client application and a server. The createSession method accepts a user name and password and returns a session object, as shown in the example below:

```
Dim orb As ORB
Dim Manager As Manager
Dim Session as Session
Dim obj as Object
...deleted manager initialization
Set obj = Manager.createSession("jagadmin","")
Set Session = obj.Narrow_("SessionManager/Session")
...
```

Creating stub instances

You call the Session.lookup method to return an object reference factory. You then use the factory to create one or more proxies for the component.

lookup takes a string that specifies the EAServer component name. By default, the name is *package/component*, where *package* is the EAServer Manager package name and *component* is the component name. Package and component names are not case sensitive. Component developers can override the default name by setting the JNDI Name property for EJB components, or the `com.sybase.jaguar.component.bind.naming` property for other types of components.

lookup returns a CORBA::Object reference. You use Narrow_ to convert the object reference into an instance of the factory for the component.

After instantiating the factory, the factory Create method returns an instance of the component proxy.

The code to instantiate a proxy for a component named *Foo/Bar* looks like this:

```
Dim Session as Session
Dim fact as Factory
Dim barComp as Bar // Component proxy
Dim obj as Object
...deleted session initialization ...
Set obj = Session.lookup("Foo/Bar")
Set fact = obj.Narrow_("SessionManager/Factory")
Set barComp = fact.Create()
```

Instantiating stub instances using the EAServer 1.1 interface

Sybase recommends that you use the CORBA style interfaces for new development. The EAServer 1.1 interface is provided for backward compatibility with existing applications.

To invoke EAServer components, your ActiveX client should:

1 Declare proxy objects – The application creates an ActiveX interface pointer for the proxy object.

2 Set connection properties – The application sets connection properties for the component instance. These properties describe the server that contains the component and the user name to be used for a connection.

3 Instantiate server components – The application calls the proxy object's Initialize method. Initialize connects to the server and creates an instance of the server component. After Initialize succeeds, the server component methods can be called through the proxy object.

4 Invoke component methods – The application invokes methods on the server, passing the appropriate ActiveX datatype for each parameter.

Declare proxy objects

Proxy objects are instantiated and invoked via ActiveX dispatch interfaces. EAServer proxy objects can be identified by their program identifier (ProgID). See "Check the ProgID for each interface" on page 340 for more information.

Different ActiveX-enabled IDEs have different mechanisms for declaring an ActiveX object. In Visual Basic, you can simply declare the proxy object and instantiate it. For example, you can write either one of the following to instantiate a proxy object:

```
Dim bar as Bar
Set bar = New Bar
```

or

```
Dim bar as Object
Set bar = CreateObject("Foo.Bar")
```

Although the ActiveX proxy object exists once you have declared it, you cannot invoke methods until after you have set connection properties and called the Initialize method.

Set connection properties

Before calling the Initialize method, set the connection properties, such as `UserName`, `Password`, `Host`, and `Name`. The ActiveX client uses connection properties to connect to the server. This example sets the connection information for the `employeeproxy` object.

```
employeeproxy.UserName = "Guest"
employeeproxy.Password = "Guest"
employeeproxy.Host = "Jaguar"
employeeproxy.Name = "Company/Employee"
```

The user name and password, which must be specified, are required for login authentication and access control. The defaults for user name and password are empty strings. If the server administrator has enabled authentication, you must use a valid user name and password. If user access to the package or component is limited, the user name must be in a group that has access to the component. For more information on security, see Chapter 2, "Securing Component Access," in the *EAServer Security Administration and Programming Guide*.

The `Host` property, which is optional, is the machine name and IIOP port number or the environment variable that specifies the machine name and IIOP port number. If the machine name and IIOP port number are specified for the `Host` property, the environment variable is ignored. See "Deploy the ActiveX client" on page 365 for more information about defining the environment variable.

The syntax for specifying the machine name and IIOP port number is:

```
"machine:port"
```

where:

machine is the machine name.

port is the IIOP port number.

Note If the `Host` property or environment variable is not specified, or defined incorrectly, the default, which is "localhost:9000", is used.

The `Name` property, which is optional, specifies the package and component names. By default, the package name is the same as the module name, and the component name is the same as the interface name. Specify the `Name` property when a component's package or component name is different from its module or interface name. The package and component are automatically located relative to the server's `Initial Context` property. The syntax for the `Name` property is:

```
"package/component"
```

where:

- *package* is the name of the package.
- *component* is the name of the component.

Note If the `Name` property is not specified, or defined incorrectly, the default is used.

Instantiate server components

To instantiate the components on the server, use the Initialize() method. Initialize() establishes a connection to the server, using the connection properties you set in the previous step. If the server host name is not valid, or if another error occurs, the APAS displays an error message. This example executes the Initialize() method on the `employeeproxy` object, and instantiates on the server an instance of the Employee component belonging to the Company package.

```
employeeproxy.Initialize()
```

Invoke component methods

"ActiveX datatype support" on page 314 lists the ActiveX types supported by EAServer, as well as the equivalent EAServer Manager and CORBA IDL types.

EAServer components appear as automation objects in the ActiveX-enabled IDE. If your IDE supports it, you can simply drag and drop the component method into your ActiveX client code and use the IDE's object browser to see the component's method syntax. You must call the proxy methods using the syntax required by your development tool.

To execute a component method, execute the method on the proxy object. In this example, the GetEmployeeInfo() and the SetEmployeeInfo() methods are executed on `employeeproxy`. The parameters in the SetEmployeeInfo() method are in parameters. The parameters in the GetEmployeeInfo() method are inout parameters.

```
String name
Long age
String sex

name = "John"
age = 32
sex = "male"

// Example for parameters using the in argument mode
employeeproxy.SetEmployeeInfo (name, age, sex)

// Example for parameters using the inout argument mode
employeeproxy.GetEmployeeInfo (REF name, REF age, REF
sex)
```

Methods may return result sets. After a method invocation, you can retrieve result sets as described in "Result-set support" on page 323.

Note If a component that the ActiveX client accesses is an ActiveX component and a C++ IDE such as Visual C++ was used to develop it, string parameter types are always passed by reference (as BSTR *). Make sure that you defined these parameters as inout in EAServer Manager.

When you invoke component methods, these restrictions apply:

- You must pass parameters in a datatype that is equivalent to the corresponding parameter's datatype in the EAServer method definition. See "ActiveX datatype support" on page 314 for more information.
- You must pass parameters by position; named arguments to method calls are not supported.
- You cannot use methods with names that differ only in case.
- The result-set parameter type is not allowed in either in or inout modes.

Code exception handling

Always make sure that your application handles exceptions gracefully. At minimum, you should display the exception text, which will aid debugging.

Errors in ActiveX proxy execution can be handled as ActiveX exceptions, or inline using a try/catch model similar to the structured exception handling model in the C++ and Java languages.

Using an ActiveX error handler

By default, the ActiveX proxy raises an ActiveX exception when an EAServer component method raises an exception or an internal error occurs. Visual Basic and most other ActiveX scripting tools do not allow you to handle these errors inline. Instead, control transfers to an error handler (specified by on error goto in Visual Basic) or to a system-wide error dialog box. To handle proxy errors inline, you must enable inline exception handling as described in "Handling exceptions inline" on page 358.

Structure of an ActiveX exception

In C++, the OLE EXCEPINFO structure describes an ActiveX exception. Different ActiveX-enabled IDEs provide different mechanisms for applications to obtain the EXCEPINFO structure contents.

In Visual Basic, exceptions are mapped to the built-in Err object. The exception number maps to Err.Number and the description is available as Err.Description. You can handle exceptions by activating error handling code with On Error Goto statement or by checking whether Err.Number is > 0.

The proxy type library defines error numbers for client-side errors in the JagORBClientErrNum enumeration and server-side error numbers in the JagORBServerErrNum enumeration.

Note IDL user-defined exceptions are not supported and are mapped to error number 9000.

Client error numbers

The following table lists the codes for client-side error numbers defined in the JagORBClientErrNum enumeration:

Table 20-1: JagORBClientErrNum error codes

Symbolic error code	Number	Description
jagClNonByteArrayErr	8000	Method arguments of type array can only have a base element type of byte.
jagClMultiDimArrayErr	8001	Multi-dimensional arrays not supported as an argument to a method.
jagClArrayRedimErr	8002	A Fatal Internal Error was encountered while attempting to resize a method argument of type array.
jagClArrayProcErr	8003	A Fatal Internal Error was encountered while processing a method argument of type array.
jagClArrayEmptyErr	8004	An array of size 0 was passed as parameter to a method.
jagClArrayBoundsErr	8005	A Fatal Internal Error was encounterd while attempting to determine the upper bound on a method argument of type array.
jagClNotJagComponentErr	8006	The component being instantiated is not a valid EAServer component or was not registered in the Windows Registry.
jagClOutOfMem	8007	The Application failed to acquire memory from the Operating System.
jagClCreateFactErr	8008	The EAServer Proxy Server could not instantiate a Factory Object. Please contact Sybase Technical Support.
jagClTypeLibErr	8009	The type library for the Component could not read from the Windows Registry. Please check if a valid directory location was specified for the Type Library while registering the component.

Symbolic error code	Number	Description
jagClTypeInfoErr	8010	The type information for the Component could not read from the Type Library. Please regenerate TLB and REG files for the component using EAServer Manager.
jagClMethInfoErr	8011	The metadata for the method or component could not be read from the Windows Registry or the method is using parameter types that are not presently supported in the EAServer ActiveX Proxy.
jagClMethNameErr	8012	The metadata for the method invoked on component could not be read from the Windows Registry. Please regenerate TLB and REG files for the component using EAServer Manager.
jagClCompNameErr	8013	The component name for the component being instantiated could not read from the Windows Registry.
jagClPkgNameErr	8014	The package name for the Component being instantiated could not read from the Windows Registry.
jagClPxyCreateErr	8015	Component creation failed.
jagClPxyDestroyErr	8016	Component deletion failed.
jagClPxyFuncDescErr	8017	The metadata information for the method could not read from the type library.
jagClArgCountErr	8018	There was a mismatch between the number of parameters passed to method and the number of parameters as described by the information in the type library.
jagClInternalErr	8019	An error was encountered while invoking an EAServer method.
jagClParamInfoErr	8020	The type information for a method parameter could not be read from the Type Library.
jagClTypeMismatchErr	8021	There is a mismatch between type of the value passed as an argument with its specified type in the Type Library.
jagClConversionErr	8022	The data conversion attempted is presently not supported.

Symbolic error code	Number	Description
jagClArgUpdateErr	8023	An error was encountered while updating an input-output or output parameter for a method.
jagClRetValSetErr	8024	An error was encountered while updating the return value for a method.
jagClRecsetArgErr	8025	The ResultSet type cannot be passed as a parameter in either the input or input-output modes by an EAServer ActiveX application.
jagClUnsuppTypeErr	8026	An unsupported OLE Automation type was used as a parameter in a method.
jagClAxConvertErr	8027	An error was encountered while converting a input-output method parameter received from the server.
jagClJagConvertErr	8028	An error was encountered while converting a input parameter prior to method invocation.
jagClNoInitErr	8029	an EAServer component instance must be created prior to invoking a method.
jagClRecordsetCreateErr	8030	An internal error was encountered while creating the Recordset object.
jagClRecordsetMoveErr	8031	Attempt to call MoveNext on a RecordSet which has its EOF property as TRUE.
jagClIteratorPosErr	8032	An invalid position was specified while attempting to retrieve an element from a collection.
jagClInvalidMethodErr	8033	The only method supported on the generic Object type is Narrow_.
jagClNarrowFailErr	8034	The object reference cannot be narrowed to the interface name specified.
jagClInvalidIntfErr	8035	The fully scoped interface name passed as an argument to the Narrow_ method is invalid.
jagClOrbInitErr	8036	An internal error was encountered while initializing client-side ORB.
jagClOrbStrToObjErr	8037	An internal error was encountered while invoking the ORB.string_to_object method.

Server error numbers

The following table lists the codes for server-side error numbers defined in the JagORBServerErrNum enumeration:

Table 20-2: JagORBServerErrNum error codes

Symbolic error code	Number	Description
jagSrvMethExcepErr	9000	The method implementation threw an user-defined exception while executing on the server.
jagSrvMethInvalidErr	9001	The method name is either invalid or is presently not defined in the component's interface.
jagSrvMethInvalidArgErr	9002	The invocation of the method on the server failed because an invalid number of parameters was passed or a parameter type mismatch occurred.
jagSrvMethNotImplErr	9003	The invocation of the method on the server failed because the component does not implement the method.
jagSrvCompPermErr	9004	The invocation of the method on the server failed because user does not have the permissions to instantiate the component.
jagSrvCompDeployErr	9005	The invocation of the method on the server failed because component implementation was not deployed on the server.
jagSrvInternalErr	9006	The invocation of the method on the server failed due a fatal internal error.
jagSrvArgCountErr	9007	The invocation of the method on the server failed because an invalid parameter type was used by the method.
jagSrvSrvConnectErr	9008	The requested operation failed since the client could not to acquire connection to the server.
jagSrvConversionErr	9009	The invocation of the method on the server failed due to a data conversion error.
jagSrvFreeMemErr	9010	The invocation of the method on the server failed while releasing memory resources.
jagSrvIntfReposErr	9011	The invocation of the method on the server failed while trying to access the interface repository.
jagSrvOutOfMemErr	9012	The invocation of the method on the server failed while trying to acquire memory from the Operating System.

Symbolic error code	Number	Description
jagSrvOutOfResErr	9013	The invocation of the method on the server failed since it could not acquire the necessary resources.
jagSrvSrvRespErr	9014	The invocation of the method on the server failed because there was no valid response from the server.
jagSrvInvObjrefErr	9015	The invocation of the method on the server failed because the object reference is invalid.

Handling exceptions inline

By default, the ActiveX proxy raises an ActiveX exception when an EAServer component method raises an exception or an internal error occurs. Visual Basic and most other ActiveX scripting tools do not allow you to handle these errors inline. Instead, control transfers to an error handler (specified by on error goto in Visual Basic) or to a system-wide error dialog box.

Inline exception handling can simplify the code that handles recoverable errors. For example, you can keep program logic that allows a user to retry a failed login in one place, rather than split into mainline code and the separate error handling code. Inline exception handling also allows you to handle errors explicitly in scripting tools that do not allow you to install user-coded error handlers.

The ActiveX proxy supports inline exception handling with Try, Catch, and End methods and an internal exception store. When an exception occurs with inline handling active, the proxy stores the error information rather than raising an ActiveX exception. Each component proxy object supports these methods and contains an exception store that is specific to that object. To handle exceptions inline, call the Try_, Catch_, and End_ methods as follows:

- **Try_** Activates inline exception handling. Errors or exceptions that occur after calling Try and before End do not raise ActiveX exceptions. Instead, the error is stored in an internal exception store that can be accessed with the Catch method.
- **Catch_** Check whether an exception of a specified type has occurred. Catch has this syntax:

  boolean Catch_(in string exceptionType, out Object exception)

Where *exceptionType* is the exception type to check for or "..." to check for the occurrence of any exception, and *exception* is an output variable. If the exception store contains an exception of the specified type, Catch_ copies the exception store to the *exception* variable, clears the exception store, and returns true.

You can call Catch_ multiple times to check for exceptions of different types.

- **End_** Deactivates inline exception handling and reverts to the standard ActiveX error handling mechanism. If the exception store contains an exception instance, End_ throws the stored exception as an ActiveX exception.

Special considerations

The Try_ and Catch_ methods do not have the same semantics of structured exception handling in Java or C++. In particular:

- Since the exception store holds only one exception instance, you must call Try_ after every proxy method invocation that can raise an exception. Otherwise, an exception in the store can be overwritten by the most recently thrown exception.
- If you return from a subroutine with inline exception handling active for an object, it remains active for that object.
- Inline exception handling in multithreaded programs requires that you use a separate copy of a proxy object in each thread. See "Using Try_ and Catch_ in multithreaded programs" on page 364 for more information.

Example: using "catch all" exception handling

When you call the Catch_ method, you can check for exceptions of a specific type, or for exceptions of any type. To check for any exception, pass "..." as the exception type parameter.

The following example illustrates this style of exception handling:

```
barcomp.Try_
barcomp.methodThatRaisesException(1007)
Dim anyExcep As Object
If (barcomp.Catch_("...", anyExcep) = True) Then
    Dim excepType as String
    excepType = anyExcep.GetExceptionType
    if (StrComp(excepType, "Foo/NotValidIdException")
== 0) then
       Dim invalidIdExcep as NotValidIdException
       set invalidIdExcep = anyExcep
       Dim id as integer
       Dim msg as String
       id = invalidIdExcep.id
```

```
        msg = invalidIdExcep.message
    Else if (StrComp(excepType, "Foo/NoAuthorizationEx
ception") == 0) then
        Dim noAuthorizationExcep as NoAuthorizationExc
eption
        set noAuthorizationExcep = anyExcep
        Dim user as String
        Dim cert as String
        user = noAuthorizationExcep.username
        cert = noAuthorizationExcep.certificate
    Else if (StrComp(excepType, "Jaguar/ClientExceptio
n") == 0) then
        Dim systemExcep as SystemException
        set systemExcep = excep
        Dim code as integer
        Dim msg as String
        code = systemExcep.code
        msg = systemExcep.message
    End if
    Else
        ' No Exception has occurred. Proceed
End If
```

Exception datatypes

Exception datatypes are used with the Try_ method when handling exceptions inline. The ActiveX proxy includes predefined system exceptions that correspond to the standard CORBA system exceptions. User-defined exceptions that are declared in an IDL module are also mapped to ActiveX types.

System exceptions In IDL, system exceptions extend the CORBA SystemException IDL type:

```
interface SystemException
{
  long code;      // numeric error code
  string message; // text error message
};
```

Unlike user-defined exceptions, a component method can throw system exceptions that are not listed in the raises clause of the IDL method signature. The C++ and ActiveX client runtime engines may also raise system exceptions when errors occur in the processing of a method invocation.

In the ActiveX proxy, system exceptions are mapped to the interface SystemException with the following properties and methods:

- The Code property specifies the numeric error code.

- The Message property specifies the text error description, if available.
- The GetExceptionType method returns the string exception identifier (see "Exception identifiers" on page 361 for more information).

The ActiveX proxy uses SystemException to represent the standard CORBA system exception types that can be returned by components, as well as errors that occur in the ActiveX proxy. "Exception identifiers" on page 361 lists the system exception types.

User-defined exceptions In IDL, user-defined exceptions are defined using syntax similar to an IDL structure. For example:

```
exception InvalidValueException
{
  string message;
  string value;
};
```

User-defined exceptions can be defined within an IDL module or interface. The IDL method signature for a component method must list user-defined exceptions thrown by the method in the raises clause. A method cannot throw user-defined exceptions that are not listed in the raises clause.

In ActiveX, the IDL exception maps to an interface with the following properties and methods:

- One get/set property for each member field in the exception, following the datatype mappings for IDL to ActiveX types.
- A GetExceptionType method that returns the string exception identifier (see "Exception identifiers" on page 361 for more information).

Exception identifiers Both system and user-defined exceptions support a GetExceptionType method that returns a string identifier for the exception. The exception identifier for a user-defined exception defined in a module is:

```
module/exception
```

Where *module* is the IDL module name and *exception* is the IDL exception type. For example, "CtsSecurity/No Certificate Exception". The exception identifier for an exception defined in an interface is:

```
module/interface/exception
```

Where *interface* is the IDL interface name.

Exception identifiers for system exceptions are predefined and listed in the following table:

Table 20-3: System exception identifiers

Identifier	Notes
Jaguar/ClientException	An error occurred internally to the ActiveX proxy. For example, you may have called a method that uses an unsupported parameter type.
CORBA/BAD_CONTEXT	
CORBA/BAD_INV_ORDER	
CORBA/BAD_PARAM	
CORBA/BAD_OPERATION	
CORBA/BAD_TYPECODE	
CORBA/COMM_FAILURE	A network error occurred. When creating a connection, this usually indicates that the server is down or you have specified the wrong listener address. When calling a method, the error may indicate a transient network fault; you can retry the method.
CORBA/DATA_CONVERSION	
CORBA/FREE_MEM	
CORBA/IMP_LIMIT	
CORBA/INTERNAL	
CORBA/INTF_REPOS	
CORBA/INV_FLAG	
CORBA/INV_IDENT	
CORBA/INV_OBJREF	
CORBA/INVALID_TRANSACTION	
CORBA/INITIALIZE	
CORBA/MARSHAL	
CORBA/NO_IMPLEMENT	The component does not implement the method that you called.
CORBA/NO_MEMORY	
CORBA/NO_RESOURCES	
CORBA/NO_RESPONSE	
CORBA/NO_PERMISSION	The user cannot access the server or a specified component.
CORBA/OBJ_ADAPTER	

Identifier	Notes
CORBA/OBJECT_NOT_EXIST	The object does not exist. This can happen if: • The component is not installed correctly on the server. For example, the component class or skeleton class cannot be loaded. • The object represents a stateful component and your reference to it has expired. Check the value of the component's Instance Timeout property, and, if needed, code your client to create another instance in response to this error.
CORBA/PERSIST_STORE	
CORBA/TRANSACTION_REQUIRED	The method you attempted to call must be called in the context of an open transaction.
CORBA/TRANSACTION_ROLLEDBACK	The method you called rolled back its transaction, or if you have started a client-managed transaction, the transaction timed out.
CORBA/TRANSIENT	
CORBA/UNKNOWN	

Example

This example calls a method CtsSecurity.SSLServiceProvider.setGlobalProperty. This method can be called to specify SSL settings for a connection to a server. For more information, see Chapter 8, "Using SSL in ActiveX Clients," in the *EAServer Security Administration and Programming Guide*.

The method signature and the exceptions raised are detailed in the following IDL:

```
module CtsSecurity
{
  interface SSLServiceProvider
  {
    string setGlobalProperty
    (
      in string property,
      in string value
    )
    raises (CtsSecurity::InvalidPropertyException,
      CtsSecurity::InvalidValueException);
  };

  exception InvalidPropertyException
  {
    string message;
```

```
    string property;
  };

  exception InvalidValueException
  {
    string message;
    string value;
  };
};
```

setGlobalProperty raises InvalidValueException if you attempt to set a property to an invalid value, and raises InvalidPropertyException if you specify a property that does not exist.

The following Visual Basic code calls setGlobalProperty and calls the Catch method to handle InvalidValueException inline. Since there is no Catch_ call for InvalidPropertyException, if this exception is thrown, it will be thrown as an ActiveX exception when End_ is called:

```
Dim ssp as CtsSecurity.SSLServiceProvider

// Assume ssp has been properly initialized

Dim ivException as CtsSecurity.InvalidValueException
// Activate inline exception handling
call ssp.Try
ssp.setGlobalProperty("qop", "An invalid value")
if (ssp.Catch_("CtsSecurity/InvalidValueException", ivException) then
  call MessageBox ("Invalid value: " & ivException.value & ". " & _
              ivException.message, , "Error");
endif
call ssp.End_
```

Using Try_ and Catch_ in multithreaded programs

If your program uses a proxy object in multiple threads and handles exceptions inline, you must call the Duplicate_ method to obtain a copy of the proxy object for use in each thread. Duplicate_ has the following syntax:

```
Object Duplicate_
```

Duplicate_ returns a proxy instance of the same type as the original.

Deploy the ActiveX client

You can deploy the ActiveX client on any number of machines. To install the ActiveX client on a client machine:

1 Install the EAServer client runtime files, including the C++ and ActiveX client options, by following the instructions in the *EAServer Installation Guide* for Windows.

2 For an ActiveX proxy automation server client, if you do not plan to specify the machine name and IIOP port number of the machine on which the server resides directly in the `Connection Host` property, you must define (in the System Properties from the Control Panel) a user environment variable for each server that the ActiveX client will invoke components on. By default, the client installer creates an environment variable JS_JAGUAR and sets its value to `localhost:9000`. The syntax for environment variable is:

   JS_JaguarServerName

   where:

   JaguarServerName is the host name used in the ActiveX client code.

   The syntax for the value of the environment value is:

   machine_name:*iiop_port#*

   where:

   machine_name is the name of the machine that the server resides on.

   iiop_port# is the IIOP port number for the server.

   For the default server, *jaguar*, on a machine, *puddle*, with the default IIOP port number, *9000*, you specify this user environment variable:

   JS_JAGUAR

   where the value for this environment variable is *puddle:9000*.

3 For an ActiveX proxy automation server client, set the JAG_LOGFILE environment variable, which specifies the log file in which initialization errors are recorded. Error messages that occur during the initialization stage are logged into a client log file. If the environment variable is not set, then the error messages in the startup phase will not be seen by the client application. For example:

   set JAG_LOGFILE=%JAGUAR%\bin\client.log

If the ActiveX proxy is running on the server, then the messages will be logged to the server log file.

4 Copy the component and package type libraries and registry files from your development machine to the client machine. The directory in which you place the files does not matter because registering the registry files specifies the type libraries location to the machine. The type library file name is the package or component name with a *.tlb* extension. The registry file name is the package or component name with a *.reg* extension.

5 Use the jagreg utility to register the APAS, component type libraries, and registry files. jagreg will also create a new file that reflects the type library and APAS DLL locations that you specify on the command line. You can use the new registry file to reregister the APAS if you change the location of the APAS DLL or type library files.

Running jagreg

To run jagreg, open an MS-DOS Command Prompt window and enter:

jagreg /d *jagproxy_dir* /f *registry_file* [/t *tlb_dir*] [/o *output*] [/nr]

or

jagreg /t *tlb_dir* /f *registry_file* [/d *jagproxy_dir*] [/o *output*] [/nr]

where:

jagproxy_dir is the directory in which the APAS DLL resides. By default, the APAS installer places *jagproxy.dll* in the APAS *dll* subdirectory. Specify this parameter if *jagproxy.dll* is in a location different from when you generated the registry file. If you are not sure what location is stored in a registry file, specify the current location of *jagproxy.dll* when you run jagreg.

tlb_dir is the directory where the type library files reside.

output is an optional path to the directory in which updated registry file(s) are written. If you don't specify an output directory, the new registry file replaces the previous file; the previous file is saved with a *.KEEP* extension.

/nr is the option that prevents the new registry files from being registered. Use this option to update the *.reg* files without immediately applying them to the Windows Registry.

registry_file is the name of the registry file that you want to change. Use wildcards to specify multiple files, for example **.reg*.

The following example updates all *.reg* files in the current directory, changing the type library location to *d:\jag_axp* and the APAS DLL location to *d:\jag_axp\dll*. *.reg* files in the current directory are updated and previous versions are saved with a *.KEEP* extension:

```
jagreg /t %JAGUAR%\dll /f *.reg /d %JAGUAR%\dll
```

Note If jagreg does not run, make sure the JAGUAR environment variable is set to the location of your EAServer installation and the PATH environment variable contains the location of the Windows *regedit.exe* tool as well as the EAServer *bin* and *dll* subdirectories.

You can use a hyphen (`-`) or forward slash (`/`) to delimit jagreg options. For example, both `-t` and `/t` are valid.

jagreg creates a new registry file from the existing registry file and:

- Replaces the `InProcServer32` entry under the `CLSID` key with the path to the APAS directory.
- Replaces the DIR entry under the TypeLib key with the path to the type library files directory.

In the registry file, the `InProcServer` entry under the `CLSID` key contains the absolute path to the *jagproxy.dll*. The `DIR` entry under the `TypeLib` key contains the absolute path to the type libraries directory.

If you move the APAS or type libraries, you must run jagreg again with the new settings.

You can run jagreg from a batch file to automate deployment of ActiveX clients. If running jagreg from a batch file, you can check for success by checking the JAGREG_STATUS environment variable. A value of 0 indicates success, and a value of 1 indicates failure.

Normally, jagreg runs silently. You can activate status tracing by setting the JAGREG_TRACE environment variable to “true” before running jagreg. With tracing enabled, jagreg prints status information to the screen as it runs.

PART 7

Web Applications

This part explains how to create Web applications with Java servlets and JavaServer Pages.

CHAPTER 21

Creating Web Applications

A Web application allows you to deploy interrelated Web content, JavaServer Pages (JSPs), and Java servlets as a cohesive unit, and configure the Web server properties required by the servlets and JSPs. EAServer's Web application model follows the J2EE and Java Servlet 2.3 specifications.

What is a Web application?

A Web application is a unit of deployment for interrelated Web content, JavaServer Pages (JSPs), and Java servlets. The Web application contains static files, servlet and JSP implementation classes, and a deployment descriptor that describes how the files, servlets, and JSPs are configured on the host server. The deployment descriptor also allows you to configure application-specific HTTP properties, such as MIME types and per-file security constraints. To tie it all together, a Web application provides an abstract naming convention for the JNDI names of database connections and EJBs.

A Web application represents a subset of the files available on a Web server. Each Web application has a **root request path** that forms a prefix for URLs that access the JSPs, servlets, and static pages. For example, *http://myhost/Finance*. Each Web application also has a **context root**, which is a directory in the server's file system where the Web application's files are deployed. In EAServer, the context root for Web application *wapp* is this directory in your EAServer installation:

```
$JAGUAR/Repository/WebApplication/wapp
```

Contents of a Web application

Web applications contain the following components.

Servlet files

Servlets are Java classes that create HTML pages with dynamic content and respond to requests from client applications that are implemented as HTML forms. Servlets also allow you to execute business logic from a Web browser or any other client application that connects using the Hypertext Transfer Protocol (HTTP). For more information on creating servlets, see Chapter 22, "Creating Java Servlets".

Web clients invoke your Web application's servlets by prepending the Web application's root request path to an alias that is mapped to the servlet. For example, the following URL invokes a servlet mapped to the alias "Account" in the application with root request path "Finance":

```
http://myhost/Finance/Account?type=add
```

JSP files and tag libraries

JavaServer Pages (JSP) allow you to embed snippets of Java code into HTML pages to create dynamic content. JSP tag libraries allow you to extend the standard HTML markup tags with custom tags backed by Java classes. See Chapter 24, "Creating JavaServer Pages" for more information on creating JSPs.

Static files

Files that provide static content for the site can be included in the Web site, including HTML, images, sounds, and so forth. You can also include Java applet files. You can configure the application's deployment descriptor to specify security constraints for static files and any unique MIME types required by your content.

Static files must be deployed to the following subdirectory in your EAServer installation directory:

```
Repository/WebApplication/web-app
```

Where *web-app* is the name of the Web application. You can include subdirectories, which are reflected in your application's URL namespace.

If you import a Web archive (WAR) file, the importer expands the application's static files to this location.

Java classes

A Web application's Java classes include the implementation class for each servlet and JSP, and any server-side utility classes used by the servlets and JSPs.

EAServer uses a custom class loader to run a Web application's servlets and classes referenced by servlet and JSP code. This feature allows hot refresh of servlets and JSPs. The custom class loader also allows each Web application to run with its own effective Java class path. To work with the custom loader and support hot refresh, you must deploy your Web application classes as described below.

Class and JAR file locations

You can deploy class files in the following locations, where *app_name* is the name of the Web application:

- *Repository/WebApplication/app_name/WEB-INF/classes* – for class files used by servlets and JSPs in the Web application.
- *Repository/WebApplication/app_name/WEB-INF/lib* – for classes contained in JAR files. All JAR files in this directory are automatically part of the Web application's effective class path.

Your Web application may use classes or JAR files that are used by Java or EJB clients and components. These files can be deployed in the EAServer *java/classes* or *html/classes* subdirectories. Classes and JAR files loaded from these locations cannot be refreshed unless added to the custom class list for the servlet or Web application. “Custom class lists for Web applications” on page 558 describes how to extend the custom class list for servlets and Web applications. You can also load JAR files from the EAServer *extensions* directory, as described in “Using Java extensions” on page 399.

Which classes are loaded by the custom loader?

In order to allow hot refresh, class references in your servlet and JSP code must be resolved by EAServer’s custom class loader. Class instances loaded by the system class loader cannot be refreshed. Class instances loaded by the custom class loader cannot be assigned to references loaded by the system class loader, or vice-versa.

Most all references will be resolved by the custom loader. The exceptions are references made with class loader calls with an explicit reference to the system class loader or another custom class loader. The following class references are all resolved by the custom class loader when they occur in servlet code:

- Classes referenced by import statements and declarations.
- Classes loaded dynamically using Class.forName(String). For example:

  ```
  obj = Class.forName("com.foo.MyClass");
  ```

- Classes loaded by explicitly calling the java.lang.ClassLoader associated with the servlet instance, which can be retrieved with this code (this refers to the servlet instance):

  ```
  ClassLoader loader = this.getClassLoader();
  ```

Code that uses the system class loader should be rewritten to use the servlet class loader when possible. The system class loader cannot load classes from the Web application *WEB-INF/classes* or *WEB-INF/lib* directories unless you add these locations to the server BOOTCLASSPATH and CLASSPATH environment variables. Classes loaded by the system class loader cannot be refreshed while the server is running.

Deployment descriptor

The application's deployment descriptor catalogs the servlets, JSPs, and files contained in the application, as well as the properties of each. The descriptor must be formatted in XML, using the DTD specified in the *Java Servlet Specification Version 2.3*. You can create a descriptor using EAServer Manager or another J2EE-compliant development tool.

EAServer maintains the deployment descriptor in two formats, the EAServer repository format, using property files, and in XML, using the standard DTD required for compatibility with the *Java Servlet Specification*. When you import a Web application from a WAR file, the XML descriptor is converted to repository format. Changes made in EAServer Manager are saved in the Repository format, and the XML descriptor is updated when the server is next refreshed or restarted.

Creating Web applications

You can create Web applications in EAServer Manager or any J2EE-compliant development tool that produces standard Web archive (WAR) files.

Using EAServer Manager, you can create a Web application that contains existing static files, servlets, and JSPs, and specifies the properties necessary for them to work together. If you are using another development tool, you can import the WAR file into EAServer Manager as described in Chapter 9, "Importing and Exporting Application Components," in the *EAServer System Administration Guide*.

If you use PowerBuilder, you can create JSPs and deploy them to EAServer in a WAR file. See the PowerBuilder *Working with Web and JSP Targets* manual for more information.

❖ **Creating a Web application in EAServer Manager**

1 Highlight the Web Applications folder and choose File | New Web Application. Enter a name for the application.

2 If necessary, create the servlets and JSPs that your application requires. For more information, see:

- Chapter 22, "Creating Java Servlets"
- Chapter 24, "Creating JavaServer Pages"

3 Configure the deployment descriptor. See “Configuring Web application properties” on page 376.

Configuring Web application properties

You can configure a Web application’s properties in EAServer Manager. If you have created a Web archive (WAR) file using another tool and imported it into EAServer, most properties are automatically set during the import process.

❖ **Displaying the Web Application Properties dialog box**

The procedures in this section require you to start with the Web Application Properties dialog box open. Display it as follows:

1 Expand the Web Applications folder, then highlight the icon that represents your application.

2 Choose File | Web Application Properties.

General properties

General properties are as follows:

- **Description** An optional text description of the Web application.
- **Distributable** Specifies whether multiple instances of the Web application can run in a distributed server environment on different servers. If you do not select this option and run the Web application in an EAServer cluster, all requests for the Web application must go to one server in the cluster. Further configuration is required for distributed Web applications, as described in “Clustered Web applications” on page 394.
- **Timeout** This option specifies how long the server should wait for each servlet’s init method to return. For any value, no client requests are serviced while the init method is running. Service requests that arrive while init is running are blocked until init returns. Clients receive browser timeout errors when attempting to execute the servlet while init is running. You can set the Timeout value to control how the server treats servlets if the init method is still running when you shut down the server or refresh the servlet. Table 21-1 describes the possible values.

Table 21-1: Initialization timeout values

Value	To indicate
-1	(The default.) init can run indefinitely, unless the server is shutdown or refreshed. If the init method is still running when the server is shutdown or refreshed, the server does not wait for init to complete before shutting down or refreshing the servlet.
0	init can run indefinitely. Sybase does not recommend this setting, because deadlocks or other hangs in the init method can cause the server to hang when shutting down or refreshing the servlet.
A positive integer.	The number of seconds to wait for init to return. If the init method is still running when the server is shutdown or refreshed, the server waits the specified time for init to return.

You can override the application-wide default for individual servlets. Display the Advanced tab in the Servlet Properties window, then set the com.sybase.jaguar.servlet.init.timeout property using the syntax in Table 21-1.

- **Destroy Timeout** EAServer calls each servlet's destroy method before shutting down or after you have refreshed or stopped the servlet using EAServer Manager. If service calls are still active, the Destroy Timeout setting specifies the number of seconds that the server should wait for the service calls to return before calling the destroy method. The default is 0, which specifies that EAServer calls destroy immediately.

  You can override the application-wide default for individual servlets. Display the Advanced tab in the Servlet Properties window, then set the com.sybase.jaguar.servlet.destroy.wait-time property to the desired number of seconds.

- **Session Timeout** This option specifies an application-wide default for the servlet Session Timeout property. Session timeouts are specified in minutes; the default is 30. A value of -1 indicates that sessions never expire. You cannot override the session timeout for individual servlets.

- **Context Path** The request-path prefix that clients use in URLs to access your Web application's static content, servlets, and JSPs. For example, if you enter "estore," users access your Web application with the prefix:

  ```
  http://host:port/estore/
  ```

  The default context path is the name of your Web application.

- **Client Session Persistent** This property determines whether the cookies used to store servlet and JSP session data is stored in persistent or temporary cookies. By default, session data is stored in temporary cookies that expire when the browser is shut down. When you select this option, EAServer sends a persistent cookie that expires when the Web application session-timeout setting expires. This property affects only the cookies that EAServer creates to store session data for the Web application (available to servlets and JSPs via request.getSession). It does not affect cookies created explicitly by servlets and JSPs.

Context initialization properties

All servlets and JSPs in a Web application share a common set of context initialization properties specified by the deployment descriptor. Servlet code can retrieve the values by calling the getInitParamers() and getInitParameterNames() methods in interface javax.Servlet.ServletContext.

Environment properties can be used for the same purpose as context-initialization properties, and allow additional datatypes besides java.lang.String. See “Environment properties” on page 387 for more information.

❖ **Configuring context initialization properties**

1 Display the Context Params tab in the Web Application Properties dialog box.

2 A list of properties and values appears. You can create, modify, and delete properties as follows:

- To define a new property, click Add. Edit the Name and Value fields in the new row. You can optionally enter text in the Description field to describe the intended use.
- To modify a property, put the cursor in the Name or Value fields, then edit the text.
- To delete a property, put the cursor in the Name or Value fields and click Delete.

Welcome and error page specifications

You can customize the list of welcome files and error-response files in your application. These settings take effect when Web clients are browsing in your Web application's subset of the server's URL namespace.

Welcome files

Welcome files are used to satisfy HTTP requests that end in a directory name, rather than specifying the full path to a file or a path that is mapped to a servlet invocation. For each request that maps to a directory, the server searches the directory for files that occur in the Web application's list of welcome files, in the listed order. For example, if the welcome-file list is "index.html, index.htm, welcome.jsp", the server looks for *index.html*, then *index.htm*, then *welcome.jsp*. If the server finds a static file on the welcome-file list, the server returns its content. If a JSP on the welcome-file list exists, the server invokes the JSP. If no match exists in the directory, the server returns an HTTP 404 (file not found) error, because EAServer does not support directory listings.

❖ **Adding a welcome file**

1 Display the File Refs tab in the Web Application Properties dialog box.

2 Click Add. A new row appears in the list of welcome files.

3 Place the cursor in the new row, and enter the name of the welcome file. Welcome files are plain files, without path information. You can prepend a directory separator (/), which will be ignored. For example, */index.html* is the same as *index.html*.

❖ **Deleting a welcome file**

1 Display the File Refs tab in the Web Application Properties dialog box. The welcome-file list displays.

2 Place the cursor in the row to be deleted, then click Delete.

Error pages

Error pages allow you to customize the response that the server sends to Web clients when an error occurs. You can specify HTML files to send in response to HTTP error codes and to Java exceptions thrown in JSPs or servlets. You can also define error pages at the server level. If your Web application does not specify an error page, EAServer invokes the corresponding server-level error page. To specify server level error pages, set the server property com.sybase.jaguar.server.servlet.error-page on the Advanced tab in the Server Properties dialog box. For information on this property, see the reference page in Appendix B, "Repository Properties Reference," in the *EAServer System Administration Guide*.

When an exception is thrown, the servlet engine will search the error page mappings for the exception and its super classes. For example, assume AException extends BException and BException extends CException and CException extends java.lang.Exception. When AException is thrown, EAServer checks if AException is mapped. If not, EAServer checks if BException is mapped, and so forth.

❖ **Adding an error page**

1 Display the File Refs tab in the Web Application Properties dialog box.

2 Under Error Mapping, click Add. A new row is added to the mapping table with default settings.

3 Place the cursor in the Error/Exception cell, and type the HTTP error number or Java exception class name.

4 Place the cursor in the URL cell, and type the path to the file relative to the Web application's context root. For example, */etc/error404.html*.

5 Verify that the file exists in your EAServer installation directory and can be read by the server process. For example, the path `/etc/error404.html` corresponds to this file in your EAServer installation directory, where *web_app* is the name of the Web application:

```
Repository/WebApplication/web_app/etc/error404.html
```

You can also specify server-level error pages—see "Error pages" on page 411.

Tag library descriptor references

JSPs can use tag libraries to serve content formatted with custom tags. The tag library is a Java class with methods to parse content that is tagged with custom tags and output formatted content to be returned in the response stream. Each tag library must have a Type Library Descriptor (TLD) file that describes the available tags and specifies the corresponding Java classes and methods.

JSPs use a type library by specifying the location of the TLD file as a URL. In your Web application, you can specify a mapping so that TLD URLs in JSPs map to a local URL. For example, you may refer to a tag library as follows in a JSP:

```
<%@ taglib uri="/example.tld" prefix="ex" %>
```

This path can be mapped to another location, such as:

```
/WEB-INF/tlds/PRlibrary_1_4.tld
```

You do not have to map TLD URLs in the Web application. If there is no mapping that matches a TLD URL, EAServer loads the file at the URL specified in the JSP and raises an error if the file does not exist.

Mapping TLD URLs provides several benefits such as:

- You can keep TLD files together in a common location.
- You can avoid multiple copies of a TLD when JSPs use different paths to refer to the same type library.
- You can code JSPs with simple paths, such as *tlds/example.tld*, while the actual TLD is stored in a versioned directory tree. For example, you can alias *tlds/example.tld* to *WEB-INF/tlds/example/v1.6/example.tld*. This mapping allows you to easily test new versions and roll back to previous versions if a problem occurs.

In an XML deployment descriptor, TLD URL mappings are specified by taglib elements.

Tag library classes
A Web application's tag library classes must be deployed in the *WEB-INF/lib* or *WEB-INF/classes* directories, with the other Java classes required by your Web application. See "Java classes" on page 373 for more information.

❖ **Configuring TLD mappings in EAServer Manager**

1 Display the Advanced tab in the Web Application Properties dialog box.

2 If necessary, add an entry for the property `com.sybase.jaguar.webapplication.taglib`. Otherwise, modify the existing value for this property.

3 In the property value, specify each mapping as follows:

```
(taglib-uri=alias, taglib-location=real-path)
```

Where *alias* is the path used in JSP source code, and *real-path* is the TLD file's location relative to the Web application's context root.

If multiple mappings are required, separate each by a comma. For example (the following must be entered without line breaks or carriage returns):

```
(taglib-uri=taglib.tld, taglib-location=TLD/abctaglib.tld),
(taglib-uri=lib2.tld,taglib-location=TLD/lib2v2.tld)
```

Naming references

Web applications allow you to use logical names for JNDI lookups in your servlet and JSP code. Logical names allow your application to run in environments where the JNDI name space does not match the names hard coded in your application. When deploying an application, you can map the logical names to actual names that match the server's configuration.

When developing an application, you must use JNDI to obtain database connections, mail sessions, and EJB proxies. You must catalog the JNDI names used by your code in the application's deployment descriptor.

All logical JNDI names used in your application must be prefixed with *java:comp/env*. The J2EE specification requires the following hierarchy, based on resource type:

- *java:comp/env/ejb* for EJB references
- *java:comp/env/jdbc* for JDBC javax.sql.DataSource references
- *java:comp/env/mail* for JavaMail session references
- *java:com/env/url* for java.net.URL references
- *java:com/env/jms* for javax.jms references

EJB references

Servlets and JSPs use EJB references to instantiate proxies for EJB home interfaces. See Chapter 8, "Creating Enterprise JavaBeans Clients," for more information. EJB references must be cataloged in the deployment descriptor so that the Web application can run independent of a specific naming configuration. When deploying the Web application, a site administrator can specify site-specific EJB JNDI names.

Servlets and JSPs can look up an EJB by specifying the reference name prefixed with *java:comp/env/*. For example, if you enter *ejb/catalog* in EAServer Manager, use *java:comp/env/ejb/catalog* in your JSP or servlet source code.

To add or configure an EJB reference, open the Web Application Properties dialog box.

Note The EJB References tab configuration is the same for Web applications, application clients, and EJB components.

❖ **Adding an EJB reference**

1 Display the EJB References tab.

2 Click Add. A reference with default settings is created. Edit the settings as described below.

❖ **Editing an EJB reference**

1 If necessary, display the EJB References tab. Existing references are displayed as a list with one row for each reference.

2 Edit the reference fields of interest as follows:

- **Name** Specifies the JNDI name used in your code to refer to the called EJB. The aliased name is displayed in the Link Value field. Enter the part of the JNDI name that begins with `ejb/`. For example, if your code refers to *java:comp/env/ejb/MyBean*, enter `ejb/MyBean`.
- **Type** Choose Session for session Beans or Entity for entity Beans.
- **Home** The Java class name of the EJB home interface, specified in dot notation. For example, `com.sybase.MyBeanHome`.
- **Remote** The Java class name of the EJB remote interface, specified in dot notation. For example, `com.sybase.MyBeanRemote`.

- **Link Value** The actual JNDI name EJB component that is installed in the server where your component, Web application, or application client is to be deployed. This must match the JNDI name property in the Component Properties of the called EJB component.

  For invocations of components on remote servers, you can also specify a corbaname interoperable naming URL, as described in "Interoperable naming URLs" on page 157.

3 To delete a reference, click anywhere in the fields for the reference of interest and click Delete.

EJB local references

To access an EJB's local interface, define an EJB local reference. Local interfaces are available only to EJB components, Java servlets, and JSPs hosted on the same server as the target component.

❖ **Adding an EJB local reference**

1 Display the EJB Local References tab.

2 Click Add. A reference with default settings is created. Edit the settings as described below.

❖ **Editing an EJB local reference**

1 If necessary, display the EJB Local References tab. Existing references are displayed as a list with one row for each reference.

2 Edit the reference fields of interest as follows:

- **Name** Specifies the JNDI name used in your code to refer to the called EJB. The aliased name is displayed in the Link Value field. Enter the part of the JNDI name that begins with `ejb/`. For example, if your code refers to *java:comp/env/ejb/MyBean*, enter `ejb/MyBeanLocal`.
- **Type** Choose Session for session Beans or Entity for entity Beans.
- **Home** The Java class name of the EJB local home interface, specified in dot notation. For example, `com.sybase.MyBeanLocalHome`.
- **Local** The Java class name of the EJB local interface, specified in dot notation. For example, `com.sybase.MyBeanLocal`.

- **Link Value** The actual JNDI name of the EJB component that is installed in the server where your component or Web application is to be deployed. This is specified by the JNDI Name property in the Component Properties of the called EJB component.

3 To delete a reference, click anywhere in the fields for the reference of interest and click Delete.

Resource references

Resource references are used to obtain connector and database connections, and to access JMS connection factories, JavaMail sessions, and URL links.

To add or configure a resource reference, open the Web Application Properties dialog box.

Note The Resource References tab configuration is the same for Web applications, application clients, and EJB components.

❖ **Adding a resource reference**

1 Display the Resource References tab.

2 Click Add. A reference with default settings is created. Edit the settings as described below.

❖ **Editing a resource reference**

1 If necessary, display the Resource References tab. Existing references are displayed as a list with one row for each reference.

2 Edit the reference fields of interest as follows:

- **Name** The partial JNDI name used in servlet and JSP code. Use the prefix *mail/* for JavaMail references, *jdbc/* for data source references, *url/* for java.net.URL references, and *jms/* for javax.jms references. For example, if your code refers to *java:comp/env/jdbc/MyDatabase*, enter `jdbc/MyDatabase`.
- **Type** Choose the type of resource:
  - javax.sql.DataSource for JDBC connections. See "JDBC DataSource lookup" on page 485 for more information.
  - java.mail.Session for JavaMail sessions. See Chapter 35, "Creating JavaMail" for more information.

- java.net.url for aliased URLs.
- javax.jms.QueueConnectionFactory for JMS queue connection factories. See “Looking up a ConnectionFactory object” on page 570 for more information.
- javax.jms.TopicConnectionFactory for JMS topic connection factories. See “Looking up a ConnectionFactory object” on page 570 for more information.

- **Sharing Scope** Choose Sharable or Unsharable. By default, connections to a resource manager are sharable across EJBs in an application that use the same resource in the same transaction context.

  Note This is available only to Web applications and EJB components.

- **Authentication** Select Container or Application.
- **Resource Link** Specify the resource link for the resource type:
  - javax.sql.DataSource – select the name of the EAServer connection cache or connector to be used for this resource.
  - java.mail.Session – specify the SMTP mail server for outgoing mail.
  - java.net.url – enter the URL string, as it would be used to construct a java.net.URL instance by calling the URL(java.lang.String) constructor. URLs must contain a protocol and host address, for example: `http://www.sybase.com` or `ftp://pub.sybase.com`.
  - javax.jms.QueueConnectionFactory – select the name of the queue connection factory.
  - javax.jms.TopicConnectionFactory – select the name of the topic connection factory.

3 To delete a resource reference, click anywhere in the fields for the resource reference of interest and click Delete.

Resource environment references

Resource environment references are logical names applied to objects administered by EAServer, which can be accessed by Web applications, application clients, and EJB components.

To add or configure a resource environment reference, open the Web Application Properties dialog box.

Note The Resource Environment References tab configuration is the same for Web applications, application clients, and EJB components.

❖ **Adding a resource environment reference**

1 Display the Resource Environment References tab.

2 Click Add. A reference with default settings is created. Edit the settings as described below.

❖ **Editing a resource environment reference**

1 If necessary, display the Resource Environment References tab. Existing references are displayed as a list with one row for each reference.

2 Edit the reference fields of interest as follows:

- **Name** The partial JNDI name used in servlet and JSP code. Use the prefix *jms/* for JMS reference. For example, if your code refers to *java:comp/env/jms/MyQueue*, enter `jms/MyQueue`.
- **Type** Choose the type of resource:
  - javax.jms.Queue for JMS message queues.
  - java.jms.Topic for JMS message topics.
- **Link Value** If the resource type is javax.jms.Queue, enter the name of a configured queue; if the resource type is javax.jms.Topic, enter the name of a configured topic.

3 To delete a resource environment reference, click anywhere in the fields for the reference of interest and click Delete.

Environment properties

Environment properties allow you to specify global read-only data for use by servlets and JSPs in the Web application.

Servlets and JSPs must use JNDI to retrieve environment properties, using the prefix `java:comp/env` in JNDI lookups. Unlike context initialization properties, environment properties can have datatypes other than java.lang.String.

The deployment descriptor catalogs the environment properties used by your servlets and JSPs, as well as each property's Java datatype and default value. Deployers can tailor the values to match a server's configuration. For example, you may have environment properties to specify the name of a logging file, or to tune cache usage.

To add or configure an environment property, open the Web Application Properties dialog box.

Note The Environment tab configuration is the same for Web applications, application clients, and EJB components.

❖ **Adding an environment property**

1 Display the Environment tab.

2 Click Add. EAServer Manager creates a new entry with default settings. Edit the settings as described below:

❖ **Editing an environment property**

1 If necessary, display the Environment tab. A list of environment properties appears.

2 Edit the fields for the property of interest:

- **Entry** The environment property's JNDI name, relative to the *java:comp/env* prefix.
- **Type** Choose the Java datatype that matches the property value from the dropdown list.
- **Value** The initial or post-deployment value of the property, specified as text in a format that is valid for the specified datatype.
- **Description** Optionally enter a comment to explain how the property is used.

Request path mappings

Your application's deployment descriptor must specify the request path mappings for the application's servlets and JSPs. You can map full paths, partial paths, or file extensions to servlets. Path mappings are specified relative to the application's root request path.

To map request paths to a JSP, the JSP must be defined in EAServer Manager as a Web component. See Chapter 24, “Creating JavaServer Pages,” for more information.

EAServer uses the precedence rules defined in the Servlet 2.3 specification to evaluate each URL:

1 EAServer checks whether a mapping uses the exact path.

2 EAServer checks whether a directory in the path is mapped to a servlet, starting at the most deeply nested directory in the path, and working back using the forward-slash character (/) as a separator. For example, if the application’s root request path is *MyApp* and the URL path is *MyApp/Accounts/Manage/add.jsp*, EAServer checks for servlets mapped to */Accounts/Manage*, then */Accounts*.

3 If the last node in the path contains an extension, EAServer checks for a servlet mapped to that file extension. A file extension is defined as the part of the URL that follows a ‘.’ occurring after the last ‘/’ in the URL. For example, in the path *MyApp/Accounts/Manage/add.calc*, the extension is *calc*.

4 If neither of the previous two rules results in a match, EAServer invokes the application’s default servlet if defined. The default servlet is mapped to the path /. If no default servlet is defined, EAServer looks for a static file matching the path.

Implicit JSP mapping
The *jsp* extension is implicitly mapped to invoke EAServer’s JSP engine. You can override this mapping in the explicit mappings for your Web application by mapping **.jsp* to a servlet or JSP. However, if you do so, there is no way to invoke the EAServer JSP engine to compile and run arbitrary JSP files. Explicit **.jsp* mappings are not recommended.

❖ **Adding a request path mapping**

1 Display the Servlet Mapping tab in the Web Application Properties dialog box.

2 Click Add. A new mapping appears with default settings. Edit the settings as described below.

❖ **Editing a request path mapping**

1 If necessary, display the Servlet Name tab in the Web Application Properties dialog box. A list of mappings appears, formatted as a table. You can edit any mapping by editing the text directly within the table cells.

2 Edit the servlet name and mapped path, using the following rules to format the path specification:

- All mappings are relative to the Web application's root request directory.
- To map a directory, enter a path that ends in a '*', for example `/foo/*` or `/foo/stuff/*`.
- To map an extension, enter `*.`***`ext`***, where *ext* is the extension.
- To specify a default servlet for the application, enter the path as a single forward slash (`/`).
- To specify an exact match, enter the full path relative to the Web application's root request directory.

MIME mappings

A file's MIME type specifies how a server or browser should interpret the file. For example, whether the file contains plain text, formatted HTML, an image, or a sound recording. In a Web server, MIME mappings specify how a static file should be interpreted by mapping file extensions to MIME types. MIME mappings affect only static files. Servlets and JSPs must be coded to specify a MIME type for their response.

For more information on MIME types, visit:

```
http://www.oac.uci.edu/indiv/ehood/MIME/MIME.html
```

EAServer includes preconfigured MIME mappings that you can customize using your Web application's properties. Web application MIME mappings override EAServer's preconfigured mappings.

❖ **Adding a MIME mapping**

1 Display the MIME Mapping tab in the Web Application Properties dialog box.

2 Click Add.

❖ **Editing a MIME mapping**

1 If necessary, display the MIME Mapping tab in the Web Application Properties dialog box. The configured mappings display.

2 Edit the fields as appropriate:

- **Extension** The file extension for files of this type.
- **MIME Type** The MIME specification, for example, `text/plain` or `text/sgml`.

JAXP properties

Configures the default JAXP, DOM, and XSLT parser implementations used by the Web application. See Chapter 36, “Configuring Java XML Parser Support,”for more information on these properties.

Java Classes properties

The Java Classes tab allows you to add classes and JAR files to the Web application’s custom class list. The custom class list specifies which Java classes must be reloaded when the Web application is refreshed. Chapter 30, “Configuring Custom Java Class Lists,” describes how to configure this setting.

Extensions properties

The Extensions tab in the Web Application properties dialog box configures dependencies on Java extensions. These settings provide a mechanism to formally declare the Java extensions required by the Web application, and to verify that required extensions are available in EAServer. “Using Java extensions” on page 399 describes these settings in detail.

Additional files

The Additional Files tab in the Web Application properties dialog box configures the `com.sybase.jaguar.webapplication.files` property, which specifies additional files that are to be archived when the Web application is exported or replicated to another server with the synchronize feature. By default, the file set includes the Web application's context root directory and its contents.

The rules for setting this property are the same as for the com.sybase.jaguar.component.files component property. See "Component properties: Additional Files" on page 69 for more information.

Security properties

Configures user authentication for the Web application and allows you to configure authorized access to URLs served by the Web application. Chapter 3, "Using Web Application Security," in the *EAServer Security Administration and Programming Guide* describes how to configure these properties.

Page Caching properties

You can use dynamic page caching to improve the response time for servlets and JSPs in your Web application. The properties on this tab allow you to configure default caching options for Web components that have caching enabled. For more information, see "Dynamic page caching" in Chapter 5, "Web Application Tuning,"in the *EAServer Performance and Tuning Guide*.

Listener properties

EAServer's implementation of application lifecycle events enables you to register event listeners that can respond to state changes in a Web application's ServletContext and HttpSession objects. See "Application lifecycle event listeners" on page 435 for more information.

❖ **Adding a listener**

1 Display the Listeners tab in the Web Application Properties dialog box.

2 Click Add. This adds a new row to the list of Listeners.

3 Enter the listener class name.

4 To modify the order in which EAServer notifies the listeners, highlight a listener name and click Move Up or Move Down until it is positioned correctly.

Filter Mapping properties

A filter is a Java class that is called to process client requests or the server's response. Filters can be used to modify the request header or the content of a servlet request or response. Chapter 23, "Using Filters and Event Listeners," describes how to create filters.

Filters can be mapped to a URL or a servlet name. When a filter is mapped to a URL (path-mapped), the filter applies to every servlet and JSP in the Web application. When a filter is mapped to a servlet name (servlet-mapped), it applies to a single servlet or JSP. The path-mapped filters are executed first, followed by the servlet-mapped filters.

❖ **Mapping a filter**

1 Display the Filter Mapping tab in the Web Application Properties dialog box.

2 Click Add. This adds a new row to the Filter Mapping list.

3 Enter the filter properties:

- Filter – logical name for the filter.
- Target – servlet class name or the URL string.
- Target Type – choose either Servlet Name or URL Pattern.
- Description – brief description of the filter's purpose.

Clustered Web applications

Web applications can be distributed by deploying them to an EAServer cluster. A distributed Web application can provide better performance since multiple machines can handle more load than one. Clusters also provide high availability: if one machine goes off-line, clients can connect to another server in the cluster. For more information on EAServer clusters, see Chapter 6, "Clusters and Synchronization," in the *EAServer System Administration Guide*.

Requirements

To deploy your Web application in a cluster, you must have a mechanism to support load balancing (and optionally failover), configure a mechanism to replicate HTTP session data between servers in the cluster, and make sure your code supports distributed deployment.

Load balancing and failover

Since the HTTP protocol does not support failover and load-balancing, you must configure a system to redirect client requests that use one logical host name to the cluster servers. You can do this using one of the following solutions:

- Use the EAServer Web server redirector plugin, running in Apache or another supported Web server. For information on this option, see the *EAServer Installation Guide* for your platform. This option allows load-balancing, but not high availability. The Web server can be a single point of failure in your configuration.
- Use Round Robin DNS (RRDNS). RRDNS is a standard feature in many operating systems, and no extra hardware is required. RRDNS allows HTTP requests to be routed in a round-robin fashion to different Web servers. For information on the advantages and disadvantages of RRDNS, see the O'Reilly article Web Applications Load Balancing at http://www.onjava.com/pub/a/onjava/2001/09/26/load.html. RRDNS provides load balancing, but not high availablility. The RRDNS service can be a single point of failure.
- Use another third-party address-redirection system that performs HTTP load-balancing and failover, such as:
  - The BIG-IP hardware load redirector, from F5 Networks at http://www.f5.com/

- The Local Director hardware load redirector, from Cisco Systems at http://www.cisco.com/
- If you have a cluster of Windows 2000 Advanced servers, you can use built in features for network load balancing across a cluster. Similar functionality is available for NT. For more information, see the Microsoft MSDN article Building a Highly Available and Scalable Web Farm at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnduwon/html/d5nlb.asp.

Session data replication

If a Web application is distributed and running in a cluster, EAServer replicates session data to all servers in the cluster, using one of the following mechanisms:

- *Persistent storage*: EAServer stores stores session data in a persistent data store to support shared sessions and session failover.
- *In-memory replication*: EAServer replicates session data between pairs of servers, each of which acts as a backup for the other. This feature can improve performance by avoiding the overhead of writing to the database.

You must configure one of these options, as described below.

Coding considerations

No changes are required to your servlet and JSP implementation code to support distributed sessions, as long as:

- You are managing session data using the servlet session APIs or some other mechanism where storage is not tied to the host server (such as an EJB session or entity bean).
- You use a database (or an EJB entity bean that connects to a database) to store global data. You can use the Web application's environment properties to store global read-only data.

Since session data is bound to a single user, you cannot use sessions to store global read-write data. Many applications use ServletContext properties to store global data, but the ServletContext is not global to a distributed application and cannot be used as a read-write shared-memory store.

Configuring in-memory session replication

EAServer can distribute HTTP session data using in-memory replication rather than database storage. This feature can improve performance by avoiding the overhead of writing to the database. This mechanism uses the mirror-pair replication model described in "Requirements for in-memory stateful failover" on page 549.

❖ **Enabling in-memory replication for a Web application**

These steps must be performed in EAServer Manager, while connected to the primary server for your EAServer cluster:

1 Select the Distributed checkbox on the General tab in the Web Application Properties dialog box.

2 Install the Web application to one or more logical servers that are part of the cluster.

3 Configure mirror pairs for the cluster as described in "Cluster configuration for in-memory failover" on page 550.

4 On the All Properties tab, set the com.sybase.jaguar.webapplication.distribute.type property to "inmemory".

5 Synchronize the cluster.

❖ **Changing the cache size**

The default cache size and entry time out values are unlimited. To change these settings:

1 In the master server installation for your cluster, create the directory *ObjectCache* in the *Repository* directory your EAServer installation if it does not exist.

2 In this directory, create a text file named *HttpSessionCache.props* if it does not exist.

3 Edit *HttpSessionCache.props* in a text editor, and enter the following lines:

```
com.sybase.jaguar.objectcache.name=HttpSessionCache
com.sybase.jaguar.objectcache.size=size
com.sybase.jaguar.objectcache.timeout=timeout
com.sybase.jaguar.objectcache.sync=mirror
```

Where *timeout* is the timeout value, in seconds, and *size* is the size in megabytes, kilobytes, or bytes with the syntax shown in the following table:

Syntax	To indicate
*n*M or *n*m	*n* megabytes, for example: `512M`
*n*K or *n*k	*n* kilobytes, for example: `1024K`

Syntax	To indicate
n	*n* bytes, for example: `536870912`

4 Synchronize the cluster to apply the changes to other member servers.

Configuring persistent session storage

When using this option to replicate session data, EAServer stores all session data in a remote database, connecting through the predefined JDBC connection cache ServletPersistenceCache. All servers in the cluster share the same database. Sybase recommends that you configure this cache to connect to an enterprise-grade database server. The database cannot be shared between servers that are not running in the same EAServer cluster.

The sample ServletPersistenceCache properties must be changed
As preconfigured, the ServletPersistenceCache connects to the sample database that is included with the EAServer sample applications. This sample uses the evaluation version of Adaptive Server Anywhere, which does not allow connections from multiple hosts. You must use another database that allows connections from multiple hosts, and supports the number of connections required by your cluster.

❖ **Configuring persistent session storage**

These steps must be performed in EAServer Manager, while connected to the primary server for your EAServer cluster:

1 Select the Distributed checkbox on the General tab in the Web Application Properties dialog box.

2 Install the Web application to one or more logical servers that are part of the cluster.

3 Configure the properties of the connection cache named ServletPersistenceCache to connect to the database that you use for persistent session storage. This cache must use JDBC and have cache-by-name access allowed. See Chapter 4, “Database Access,” in the *EAServer System Administration Guide* for instructions.

4 Make sure the ServletPersistenceCache cache is installed in each logical server where the Web application is installed.

5 If using a database other than Sybase Adaptive Server Enterprise or Adaptive Server Anywhere, create a table as described in "Creating the database table" on page 398.

6 Synchronize the EAServer cluster to propagate the configuration changes to other servers in the cluster. See Chapter 6, "Clusters and Synchronization," in the *EAServer System Administration Guide* for more information.

Creating the database table

If you are storing session data in a database other than Sybase Adaptive Server Enterprise or Adaptive Server Anywhere, you must manually create the table that stores the session data. Create a table named ps_HttpSession with the following schema:

Column	Data format
ps_key (primary key).	Variable length binary, 255 bytes maximum length, cannot be null.
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.

The following table definitions can be used for creating an Oracle 8.1.7 database:

```
PS_KEY   RAW (255)    NOT NULL,
PS_SIZE  NUMBER       NOT NULL,
PS_BIN1  RAW (255),
PS_BIN2  RAW (255),
PS_BIN3  RAW (255),
PS_BIN4  RAW (255),
PS_DATA  LONG RAW
```

Using Java extensions

In Java, an *extension* is a formally described set of related classes that extends the functionality offered by the base Java platform or by a J2EE application server. Extensions are packaged as Java JAR files, and include additional information in the JAR file's *manifest.mf* file to describe the extensions characteristics, such as:

- The vendor, name, and version for the specification that is implemented by the package (for example, Sun Microsystems JavaHelp 1.3)
- The vendor, name, and version for the implementation of the specification.

For more information on Java extensions, see the the Java documentation at http://java.sun.com/j2se/1.3/docs/guide/extensions/index.html.

In accord with the Servlet 2.3 specification, EAServer allows you to install extensions and define the extensions required by a Web application. If you import a Web application (in WAR format) that requires extensions that are not installed, EAServer Manager warns you of the unfulfilled dependencies.

Installing extensions in EAServer

In your EAServer installation, installed Java extensions are stored in the *extensions* subdirectory. All Web applications in one EAServer installation have access to the same set of installed Java extensions. In EAServer Manager, you can manage Java extensions from the top-level Web Applications or from the Installed Web Applications folders for any application or server.

❖ **Viewing installed extensions in EAServer Manager**

1 Highlight the top-level Web Applications folder, or the Installed Web Applications folder in your application or server.

2 Choose File | View Installed Extensions

3 EAServer Manager displays the list of installed extensions. “EAServer Java extension properties” on page 401 describes the fields.

❖ **Installing a new extension using EAServer Manager**

1 Make sure the extension JAR file is accessible on the machine where you are running EAServer Manager

2 Follow the steps in “Viewing installed extensions in EAServer Manager” on page 399 to display the Installed Extensions dialog box.

3 Click Add, then specify the full path to the extension JAR file.

4 EAServer Manager verifies that the extension’s *manifest.mf* file is formatted correctly, and if so, copies the extension JAR file to the EAServer *extensions* subdirectory.

Defining required extensions for Web applications

You can define the extensions required by your Web applications in EAServer Manager or in the *manifest.mf* file bundled within a Web application archive (WAR) file. Doing so increases the portability of the Web application among J2EE servers from different vendors. When a server imports a WAR file that specifies required extensions, it checks that the required extensions are available.

Defining required extensions in EAServer Manager

Follow this procedure if you are defining or modifying a Web application in EAServer Manager. When you export the Web application in WAR format, EAServer Manager includes the dependency information.

❖ **Defining required extensions in EAServer Manager**

1 Display the Web Application properties dialog box.

2 Display the Extensions tab.

3 Click Add to create a new extension in the list, then edit the fields described in Table 21-2 on page 401.

EAServer Java extension properties

Table 21-2 describes the fields in EAServer Manager's Installed Extensions dialog box and the corresponding entries in the *manifest.mf* file within an extension JAR file.

Table 21-2: Java Extension Properties

EAServer Manager field	Manifest entry	Description
Extension Name	`Extension-Name`	The extension name.
Specification Version	`Specification-Version`	The version number of the specification that the extension conforms to.
Specification Vendor	`Specification-Vendor`	The company or organization responsible for the specification that the extension conforms to.
Implementation Version	`Implementation-Version`	The implementation version number.
Implementation Vendor	`Implementation-Vendor`	The company or organization responsible for the implementation.
Implementation Vendor ID	`Implementation-Vendor-ID`	A unique identifier for the company or organization responsible for the implementation. Usually follows the reverse-domain naming convention used in Java packages, for example, "com.sybase."
Implementation URL	`Implementation-URL`	A Web URL to obtain information on the implementation.

Defining required extensions in the WAR manifest file

If you are creating Web applications outside of EAServer Manager, you must specify required Java extensions by adding entries to the manifest file within the WAR (path *META-INF/MANIFEST.mf*). If you are using a Java development tool that supports the Servlet 2.3 specification, your tool most likely provides graphical support for specifying dependencies. See your tool's documentation for details.

WAR manifest format

The `Extension-List` manifest entry lists the names of required extensions. This entry has the form:

```
Extension-List: ext1 ext2 ext3 ...
```

Where *ext1*, *ext2*, *ext3*, and so forth are the names of the required extensions. For each name, you must specify additional entries from the Manifest entry column of Table 21-2 on page 401, prefixed with the name and a hyphen. For example, if the name is javahelp, you must specify a `javahelp-Extension-Name` entry as well as the other manifest entries from Table 21-2. You may specify additional entries not in Table 21-2, but these are ignored by EAServer.

Example

The following example shows a section of a WAR manifest that requires two extensions, `javahelp` and `java3d`:

```
Extension-List: javahelp java3d
javahelp-Extension-Name: javax.help
javahelp-Specification-Version: 1.0
javahelp-Implementation-Version: 1.0.3
javahelp-Implementation-Vendor-Id: com.sun
java3d-Extension-Name: javax.3d
java3d-Specification-Version: 1.0
java3d-Implementation-Version: 1.2.1
java3d-Implementation-Vendor-Id: com.sun
```

Localizing Web applications

EAServer supports the HTTP 1.1 internationalization features defined in the Java Servlet 2.3 specification. Using these features, you can develop servlets that respond in the language specified by the request header, or configure localized versions of Web site's static pages.

For complete information about HTTP 1.1 internationalization, refer to the Java Servlet 2.3 specification and the HTTP 1.1 specification.

Enabling accept-language header parsing

HTTP 1.1 supports internationalization via an accept-language header that can be included in requests. The accept-language headers describe the languages the client accepts. For example, if documents are stored on the server in Japanese and English, clients that use Japanese as the accept-language header receive the Japanese version of the page. When clients use English as the accept-language header, they receive the English version. Accept-language headers can be sent only by Web browsers that use the HTTP 1.1 protocol.

The com.sybase.jaguar.server.http.acceptlang property determines whether EAServer parses accept-language headers to respond to requests for localized content. To enable accept-language header parsing, set this property to true using the Advanced tab in the Server Properties window in EAServer Manager.

Internationalization for servlets

For servlet development, EAServer supports internationalization compliant methods that are described in the Java Servlet 2.3 specification. These methods, getLocale and getLocales on the ServletRequest interface and setLocale on the ServletResponse interface:

- getLocale and getLocales - parse the accept-language header, extract the language and quality value information, and return the specified locale names. If the request specifies no locale, return the server's default locale.
- setLocale - sets the language attributes in the Content-Language header. The default is the server's default locale.

Deploying localized static files

A separate directory is required for each supported language along with a default directory. EAServer refers to these directories to locate different language versions of a document. For example, if the client requests the URL:

```
http://www.someplace.com/somepage.html
```

and EAServer supports English and French. There will be two versions of the page on the server plus the default:

- The English Version –
  `http://www.someplace.com/en/somepage.html`

- The French Version – `http://www.someplace.com/fr/somepage.html`
- A default Version – `http://www.someplace.com/somepage.html`

Language selection algorithm

A Language selection algorithm selects the appropriate language after evaluating the override criteria and the quality values specified. If multiple languages are specified, then the algorithm checks the various options in descending order of priority. For example, if the client requests this URL with en, fr specified in the accept-language header:

```
http://www.someplace.com/somepage.html
```

EAServer first looks for:

```
http://www.someplace.com/en/somepage.html
```

If not found, the server looks for:

```
http://www.someplace.com/fr/somepage.html
```

If this is not found, the server tries to load the default page:

```
http://www.someplace.com/somepage.html
```

Similarly, for static Web resources in a Web applications, the language name tag is prefixed to the static web resource URL to construct the URL for the resource. EAServer provides multiple language support to the following Web application resources:

- Servlets
- Web application with static Web resources
- Static Web pages

Localizing JSP content

JSPs that use a character set other than the server default require additional changes in source code and deployment properties.

In your JSP source code, specify the encoding in the page declaration, for example:

```
<%@ page contentType="text/html;charset=BIG5" %>
```

When initializing strings, pass the encoding name to the String constructor, for example:

```
byte[] b = { (byte)'\u00A4', (byte)'\u00A4',
             (byte)'\u00A4', (byte)'\u00E5' };
String s = new String(b, "big5");
```

If you do not specify the encoding name, the byte array may be converted incorrectly.

When deploying localized JSPs, group JSPs for each language in their own directory tree under your Web application's context root. For example, all files under */en* are English, 8859_1 encoded and all files under */ko* are Korean, KSC5601 encoded. Additionally, configure the following Web application properties:

Property name	Used to specify
com.sybase.jaguar.webapplication.charset.inputparam	Character set for request parameters.
com.sybase.jaguar.webapplication.charset.inputdata	Character set for request body data (retrieved with ServletRequest.getReader or ServletRequest.getInputStream).
com.sybase.jaguar.webapplication.charset.jspcompile	Character set for JSP compilation.

The property values must contain a list of URL-pattern and Java character set name pairs. Use this syntax, where *URL_pattern* is the url-pattern to which the character set applies, and *character_set* is the name of the Java character set:

```
(url-pattern=URL_pattern,charset=character_set),
(url-pattern=URL_pattern,charset=character_set)
```

For example, for a Web application with two directories, */en* and */ko*, in its document root where all files under */en* are 8859_1 encoded and all files under */ko* are KSC5601 encoded, specify the character sets like this:

```
(url-pattern=/en/*,charset=8859_1),
(url-pattern=/ko/*,charset=KSC5601)
```

If a URL pattern is not listed, the server's default character set is used. If you specify a character set that is not supported, it is not added to the mapping and the server's default character set is used.

Note These character set properties are not supported for the default Web application.

CHAPTER 22

Creating Java Servlets

EAServer supports version 2.3 of the Java Servlet API. Running in EAServer, servlets can create HTML pages with dynamic content and respond to requests from client applications that are implemented as HTML forms. Servlets also allow you to execute business logic from any Web browser or any other client application that connects using the Hypertext Transfer Protocol (HTTP).

Introduction to Java servlets

The Java Servlet API is a Java Standard Extension Java classes that extend the functionality of a Web server.

Use of servlets in EAServer

Java servlets respond to HTTP requests from Web browser clients (or any other client that connects to EAServer using the HTTP protocol). You can associate an HTTP URL with a servlet that you have installed in EAServer. The servlet can dynamically create HTML documents, or act as a gateway between HTML-forms based applications and EAServer components. For example, you might create servlets to:

- **Create dynamic HTML page content** Your servlet creates pages for an online catalog by selecting part descriptions from a database.

- **Act as a gateway between HTML forms and EAServer components**
  Your client application consists of an HTML page with embedded HTML forms that submits the data to the servlet. When invoked, the servlet calls EAServer components, supplying the form data as parameters. For simple user interfaces, HTML forms can offer better performance than Java applet clients, since the browser does not download applet code.

EAServer provides an extended version of the Servlet API so that servlets may use EAServer services such as inter-server component invocations and database connection caching.

Java servlets versus Java components

Java servlets enhance the functionality offered by Java components, but do not replace Java components. Servlets in EAServer can only be invoked by HTTP clients, and must return all output by writing to a ServletOutputStream instance. Typically, servlets are invoked from HTML pages loaded in a Web browser and return formatted HTML as their output.

Java components can be executed by any EAServer client model, and can return complex objects in their natural format. To invoke Java components from a Web browser, you must create a Java applet that connects to EAServer and instantiates proxy objects for the component.

Servlets can make use of some, but not all, server-side services; for example, servlets can use cached database connections and can issue in-memory calls to components installed on the same server. Servlets cannot, however, participate in EAServer transactions, except as a base client. Servlets cannot use other server-side APIs besides connection caching and the Java ORB.

Java components have access to all Java server-side APIs and can participate in EAServer transactions.

For more information

The JavaSoft Servlet Web pages at http://java.sun.com/products/servlet/ describe how to code servlet classes.

Writing servlets for EAServer

You can implement servlets for EAServer as you would for any other server that follows the Java servlet specification. Servlets for EAServer can be coded to the standard Java servlet API and use classes in the javax.servlet and javax.servlet.http packages. This section lists coding information specific to EAServer and describes EAServer's extensions to the standard servlet API.

Connection caching

Servlets can use these classes to retrieve cached connections:

- com.sybase.jaguar.jcm.JCMCache, which represents a configured connection cache and provides methods to manage connections in the cache.
- com.sybase.jaguar.jcm.JCM, which provides access to JDBC connection caches defined in EAServer Manager. JCM is a factory for JCMCache instances.

For more information, see "Using Java Connection Manager classes" on page 486.

Component invocations

Servlets in EAServer can instantiate component instances using the same technique used within EJB or Java/CORBA components. Use the EJB technique when portability to other J2EE servers is required.

Using the EJB technique

To invoke component methods, use the lookup method in class javax.naming.InitialContext to resolve the Bean's home interface, then create a reference to the remote interface. For example:

```
import javax.ejb.*;
import javax.naming.*;

QueryBean _queryBean;
String _queryBeanName =
   "java:comp/env/ejb/querybean" ;
Context ctx = getInitialContext();
try {
    Object h = ctx.lookup(_queryBeanName);
    QueryBeanHome qbHome = (QueryBeanHome)
    javax.rmi.PortableRemoteObject.narrow(h,
       QueryBeanHome.class);
    _queryBean = qbHome.create();
}
catch (NamingException ne)
{
    System.out.println("Error: Naming exception: "
       + ne.getExplanation() + ne.toString());
    throw new Exception(
      "Lookup failed for EJB " + _queryBeanName);
}
```

For more information on the EJB client interfaces, see Chapter 8, “Creating Enterprise JavaBeans Clients.” For servlets installed in a Web application, you can define an EJB reference in the Web application properties to alias the name used in your source code. The EJB reference allows the Web application to be deployed on another J2EE server without changing your servlet code. See “EJB references” on page 383 for more information.

Using the Java/CORBA technique

To invoke component methods, create an ORB instance to obtain a proxy for the components, then invoke methods on the proxy object reference. For components on the same server, call the string_to_object method with the IOR string specified as *Package*/*Component*. For example, the fragment below obtains a proxy object for a component called *Payroll* that is installed in the *Finance* package:

```
java.util.Properties props = new
java.util.Properties();
props.put("org.omg.CORBA.ORBClass",
                  "com.sybase.CORBA.ORB");
ORB orb = ORB.init((java.lang.String[])null, props);
Payroll payroll =
PayrollHelper.narrow(orb.string_to_object(
                        "Finance/Payroll"));
```

By default, servlets run without a user name and password. A servlet client, authenticated by EAServer, runs with the client’s user name and password. If an unauthenticated servlet client invokes a component method, the component is instantiated without a user name and password. If roles limit access to a component or method and the servlet has no user name, a method invocation attempt fails. To specify a user name, use this syntax:

```
orb.string_to_object("iiop://0:0:user_name:password/Package/Component"));
```

You can retrieve the system user name and password with these methods in class com.sybase.CORBA.ORB, which both return strings:

- getSystemUser() returns the system user name.
- getSystemPassword() returns the system password.

When called from components, string_to_object returns an instance running on the same server if the component is locally installed; otherwise, it attempts to resolve a remote instance using the naming server.

Threading

If possible, servlets should be coded to be thread-safe, such that the service method can be called concurrently from multiple threads. This threading model is the default for servlets running in EAServer. In most cases, it offers the best performance. If your servlet cannot support this threading model, you must do one of the following to ensure that the servlet executes safely in EAServer:

- Code the servlet to implement the SingleThreadModel marker interface. This interface has no methods; the server recognizes that instances of any class that implements the interface must be single-threaded.
- Configure the servlet's threading properties as described in "Threading settings" on page 418.

Logging

Servlets can log error messages or other text to the EAServer servlet log file, using the standard servlet log methods in the ServletContext class (or the equivalent methods in the GenericServlet class). EAServer records servlet log messages in the *httpservlet.log* file, located in EAServer's *bin* subdirectory. If you define additional servers, the name of the servlet log file is prepended with the server name. For example, if you create a server named Test_server, then servlet messages for that server are directed to the *Test_serverhttpservlet.log* file.

To enable trace logging in the EAServer servlet execution engine, add the com.sybase.jaguar.server.servlet.trace property in the Server Properties dialog box on the Advanced tab, and set it to true.

Error pages

You can customize error and exception reports that are sent to clients by creating error pages. When the servlet engine detects an error or catches an exception thrown by a servlet, it searches for a corresponding error page to handle the response. You can declare error pages for a Web application, or at the server level.

This example illustrates how to declare an error page for a Web application in the deployment descriptor:

```
<error-page>
  <error-code>404</error-code>
  <location>/etc/404.html</location>
```

```
</error-page>
```

The location is the path relative to the Web application's context root. For example, */etc/404.html* corresponds to this file in your EAServer installation directory, where *web-app* is the name of the Web application:

```
Repository/WebApplication/web-app/etc/404.html
```

For information about how to use EAServer Manager to set up an error page for a Web application, see "Error pages" on page 380.

To set up server-level error pages, set the value of the com.sybase.jaguar.server.servlet.error-page property on the Advanced tab of the Server Properties dialog box. Use the following syntax to define a comma-delimited list of complex properties. Each property includes a location, and either an error code or an exception field.

(location=*/file.jsp*,error-code=*code*),
(location=*/exception.htm*,exception=*java.lang.Exception*)

Where:

- The locations of *file.jsp* and *exception.htm* are relative to the HTML document root.
- *code* is the error code that triggers the error page.
- *java.lang.Exception* is the fully-qualified Java class name of the exception that triggers the error page.

Request dispatching

A RequestDispatcher instance allows one servlet to invoke another and either forward a request, or include the target servlet's response with its own. The RequestDispatcher interface provides methods to accomplish both. To obtain an object that implements the RequestDispatcher interface, use one of these ServletContext methods:

- getRequestDispatcher(<*URL map to resource*>)
- getNamedDispatcher(<*servlet name*>)

To forward a request, the initial servlet calls the forward method of the RequestDispatcher interface. The target servlet returns the response. This method can be called only if no output has been committed to the client. Before the forward method returns, the response must be committed and closed by the servlet container.

To include a target servlet's response with its own, the initial servlet calls the include method of the RequestDispatcher interface. The target servlet has full access to the request object but can write only to the ServletOutputStream or Writer of the response object and it cannot modify the response headers. The target servlet can commit a response by either writing past the end of the response buffer, or explicitly calling the flush method of the ServletResponse interface.

URL interpretation

The ServletContext and ServletRequest objects both contain methods to retrieve a RequestDispatcher instance. ServletContext methods require an absolute URL. ServletRequest methods can interpret a relative URL. Both URL types must follow these guidelines:

- The path cannot include the context.
- Mappings must agree with the servlet mappings defined for the Web application—if a mapping does not exist, use the static page in the Web applications's context root directory *$JAGUAR/Repository/WebApplication/<web-app-name>*.
- You must resolve dots in the path before mapping the URL.
- There can be no static content access at *WEB-INF/META-INF*.

A ServletContext.getRequestDispatcher URL must begin with a forward slash ('/'). If a ServletRequest.getRequestDispatcher URL begins with a forward slash, the servlet engine interprets it as an absolute URL. Otherwise, the servlet engine appends the relative URL to the current request's URI path. For example, if the current request is */catalog/garden.html* and the relative URL is *sports.html*, then the new URL is */catalog/sports.html*.

Implementation

EAServer's servlet engine passes all servlet invocation requests through a RequestDispatcherobject instance. When the servlet engine receives a request from a client, it calls the RequestDispatcher.service method. This method loads, initializes, and handles instance pooling of single-threaded servlets. It also invokes the servlet and handles errors.

Static content

A RequestDispatcher instance would typically be used for servlets and JSPs, but it can also be used for static content. If the servlet engine forwards a request to a static content RequestDispatcher, the RequestDispatcher must set the response status, the response headers, and the response data. If a static content RequestDispatcher is called to set the data for the current request, it only needs to return the content of the static page.

Response buffering

The Java servlet API supports response buffering that allows the servlet to control how the servlet container buffers responses, and when to send a response to a client. The ServletResponse interface provides these methods that allow a servlet to access buffering information:

- getBufferSize – returns the size of the response buffer; if buffering is not used, returns integer value of zero.
- setBufferSize – sets buffer size greater than or equal to the servlet's request.
- isCommitted – returns a boolean value to indicate whether any part of the response has been returned to the client.
- reset – clears the buffer of an uncommitted response.
- flushBuffer – writes buffer contents to a client.

See the *Java Servlet Specification, v2.3* for detailed information about using response buffering.

Encoding responses and double-byte characters

When you compile a Java servlet, the characters are encoded according to the locale of your machine unless you specify encoding in the javac compile command. When a client sends a request from a browser, the parameters are always ISO 8859-1 encoded.

To provide a client's browser with the encoding information it needs to translate the content of a response correctly, declare the encoding in the response header. If you specify the content type without the encoding information, for instance:

```
response.setContentType("text.html");
```

the client's browser assumes that the content is ISO 8859-1 encoded. If the content has been encoded using some other standard, the client's browser does not translate the data correctly. This example specifies the double-byte character set big5, the encoding name of traditional Chinese characters:

```
response.setContentType("text/html;charset=big5");
```

To encode the response content, compile the servlet with this encoding option:

```
javac -encode iso-8859-1 <java source file>
```

or convert static strings within the servlet code, for instance:

```
String origMsg = "<double-byte character string>";
String newMsg = new String(origMsg.getBytes(),
                           "iso-8859-1");
```

Installing and configuring servlets

After you have created or obtained the Java class that implements your servlet's functionality, you must define a new servlet in EAServer Manager, associate it with your class, then configure the properties that control how the servlet's class is loaded and executed.

Installing servlets

In EAServer Manager, servlets that are installed in EAServer display in the Installed Servlets folder under the server's icon. All servlets that have been defined are displayed in the top-level Servlets folder. You must install a servlet in a server before that server's clients can execute the servlet.

Defining a new servlet

When defining a new servlet, you can install it in a server at the same time, or you can define the servlet in the top-level Servlets folder, then install it in one or more servers later.

1 To create a servlet and install it in a server:

- Expand the server's icon, then highlight the Installed Servlets folder within it.

- Choose File | Install Servlet.
- In the Servlet Wizard, click Create and Install a New Servlet.

To define a servlet that is not installed in a server:

- Highlight the icon for the top level Servlets folder.
- Choose File | New Servlet.

2 Enter a name for the servlet. This name will be used in HTTP URLs that invoke the servlet.

3 Configure the servlet properties as described in “Configuring servlet properties” on page 417.

Installing existing servlets into a server

You must install servlets in a server before that server’s clients can invoke the servlet. You can install a servlet into multiple servers. To install a servlet into a server:

1 Expand the server’s icon, then highlight the Installed Servlets folder within it.

2 Choose File | Install Servlet.

3 In the Servlet Wizard, click Install an Existing Servlet.

4 In the Install Servlet dialog box, highlight the servlet to be installed, then click Ok.

Uninstalling servlets from a server

Uninstalling a servlet from a server makes that servlet unavailable to clients of that server. The server definition persists in EAServer Manager, under the top level Servlets folder. To uninstall a servlet:

1 Expand the server’s icon, then highlight the Installed Servlets folder within it.

2 Highlight the servlet to uninstall.

3 Choose File | Remove Servlet.

Deleting servlet definitions

Deleting a servlet from the top-level Servlets folder removes it entirely from EAServer Manager. To delete a servlet definition:

1 Expand the top-level Servlets folder.

2 Highlight the servlet to delete.

3 Choose File | Delete Servlet.

Configuring servlet properties

The settings in the Servlet Properties dialog box specify the Java class for the servlet and control how EAServer loads and runs instances of the class. The dialog contains the tabs described below.

General settings

Properties on the general tab define the basic information required to load and run the servlet.

Table 22-1: General Tab Settings

Control Name	Specifies	Applies to
Description	An optional comment describing the servlet.	All servlets
Servlet's fully qualified class name	The name of the Java class that implements the servlet functionality, in Java dot notation; for example, com.sybase.jaguar.DemoServlet. In a Web application, you can map a servlet name to either a servlet class or a JSP file.	All servlets
Load during startup	Choose Yes if the servlet must be loaded and initialized when the server starts. If you choose No, the class is loaded when the first client requests to run the servlet. Classes that perform lengthy processing in the init method can be loaded at start-up so that the first client to invoke the servlet does not experience increased response time. If you choose Yes, servlets are reloaded when you refresh the Web application.	All servlets
Startup load sequence position	EAServer loads servlets serially. If you choose Yes, to load the servlet during startup, define the order, relative to other servlets in the application. To load the servlet first, enter 1.	All servlets

Control Name	Specifies	Applies to
Web component type	Choose Servlet or JSP	Web application servlets only

Init-Args settings

Servlets may require initialization parameters that are specified outside of the source code. For example, you might specify the name of an EAServer connection cache as an initialization parameter. You can use the Init-args properties to define optional initialization parameters for the server.

The Init-Params tab lists the initialization parameters that have been defined for the servlet. Click Add to define a new initialization parameter. Enter the parameter name and the text of the value. The servlet can retrieve the value as a Java String, as explained below. To change a parameter's value, highlight the parameter in the list, then click Modify. To remove a parameter, highlight it, then click Delete.

Your servlet's init method can retrieve the specified settings using the ServletConfig.getInitParameter(String) and ServletConfig.getInitParameterNames() methods. The following code fragment shows how:

```
void init(ServletConfig config) throws ServletException
{
  ....
  Enumeration paramNames =
    config.getInitParameterNames();
  while (paramNames.hasMoreElements())
  {
    String name = (String) paramNames.nextElement();
    String value = config.getInitParameter(name);
  }
```

Threading settings

By default, EAServer loads one instance of a servlet class and calls methods from multiple threads—to service multiple clients, multiple threads may call the service method simultaneously. If an instance of your servlet cannot safely execute in multiple threads, you must configure the Threading tab to specify that the servlet class is single threaded. You can also specify how EAServer should serialize invocations of the service method for a single-threaded servlet.

Check the Single Threaded option if calls to your servlet's service method must be serialized. When this option is selected, you can specify the number of instances that EAServer creates to serve client requests. Calls to the service method within a given instance are serialized. EAServer creates multiple instances to minimize the time that clients have to wait for a blocked service request. EAServer calls the service method in an instance that is not already busy serving a previous request. If all available instances are busy, the request is delayed until a service call returns.

If multiple instances are created, calls to the service method are not necessarily serialized; service calls may occur simultaneously in different instances. If your service method changes static variables, you must add code to synchronize these changes or configure the servlet properties so that only one instance is created.

Implementing the SingleThreadModel interface Your servlet class can implement the SingleThreadModel interface to indicate that calls to an instance's service method must be serialized. Instances of these classes are always single-threaded by EAServer, regardless of whether the Single Threaded option is enabled.

Sessions settings

The Java Servlet API provides classes to create a session between a given HTTP client and servlets running on EAServer. You can use the session to record data related to the end-users session. If your servlet uses sessions, configure the following properties on the Sessions tab:

- **Time-out** The duration, in seconds, that a session can remain inactive.

  In Web applications, servlets share a common timeout value. To set the session timeout property for a Web application servlet:

  a Expand these icons: Servers, *server-name*, Installed Applications, *application-name*, and Web Applications. Highlight the Web application, right-click and select Web Application Properties.

  b On the General tab, set Session Timeout to the number of minutes that a session can remain inactive. The default is 0, which indicates that the session never times out.

- **Enable Session Tracking** If selected, sessions are enabled in the servlet. If not checked, sessions are disabled. Sessions are enabled by default.

  For Web application servlets, session tracking is always enabled.

Java Classes settings

On the Java Classes tab, specify a list of additional Java classes that must be reloaded when you refresh the servlet. By default, EAServer reloads only the servlet implementation class. You can configure the classes to be custom loaded at the servlet, Web application, J2EE application, or server level. For more information, see Chapter 30, "Configuring Custom Java Class Lists."

Additional Files settings

The Additional Files tab enables you to associate additional files with the servlet definition. If you synchronize the server where the servlet is installed, and elect to synchronize servlet files, these files will be transferred to the target servers. See Chapter 6, "Clusters and Synchronization," in the *EAServer System Administration Guide* for more information on this feature.

You can specify the name of a Java class or package to be added to the archive, using the Java dot notation. For example, "com.sybase.CORBA" adds all files in the com.sybase.CORBA package.

Any other files must be separated by commas and specified relative to EAServer's *Repository* subdirectory or with a full path. Full paths require that any server to which you synchronize share the same directory structure.

When you include additional files, you can either enter the file names individually, or you can use the Additional Files wizard to add multiple files, packages, classes, and directories.

To enter file names individually:

1 Click Add. This opens the Add a File Name to the List dialog box.

2 Enter the file name and click Ok.

To add multiple items:

1 Click Additional Files Wizard. This open the Additional Files dialog box. Each item that you add is appended to the list.

2 To add Java packages or classes:

   a Click Browse

   b Choose a **.class* file and click Select.

   The class files must be deployed under EAServer's *java/classes* directory.

3 To add files or directories:

   a Optionally, specify a file filter, such as **.txt*.

   b Optionally, select to use the JAGUAR environment variable.

   c Click Browse.

   d Choose a file or directory and click Select.

4 To add property files from other entities:

   a Click Browse.

b Choose a **.props* file from under the *Repository* directory and click Select.

5 To add file lists from other entities:

a Click Browse.

b Choose an entity's **.files* file and click Select.

6 Click Add Files to Additional Files List.

Run As Identity

Configures an alternate identity used for authentication of component invocations from the servlet or JSP. By default, component invocations use the Web client's identity. The settings are:

- Run as – Choose "specified" to configure an alternate identity. The default, "client," specifies that the Web client identity is used.
- Role – Specify a role name. The identity specified in the Mapped to Jaguar identity field should be in this role.
- Run as identity – Specify a logical identity name. This name is used if the component is exported to an EJB-JAR file.
- Mapped to Jaguar identity – Choose an EAServer identity from the pull down menu. This is the identity with which the component executes.
- Description – Enter an optional text comment. This field can be used to provide identity mapping instructions for the deployer when the component is deployed to another server.

To enable use of the run-as identity for EJB component calls to remote servers, you must specify corbaname URLs in the EJB Reference properties for the Web application where the servlet is installed. For more information, see "Interoperable naming URLs" on page 157 and "EJB references" on page 383.

Advanced settings

The Advanced tab allows you to edit property settings as they are stored in the EAServer configuration repository. You can only delete properties that you have added—you cannot delete default properties, such as the com.sybase.jaguar.servlet.name property. Repository property names are documented in Appendix B, "Repository Properties Reference," in the *EAServer System Administration Guide*.

You can set properties as follows:

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.

When to use the Advanced tab Though you can use the Advanced tab to set any property prefixed with "com.sybase.jaguar.servlet," Sybase recommends that you use this tab to set properties only as specified by the EAServer documentation or by Sybase Technical Support. Most properties can be configured graphically elsewhere in the EAServer Manager user interface.

Deploying and refreshing servlet classes

Deploying servlet classes

Though you can deploy servlet classes under any codebase that is specified in the CODEBASE environment variable, servlet classes should be deployed under EAServer's *java/classes* subdirectory to simplify debugging. Only classes deployed under this codebase can be refreshed.

Refreshing servlets

The refresh feature is useful for debugging, since it allows you to load a changed version of the implementation class without restarting the server. If your implementation relies on other classes that must also be reloaded when the implementation is refreshed, specify them on the Java Classes tab in the Servlet Properties window. See "Java Classes settings" on page 419.

All servlets that are not installed in a Web application are considered to be part of the *default* Web application, and all servlets within the same Web application are refreshed at the same time.

To refresh all servlets in the Installed Servlets folder:

1 Highlight the Servlets folder under the server icon where the servlet is installed.

2 Choose File | Refresh.

3 EAServer Manager will ask if you want to terminate any active requests. Choose Yes to refresh. If one of the servlets is servicing a request at this time, the client may receive partial data or an error. Choose No to cancel the refresh operation.

To refresh all the servlets in a Web application:

1 Expand the server icon where the servlet is installed and highlight the Web application name under one of these folders:
   - Installed Web Applications
   - Installed Applications | <*application name*> | Web Applications
2 Choose File | Refresh.
3 EAServer Manager will ask if you want to terminate any active requests. Choose Yes to refresh. If one of the servlets is servicing a request at this time, the client may receive partial data or an error. Choose No to cancel the refresh operation.

When you refresh a servlet, EAServer calls the servlet's Servlet.destroy() method, reloads the implementation class and any classes specified on the Java Classes tab, and then calls the Servlet.init() method in the new instance.

Starting and stopping servlets

At times you may wish to stop and restart the servlet without reloading the class. Also, starting a servlet causes EAServer to load the implementation class if it has not already been loaded at startup or in response to a client request.

When you stop the servlet, EAServer calls the Servlet.destroy() method. When you start the servlet, EAServer calls the Servlet.init() method, unless it has already been called on the current instance of the implementation class.

To start a servlet:

1 Expand the Servlets folder under the server icon where the servlet is installed. Highlight the servlet's icon.
2 Choose File | Start.

To stop a servlet:

1 Expand the Servlets folder under the server icon where the servlet is installed. Highlight the servlet's icon.
2 Choose File | Stop.

3 EAServer Manager will ask if you want to terminate any active requests. Choose Yes to stop the servlet. If the servlet is servicing a request when you stop the servlet, the client may receive partial data or an error.

Web application support

Java servlets support packaging and deploying Web applications. A Web application archive (WAR) file contains all the components of a Web application including servlets, HTML files, JavaServer Pages (JSPs), classes, and other resources. See Chapter 21, “Creating Web Applications” for more information.

EAServer includes a servlet container that provides network services for requests and responses, decodes MIME-based requests, formats MIME-based responses, and manages servlets.

Adding servlets to a Web application

To add servlets to a Web application, copy the servlet class files under *%JAGUAR%\Repository\WebApplication\<web-app>\WEB-INF\classes*, and use EAServer Manager to add the servlet to your Web application.

❖ **Adding a servlet to a Web application**

1 In EAServer Manager, select either Web Applications | <*Web application*> or Applications | <*application*> | Web Applications | <*Web application*>.

2 Right-click and select New Web Component, and enter the name of the servlet.

3 Select the servlet, right-click, and select Web Application Component Properties.

4 Enter values for the servlet properties described in “Configuring servlet properties” on page 417.

❖ **Mapping a Web application’s servlet to a URL**

1 In EAServer Manager, select either Web Applications | <*Web application*> or Applications | <*Application*> | Web Applications | <*Web application*>.

2 Right-click and select Web Application Properties.

3 Select the Servlet Mapping tab and click Add. A new row is added to the mapping table.

4 Place the cursor in the Servlet cell and enter the servlet name that displays in EAServer Manager.

5 Place the cursor in the URL Pattern cell and enter a string to invoke the servlet from an HTTP URL. For example, if the Web application name is *WebApp1* and the URL Pattern string for the servlet is */MyServlet*, this URL invokes the servlet:

```
http://host:port/WebApp1/MyServlet
```

6 Place the cursor in the Description field and enter a description of the servlet.

7 Click OK.

Note Servlets installed in a Web application have no default URL mappings. To invoke a servlet, clients must use the path mapped to the servlet in the Web application properties.

In the normal configuration, you cannot run servlets without using an alias or Web application name in the request URL. You can configure servlets to run with no alias as follows:

1 Install the servlets of interest in your server's Installed Servlets folder, as described in "Installing existing servlets into a server" on page 416.

2 Display the Server Properties dialog for your server, then display the Advanced tab.

3 Search for com.sybase.jaguar.server.servlet.servlet-mapping in the list. If the property is present, highlight it and click Modify. Otherwise, click Add and enter the property name.

4 For the property value, enter a comma-separated list of entries with this format:

```
(url-pattern=/pattern,servlet-name=servlet)
```

Where:

- *pattern* is the alias to invoke the servlet, for example, `MyServlet`.
- *servlet* is the servlet name, as defined in EAServer Manager.

For example, to map MyServlet to the path */myservlet*, and HelloServlet to the path */hello*, enter this value (on one line):

```
(url-pattern=/myservlet,servlet-name=MyServlet),(url-
pattern=/hello,servlet-name=HelloServlet)
```

With these settings, HelloServlet can be invoked with this URL:

```
http://host:port/hello
```

Server properties for servlets

On the Servlet tab in the Server Properties window, you can disable servlet execution in a server and configure additional properties to control the execution of servlets.

❖ **Displaying the servlet execution properties**

1 Highlight the icon for the server of interest.

2 Choose File | Server Properties.

3 Scroll the tab display at the top of the window until the Servlet tab displays, then click on it.

Servlet tab controls

The Servlet tab specifies how EAServer executes servlets, as follows:

- **Servlet Execution Enable/Disable** This option determines whether servlets can execute on a server. If the option is disabled, no installed servlets can be invoked. By default, servlet execution is enabled.
- **Enable Class-Name Request** If this option is enabled, users can invoke a servlet by typing the name of its Java class rather than a EAServer Manager servlet name. For example:

  ```
  http://yourhost:8080/servlet/com.yours.AServlet
  ```

  By default, invocation by class name is allowed. You may wish to disable such access for the following reasons:

  - Disabling the option restricts available servlets to those that have been installed in the server's Servlets folder in EAServer Manager.
  - When Servlets are invoked by class name, they are run according to the default servlet property settings. These settings may not be appropriate for some servlets.

- **Servlet Aliases** Specifies the list of path prefixes that users can use to invoke servlets from HTTP URLs. For example, if */servlet/* is a path prefix, this URL invokes a servlet named *MyServlet*:

  ```
  http://yourhost:8080/servlet/MyServlet
  ```

  The default setting specifies */servlet/* as the only path prefix. To override the default, enter one or more prefixes, each on a line by itself. For example:

  ```
  /servlet/
  /servlets/
  ```

- **Timeout** This option specifies how long the server should wait for each servlet's init method to return. For any value, no client requests are serviced while the init method is running. Service requests that arrive while init is running are blocked until init returns. Clients receive browser timeout errors when attempting to execute the servlet while init is running. You can set the Timeout value to control how the server treats servlets if the init method is still running when you shut down the server or refresh the servlet. Table 22-2 describes the possible values.

Table 22-2: Initialization timeout values

Value	To indicate
-1	(The server or Web application default.) init can run indefinitely, unless the server is shutdown or refreshed. If the init method is still running when the server is shutdown or refreshed, the server does not wait for init to complete before shutting down or refreshing the servlet.
0	init can run indefinitely. Sybase does not recommend this setting, because deadlocks or other hangs in the init method can cause the server to hang when shutting down or refreshing the servlet.
A positive integer.	The number of seconds to wait for init to return. If the init method is still running when the server is shutdown or refreshed, the server waits the specified time for init to return.

For servlets installed in a Web application, set the Web application Timeout property described in "General properties" on page 376.

You can override this setting for individual servlets. Display the Advanced tab in the Servlet Properties window, then set the com.sybase.jaguar.servlet.init.timeout property using the syntax in Table 22-2.

- **Destroy Time-out** EAServer calls each servlet's destroy method before shutting down or after you have refreshed or stopped the servlet using EAServer Manager. If service calls are still active, this setting specifies the number of seconds that the server should wait for the service calls to return before calling the destroy method. The default is 0, which specifies that EAServer calls destroy immediately.

  You can override the server-wide default wait-time for individual servlets. Display the Advanced tab in the Servlet Properties window, then set the com.sybase.jaguar.servlet.destroy.wait-time property to the desired number of seconds.

CHAPTER 23

Using Filters and Event Listeners

This chapter discusses how to use servlet filters and listeners that can respond to application lifecycle events.

For complete information on servlets, see Chapter 22, “Creating Java Servlets.”

Servlet filters

You can use filters to modify the header or the content of a servlet request or response. Within a Web application, you can define many filters, and a single filter can act on one or more servlets or JavaServer Pages (JSPs). Filters can help you accomplish a number of tasks, including data authentication, logging, and encryption.

You can map filters to a URL or a servlet name. When a filter is mapped to a URL (path-mapped), the filter applies to every servlet and JSP in the Web application. When a filter is mapped to a servlet name (servlet-mapped), it applies to a single servlet or JSP. EAServer constructs a list of the filters declared in a Web application’s deployment descriptor; this list is called a *filter chain*. The order of the filters in the filter chain determines the order in which the filters are executed. EAServer constructs the filter chain by first adding the path-mapped filters, in the order that they are declared in the deployment descriptor, then it adds the servlet-mapped filters in the order in which they appear in the deployment descriptor. As a result, the path-mapped filters are executed first, followed by the servlet-mapped filters.

This sample declares the path-mapped filter, MyFilter:

```
<filter>
   <filter-name>
      MyFilter
```

```
    </filter-name>

    <filter-class>
       MyFilter
    </filter-class>

</filter>

<filter-mapping>
    <filter-name>MyFilter</filter-name>
    <url-pattern>/*</url-pattern>
<filter-mapping>
```

Use EAServer Manager to add a new filter to a Web application and map it to either a servlet name or a URL pattern.

❖ **Adding a new filter to a Web application**

1 Expand the Web Applications folder, then highlight the icon that represents your application.

2 Choose File | New Web Component.

3 Select Filter and enter a name for the filter.

4 Click OK. This displays the Filter Component Properties dialog box.

5 On the General tab, enter:

- A description of the filter.
- The filter's fully-qualified class name.

6 On the Init-Params tab, enter the initialization parameters as name/value pairs. When the filter is initialized, it receives a FilterConfig object that contains these parameters.

   a Click Add to display the New Property dialog box.

   b Enter a property name and property value, and optionally, a description, then click OK.

   To edit an initialization parameter, highlight the property and click Modify. Edit the property name or value, and click OK.

   To delete an initialization parameter, highlight the property and click Delete.

7 Click OK.

"Filter Mapping properties" on page 393 describes how to map a Web application filter.

You can also define filters at the server level—see "HTTP Custom Response Header" on page 41 in Chapter 3, "Creating and Configuring Servers," in the *EAServer System Administration Guide*.

Servlet filters must implement the javax.servlet.Filter interface and define these methods:

Interface method	Description
init	Calls a filter into service and sets the filter's configuration object.
doFilter	Performs the filtering work.
getFilterConfig	Returns the filter's configuration object.
destroy	Removes a filter from service.

Note The setFilterConfig method is no longer supported (as of version 4.1); it has been replaced by init and destroy.

To initialize each filter, EAServer calls the init method and passes in a FilterConfig object, which provides the filter with access to the Web application's ServletContext, the initialization parameters, and the filter name. After all the filters in a chain have been initialized, EAServer calls FilterChain::doFilter for the first filter in the chain and passes it a reference to the filter chain. Subsequently, each filter passes control to the next filter in the chain by calling the doFilter method. The requested resource, servlet or JSP, is served after all the filters in the chain have been served. To halt further filter and servlet processing from within a filter, do not call doFilter. To notify a filter that it is being removed from service, EAServer calls the destroy method. Within this mthod, the filter should clean up any resources that it holds: memory, file handles, threads, and so on. destroy is called only once after all the threads within the filter's doFilter method have exited.

Here is a sample implementation of a servlet filter, which records either the amount of time it takes to process the request, or the time the request finishes processing. The time is recorded using the ServletContext::log method. The filter uses the value of the initialization parameter *type* to determine whether to record the absolute time the filter finished, or the amount of time it took to process the request. If the value of *type* is "absolute", the filter logs the time the request completes; otherwise, it logs the processing time, in milliseconds.

```
package filters;
```

```
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.util.Date;

public class TimerFilter implements Filter
{
   private FilterConfig _filterConfig = null;

   /**
   * The server calls this method to initialize the Filter and
   * passes in a FilterConfig object.
   */
   public void init (FilterConfig filterConfig)
               throws javax.servlet.ServletException
   {
      _filterConfig = filterConfig;
   }

   /**
   * Return the FilterConfig object
   */
   public FilterConfig getFilterConfig()
   {
      return _filterConfig;
   }

   /**
   * EAServer calls this method each time a servlet, JSP or static Web
   * resource is invoked.
   */
   public void doFilter (ServletRequest request,
                         ServletResponse response,
                         FilterChain chain)
               throws java.io.IOException, javax.servlet.ServletException
   {
      // This is executed before the servlet/jsp/static resource is served.
      long startTime = System.currentTimeMillis();

      // Pass control to the next filter in the chain.
      chain.doFilter(request, response);

      // This is executed after the servlet/jsp/static resource has been served.
      long endTime = System.currentTimeMillis();

      // Get the ServletContext from the FilterConfig
      ServletContext context = _filterConfig.getServletContext();
```

```
        // Get the type parameter from the filter's initialization
        // paramters.  Return null if the parameter was not set
        String type = (String)_filterConfig.getInitParameter("type");

        // Get the filter's name to include in the log
        String filterName = _filterConfig.getFilterName();

        HttpServletRequest httprequest = (HttpServletRequest)request;
        String path = httprequest.getRequestURI();

        // By default, record the absolute time
        if ((type == null) || (type.equals("absolute")))
        {
           Date date = new Date(endTime);
           context.log(filterName + " - " + path + " finished: " +
                       date.toString());
        }
        else
        {
           context.log(filterName + " - time to process " + path + ": " +
                       (endTime - startTime) + "ms");
        }
     }
     /**
     * Notifies the filter that it is being taken out of service.
     */
     public void destroy()
     {
        // free resources
     }

}
```

Note To use page caching for servlets whose responses are modified by a filter, see "Using page caching with filters that modify a response" in Chapter 5, "Web Application Tuning," in the *EAServer Performance and Tuning Guide*.

Custom headers

To add custom response headers for static resources, EAServer provides the filter class com.sybase.jaguar.servlet.AddHeadersFilter. The filter is designed to add cache-related and simple name/string header information to a response. The initialization parameters that you pass to the filter must be in this format:

```
(name=header_name, value=type:value_type&val:value)
```

where *header_name* is the title of the header, and *value_type* and *value* can be:

Value type	Description	Sample value
String	A text string.	"Add this to the header."
Date	GMT date specified as, *[+|-] ddDhhHmmMssS.*	To indicate the current date, set *value* to `+0`. To indicate the current date plus 20 days, 10 hours, 30 minutes, and 20 seconds, set *value* to `+20D10H30M20S`.
Etag	EAServer generates an entity tag (etag) based on the file's length, last modified time, and the sum of the file.	*value* can be either "E" or "WE". "WE" specifies a weak etag. If you set *value* to "E", EAServer writes something like this in the header: `b222308-205-e28daea590`.

This example creates an instance of AddHeadersFilter and passes the header name, value type, and value:

```
com.sybase.jaguar.servlet.filter.init-param=
(name=Expires, value=type:date&val:+365D),
(name=Cache-Control, type:string&val:max-age=3600)
```

which adds these two lines to the response:

```
Expires: Wed, 15 May 2002 15:31:22 GMT
Cache-Control: max-age=3600
```

For more information

For more information on filters and programming customized responses, see the Java Web page at http://java.sun.com/products/servlet/Filters.html.

Application lifecycle event listeners

EAServer's implementation of application lifecycle events enables you to register event listeners that can respond to state changes in a Web application's ServletContext and HttpSession objects. When a Web application starts up, EAServer instantiates the listeners that are declared in the deployment descriptor. The servlet API provides four listener interfaces, which EAServer calls when each event occurs.

Event type	Listener interface	Description
Servlet context: lifecycle event	javax.servlet.ServletContextListener	The servlet context was just created and is available to service its first request, or the servlet context is about to be shut down.
Servlet context: attribute changes	javax.servlet.ServletContextAttributeListener	Servlet context attributes have been added, removed, or replaced.
HTTP session: lifecycle event	javax.servlet.http.HttpSessionListener	An HttpSession has just been created, invalidated, or timed out.
HTTP session: attribute changes	javax.servlet.http.HttpSessionAttributeListener	HttpSession attributes have been added, removed, or replaced.

Note The interfaces javax.servlet.ServletContextAttributeListener and javax.servlet.http.HttpSessionAttributeListener are both new for EAServer version 4.1. The corresponding interfaces from EAServer version 4.0, where "Attributes" was plural, are not supported in EAServer 4.1.

If you need your code to remain compatible with EAServer 4.0 or other servers that require the older interface names, implement both the old and new interfaces.

"Listener properties" on page 392 describes how to add a listener to a Web application.

Sample listener

Here is an example of how a ServletContextListener could be used to maintain a database connection for each servlet context. The database connection that gets created is stored in the ServletContext object as an attribute, so it is available to all the servlets in the Web application.

```
package listeners;

import javax.servlet.*;
import java.sql.*;

public final class ContextListener implements ServletContextListener
```

```
{
   ServletContext _context = null;
   Connection _connection = null;

   /**
   * This method gets invoked when the ServletContext has
   * been destroyed. It cleans up the database connection.
   */
   public void contextDestroyed(ServletContextEvent event)
   {
      // Destroy the database connection for this context.
      _context.setAttribute("DBConnection", null);
      _context = null;

      try {
         _connection.close();
      } catch (SQLException e) {
      // ignore the exception
      }
   }

   /**
   * This method is invoked after the ServletContext has
   * been created.  It creates a database connection.
   */
   public void contextInitialized(ServletContextEvent event)
   {
      _context = event.getServletContext();
      String jdbcDriver="com.sybase.jdbc2.jdbc.SybDriver";
      String dbURL="jdbc:sybase:Tds:localhost:2638";
      String user="dba";
      String password="";

      try {
         // Create a connection and store it in the ServletContext
         // as an attribute of type Connection.

         Class.forName(jdbcDriver).newInstance();
         Connection conn =
            DriverManager.getConnection(dbURL,user,password);
         _connection = conn;
         _context.setAttribute("DBConnection", conn);

      } catch (Exception e) {
         // Unable to create the connection, set it to null.
         _connection = null;
```

```
            _context.setAttribute("DBConnection", null);
        }
    }
}
```

CHAPTER 24

Creating JavaServer Pages

This chapter provides an overview of JavaServer Pages (JSP) and their place in distributed application development, as well as configuration instructions for running your JSPs in EAServer.

For detailed information about JavaServer Pages technology, see the JavaServer Pages specification, available at http://java.sun.com/products/jsp/download.html.

About JavaServer Pages

JavaServer Pages (JSP) technology provides a quick, easy way to create Web pages with both static and dynamic content. JSPs are text-based documents that contain static markup, usually in HTML or XML, as well as Java content in the form of scripts and/or calls to Java components. JSPs extend the Java Servlet API and have access to all Java APIs and components.

You can use JSPs in many ways in Web-based applications. As part of the J2EE application model, JSPs typically run on a Web server in the middle tier, responding to HTTP requests from clients, and invoking the business methods of Enterprise JavaBeans (EJB) components on a transaction server.

How JavaServer Pages work

JSPs are executed in a JSP engine (also called a JSP container) that is installed on a Web or application server. The JSP engine receives a request from a client and delivers it to the JSP. The JSP can create or use other objects to create a response. For example, it can forward the request to a servlet or an EJB component, which processes the request and returns a response to the JSP. The response is formatted according to the template in the JSP and returned to the client.

Translating into a servlet class

You can deploy JSPs to the server in either source or compiled form. If a JSP is in source form, the JSP engine typically translates the page into a class that implements the servlet interface and stores it in the server's memory. Depending on the implementation of the JSP engine, translation can occur at any time between initial deployment and the receipt of the first request. As long as the JSP remains unchanged, subsequent requests reuse the servlet class, reducing the time required for those requests.

Deploying the JSP as a compiled servlet class eliminates the time required to compile the JSP when the first request is received. It also eliminates the need to have the Java compiler on the server.

Requests and responses

Some JSP engines can handle requests and responses that use several different protocols, but all JSP engines can handle HTTP requests and responses. The JspPage and HttpJspPage classes in the *javax.servlet.jsp* package define the interface for the compiled JSP, which has three methods:

- jspInit()
- jspDestroy()
- _jspService(HttpServletRequest request,
  HttpServletResponse response)

For more information about the EAServer implementation of the JSP engine, see "Using JSPs in EAServer" on page 451.

What a JSP contains

A JSP contains static template text that is written to the output stream. It also contains dynamic content that can take several forms:

- *Directives* provide global information for the page, or include a file of text or code.

- *Scripting elements* (declarations, scriptlets, and expressions) manipulate objects and perform computations.
- *Standard tags* perform common actions such as instantiating or getting or setting the properties of a JavaBeans component, downloading a plug-in, or forwarding a request.
- *Custom tags* perform additional *actions* defined in a custom tag library.

For more detailed information about using these content types, see "Application logic in JSPs" on page 447.

A simple example

This sample JSP contains a directive, a scripting element (in this case an expression), and a standard tag. The dynamic content is shown in bold:

```
<HTML>
<HEAD><TITLE>Simple JSP</TITLE>
</HEAD>
<BODY>
<P>This page uses three kinds of dynamic content: </P>
<UL><LI>A page directive that imports the java util
package.
<%@ page import = "java.util.*" %>
<LI>An expression to get the current date using
java.util.Date. Today's date is <%= new Date() %>.
<LI>An include tag to include data from another file
without parsing the content.
<jsp:include page="includedpage.txt" flush="true"/>
</UL>
</BODY>
</HTML>
```

The page referenced is a text file that contains one sentence and is in the same directory as the JSP file. The included page might also be another resource, such as a JSP file, and its location can be specified using a URI path.

You can call the JSP from an HTML page with a hypertext reference:

```
<html><body>
<p><a href="simplepage.jsp">Click here to send a
request to the simple JSP.</p>
</body></html>
```

This HTML is returned to the browser:

```
<HTML>
<HEAD><TITLE>Simple JSP</TITLE>
</HEAD>
<BODY>
<P>This page uses three kinds of dynamic content: </P>
```

```
<UL><LI>A page directive that imports the java util
package.
<LI>An expression to get the current date using
java.util.Date. Today's date is Mon Feb 14 17:03:51 EST
2000.
<LI>An include tag to include data from another file
without parsing the content.
In this case the included file is a static file
containing this sentence.
</UL>
</BODY>
</HTML>
```

Why use JSPs?

JavaServer Pages inherit the concepts of Applications, ServletContexts, Sessions, Requests, and Responses from the Java Servlets API and offer the same portability, performance, and scalability as servlets.

About Java servlets

Java servlets overcome many of the deficiencies of CGI, ISAPI, and NSAPI. Although the CGI-BIN interface itself is not platform-specific, code has to be recompiled for different platforms, and performance is poor for large-scale applications because each new CGI request requires a new server process. Similar platform-specific interfaces such as ISAPI and NSAPI improve performance, but at the cost of even less portability.

Because Java servlets are written in Java, they are completely platform- and server-independent. They provide superior performance and scalability because they can be compiled, loaded into memory, and reused by multiple clients while running in a single thread, and they can take advantage of connection caching or pooling.

Java servlets are described in more detail in Chapter 22, “Creating Java Servlets”.

Java servlets and JSPs

Java servlets and JSPs are based on the same API, and either can be used to fill some roles in a Web application. But while Java servlets are Java code with embedded HTML, JSPs are HTML (or XML) pages with embedded Java code. This difference provides additional advantages.

Servlets need to be recompiled and deployed whenever there is a change to the page presentation, so they are best used where such changes are not required. Use servlets to generate binary data—such as image files—dynamically, and to perform complex processing with no presentation component.

Separating logic and presentation

The JavaServer Pages API provides tags that make it easy for a Web-page developer to add dynamic content to a Web page without writing Java code. The application logic in the page can be separated from page format and design. This separation supports multitiered development. An application developer can build EJBs, JavaBeans, and custom tag libraries. The page author needs only know how to call these components and what arguments to pass.

Application partitioning

In a typical architecture for multitier applications, a Web server communicates with a client via HTTP, with a transaction server hosting components that handle database transactions. JSPs make it easier to partition and maintain an application on multiple servers. The JSP runs on the Web server and can be updated whenever the page designer needs to change elements of the presentation. The components called by the JSP run on the transaction server, or on a cluster of transaction servers, and can be updated whenever the business logic needs to change.

You can also separate request handling from presentation using JSPs as **front components** and **presentation components**. A front component receives a request from the client, creates, updates, or accesses server components, then forwards the request to a presentation component. A presentation component incorporates fixed template data and returns the response to the client. Both types of JSP typically use custom actions to access the server-side data.

Syntax summary

This section lists the most useful syntax elements available in the JavaServer Pages API with simple usage examples. For complete details, see the JavaServer Pages 1.2 specification, available at http://java.sun.com/products/jsp/download.html. For a reference card that includes all the attributes of tags and directives, see the JavaServer Pages Syntax Card at http://java.sun.com/products/jsp/syntax.pdf.

Directives

Directives are messages to the JSP engine that provide global information for the page or include a file of text or code. Directives begin with the character sequence <%@ followed by the name of the directive and one or more attribute definitions. They end with the character sequence %>.

There are three directives: page, include, and taglib.

Page directive

The page directive defines attributes that apply to an entire JSP, including language, the class being extended, packages imported for the entire page, the size of the buffer, and the name of an error page. For example:

```
<%@ page language="java" import="mypkg.*"
       session="true" errorPage="ErrorPage.jsp" %>
```

For more information about error pages, see “Error handling” on page 450.

Include directive

The include directive includes a static file, parsing the file’s JSP elements:

```
<%@ include file="header.htm" %>
```

Include directive and include standard tag Note that the include directive parses the file’s contents, while the include tag does not.

Taglib directive

The taglib directive defines the name of a tag library and its prefix for any custom tags used in a JSP:

```
<%@ taglib uri="http://www.mycorp/printtags"
       prefix="print" %>
```

If the tag library includes an element called doPrintPreview, this is the syntax for using that element later in the page:

```
<print:doPrintPreview>
...
</print>
```

For more information, see “Customized tag libraries” on page 449.

Scripting elements

Scripting elements manipulate objects and perform computations. The character sequence that precedes a scripting element depends on the element’s type: <% for a scriptlet, <%= for an expression, and <%! for a declaration. Scriptlets, expressions, and declarations are all closed with the sequence %>.

Scriptlets

Scriptlets contain a code fragment valid in the scripting language:

```
<% cart.processRequest(request); %>
```

Expressions

Expressions contain an expression valid in the page scripting language:

```
Value="<%= request.getParameter("amount") %>"
```

Declarations

A declaration declares variables or methods valid in the page scripting language (usually Java, but other languages can be defined in the page directive):

```
<%! Connection myconnection; String mystring; %>
```

Comments

There are two kinds of comments:

- HTML comments optionally contain an expression, and are sent to the client and can be viewed in the page source:

```
<!-- Copyright (C) 2001 Acme Software -->
```

- Hidden comments document the source file and are not sent to the client:

```
<%-- Add new module here --%>
```

Standard tags

Standard tags perform common actions. The useBean, getProperty, and setProperty tags are all used with JavaBeans components. The useBean id attribute is the name of the bean and corresponds to the name attribute for getProperty and setProperty.

<jsp:useBean>

The useBean tag locates or instantiates a JavaBeans component:

```
<jsp:useBean id="labelLink" scope="session"
class="LinkBean.labelLink" />
```

The bean class and classes required by the bean class must be deployed under a JavaCode base that is available to the Web Application where the JSP is installed. See “Java classes” on page 373 for more information.

<jsp:getProperty>

The getProperty tag gets the value of a JavaBeans component property so that you can display it in a result page:

```
<jsp:getProperty name="labelLink" property="url" />
```

<jsp:setProperty>

The setProperty tag sets a property value or values in a JavaBeans component:

```
<jsp:setProperty name="labelLink" property="url"
value="<%= labelLink.getURL() %>"/>
```

<jsp:include> The include tag includes a static file or sends a request to a dynamic file:

```
<jsp:include page="/jsp/datafiles/ListSort.jsp" />
```

<jsp:forward> The forward tag forwards a client request to an HTML file, JSP file, or servlet for processing:

```
<jsp:forward page="/jsp/datafiles/ListSort.jsp" />
```

<jsp:plugin> The plugin tag downloads plug-in software to the Web browser to execute an applet or JavaBeans component:

```
<jsp:plugin type="applet" code="Calc.class"
       codebase="/utils/applets" >
```

Objects and scopes

When a JSP processes a request, it has access to a set of implicit objects, each of which is associated with a given scope. Other objects can be created in scripts. These created objects have a scope attribute that defines where the reference to that object is created and removed.

Scopes

There are four scopes:

- Page – accessible only in the page in which the object is created. Released when the response is returned or the request forwarded.
- Request – accessible from pages processing the request in which the object is created. Released when the request has been processed.
- Session – accessible from pages processing requests in the same session in which the object is created. Released when the session ends.
- Application – accessible from pages processing requests in the same application in which the object is created. Released when the runtime environment reclaims the ServletContext.

References to the object are stored in the PageContext, Request, Session, or Application object, according to the object's scope.

Implicit objects

The following implicit objects are always available within scriptlets and expressions:

- request – the request triggering the service invocation.
- response – the response to the request.
- pageContext – the page context for this JSP.
- session – the session object created for the requesting client (if any).
- application – the servlet context obtained from the servlet configuration, as in the call getservletConfig().getContext().
- out – an object that writes to the output stream.
- config – the ServletConfig for this JSP.
- page – the instance of this page's implementation class that is processing the current request. A synonym for *this* when the programming language is Java.

For information about the scope and type of each implicit object, see the JavaServer Pages Syntax Card at http://java.sun.com/products/jsp/syntax.pdf.

The exception implicit object

If the JSP is an error page (the page directive's isErrorPage attribute is set to true), the following implicit object is also available:

- exception – the uncaught Throwable that resulted in the error page being invoked.

For more information, see "Error handling" on page 450.

Application logic in JSPs

The application logic in JSPs can be provided by components such as servlets, JavaBeans, and EJBs, customized tag libraries, scriptlets and expressions. Scriptlets and expressions hold the components and tags together in the page.

JavaBeans

You can easily use JavaBeans components in a JSP with the useBean directive. For more information, see "<jsp:useBean>" on page 445.

Enterprise JavaBeans

To use an EJB component, write a scriptlet that uses JNDI to establish an initial naming context for the EJB's home interface. For more information about establishing the naming context and calling remote methods on the EJB's home interface, see Chapter 8, "Creating Enterprise JavaBeans Clients." This example, *HotSpots.jsp*, uses an EJB called HotSpots to return a list of places to go that fit a category and date requirement passed in the HTTP request:

```
<HTML>
<HEAD></HEAD><BODY>
<%@ page language="java" import="hotspots.*"
    session="true" errorPage="ErrorPage.jsp" %>
<%@ include file="header.htm" %>
<h1>HotSpots</h1>
<%-- GET SEARCH PARAMETERS FROM REQUEST OBJECT --%>
<%
    String category =
       request.getParameter("category");
    String date = request.getParameter("date");
%>
<%-- CREATE FORM WITH SEARCH PARAMETERS --%>
<form action="HotSpots.jsp">
    <table border=0>
    <tr><td>Category:</td><td>
    <input name="category" value="<%= category %>">
    </td></tr>
    <tr><td>Date:</td><td><input name="date"
       value="<%= date %>"></td>
    </tr>
    </table>
    <br><input type="submit" value="Search">
</form>
<%-- INSERT TABLE TO SHOW RESULTS AND USE SCRIPTLET TO
GET A REFERENCE TO THE HOTSPOTS HOME INTERFACE AND GET
A RESULT SET--%>
<p><table border=1 cellpadding=4>
<tr><th>Book</th><th>Place</th><th>Date</th>
    <th>Price</th></tr>
<%
if ( category !=null &&  date!=null) {
 try {
    java.util.Properties
       p = new java.util.Properties();
    p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
       "com.sybase.ejb.InitialContextFactory");
    p.put(javax.naming.Context.PROVIDER_URL,
       "iiop://localhost:9000");
```

```
        p.put(javax.naming.Context.SECURITY_PRINCIPAL,
            "jagadmin");
        p.put(javax.naming.Context.SECURITY_CREDENTIALS,
             "");
        javax.naming.InitialContext ctx =
            new javax.naming.InitialContext(p);
        HotSpotsHome home = (HotSpotsHome)
            ctx.lookup("HotSpots");
        HotSpots hotSpots = home.create();
        java.sql.ResultSet rs =
            com.sybase.helper.IDL.getResultSet(
                hotSpots.getList(category, date) );
        while (rs.next()) {
    %>
    <%-- POPULATE TABLE WITH RESULT SET --%>
    <tr><td><a href=Payment.jsp?trip=
        <%= rs.getInt("trip_id") %>
        &amount=<%= rs.getDouble("price") %> >
        <img src="images/bookit.gif"></a></td>
    <td><%= rs.getString("place") %></td>
    <td><%= rs.getDate("date") %></td>
    <td><%= rs.getDouble("price") %></td>
    </tr>

    <%-- CLOSE WHILE LOOP AND TRY CATCH BLOCK --%>
    <%
        }
     } catch (Exception e) {
            out.println(e);
     }
    }
    %>
    </table>
    </BODY></HTML>
```

Customized tag libraries

Customized tag libraries, also called tag extensions, extend the capabilities of JSPs. Tag libraries define a set of actions to be used within a JSP for a specific purpose, such as handling SQL requests.

JSP authors can use tag libraries whether they are editing a page manually or using an authoring tool. To associate a tag library with the page, the page author uses a taglib directive that identifies the tag library's URI (see "Taglib directive" on page 444). The URI identifying the tag library is associated with a Tag Library Descriptor (TLD) file and with tag handler classes. Tag libraries are usually packaged as JAR files with a tag library descriptor file named *META-INF/taglib.tld*.

A tag handler is a Java class that defines the semantics of an action. The implementation class for the JSP instantiates a tag handler object for each action in the page. Tag handler objects implement the javax.servlet.jsp.tagext.Tag interface which defines basic methods required by all tag handlers, including doStartTag and doEndTag. The BodyTag interface extends the Tag interface by adding methods that enable the handler to manipulate its body.

You can use the same tag library in multiple Web applications by placing the JAR file containing the tag library in the EAServer *extensions* subdirectory.

Error handling

When a client request is processed, runtime errors can occur in the body of the implementation class for the JSP or in Java code that is called by the page. These exceptions can be handled in the code in the JSP using the Java language's exception mechanism.

Uncaught exceptions

Any exceptions that are thrown from the body of the implementation class and are not caught can be handled using an error page that you specify using a page directive. Both the client request and the uncaught exception are forwarded to the error page. The java.lang.Throwable exception is stored in the javax.ServletRequest instance for the client request using the putAttribute method, using the name javax.servlet.jsp.jspException.

Using an error page JSP

If you specify a JSP as the error page, you can use its implicit exception variable to obtain information about the exception. The exception variable is of type java.lang.Throwable and is initialized to the Throwable reference when the uncaught exception is thrown.

To specify an error page for a JSP, set its errorPage attribute to the URL of the error page in a page directive:

```
<%@ page errorPage="ErrorPage.jsp" %>
```

To define a JSP as an error page, set its isErrorPage attribute to true in a page directive:

```
<%@ page isErrorPage="true" %>
```

This sample error page JSP uses the exception variable's toString method to return the name of the actual class of this object and the result of the getMessage method for the object. If no message string was provided, toString returns only the name of the class.

The example also uses the getParameterNames and getAttributeNames methods of the request object to obtain information about the request.

```
<%@ page language="java" import="java.util.*"
    isErrorPage="true" %>
<H1 align="Center">Exceptions</H1>
<br><%= exception.toString() %>
<%! Enumeration parmNames; %>
<%! Enumeration attrNames; %>
<br>Parameters:
<%  parmNames = request.getParameterNames();
    while (parmNames.hasMoreElements()) {
%>
    <br><%= parmNames.nextElement().toString() %>
<%
    }
%>
<br>Attributes:
<%  attrNames = request.getAttributeNames();
    while (attrNames.hasMoreElements()) {
%>
       <br><%= attrNames.nextElement().toString() %>
<%
    }
%>
```

Using JSPs in EAServer

For JSPs to run in EAServer, they must belong to a Web application. In addition, you can map servlets to JSPs. When the servlet is called, the corresponding JSP is invoked. This section discusses:

JSP and EAServer overview

EAServer fully supports the features described in the JavaServer Pages 1.2 specification as well as mapping requests to JSPs as described in the Java Servlet 2.3 specification. In EAServer the JSP Engine is implemented as a generic servlet, which is referred to as the JSP servlet. The JSP servlet handles runtime translation and compilation of JSPs, if required, as well as invoking the generated servlet for a given JSP.

The JSP servlet supports translation of JSPs containing JSP standard directives, standard actions, custom tags and scripting elements such as declarations, scriptlets and expressions. For JSPs that include custom JSP tags, a tag handler is loaded every time it is needed. Tag handlers are not pooled. The JSP servlet also supports all the semantics associated with the “extends” attribute.

A Web application is a collection of resources that is mapped to a specific Uniform Resource Identifier (URI) prefix. These resources may include JSPs, servlets, HTML files, and images. The URI that is stored in the request data structure is used to retrieve a JSP. The JSP Servlet creates a unique name for a generated servlet. Generated servlet names are stored in a hash table. For a given request URI, the JSP Servlet determines which generated servlet name it corresponds to. It then looks up the generated servlet name in the hash table; an entry in the hash table indicates that the JSP has been precompiled.

If a JSP is not precompiled, the JSP servlet invokes the compiler and saves the generated files in the appropriate directory. It then executes the page by invoking the _jspService method on the generated servlet.

If a JSP is precompiled, the JSP servlet compares the timestamp of the JSP and all its nested include files, if any, with the timestamp of the generated servlet. If any time stamp of the JSP is more recent than that of the generated servlet, the JSP is recompiled. If the generated servlet is current, the JSP Servlet creates a new instance of the precompiled servlet class and calls _jspService method on it.

The JSP Servlet uses CLASSPATH from ServletConfig for compiling the generated servlet. To change the directory where servlets are generated, set the “scratchdir” parameter as one of the parameters in ServletConfig. “scratchdir” is passed to the “init” method on the JSP Servlet; you can set it by using one of the init-args on the Servlet Properties tab.

JSP 1.2 highlights

The JSP 1.2 specification extends JSP 1.1 in a number of ways:

- Uses Servlet 2.3 as the foundation for its semantics.
- Defines the XML syntax for JSPs.
- Provides for translation-time validation of JSPs.

  A new compilation phase has been added that gives custom tag libraries the opportunity to examine an XML view of the parsed page, and throw a translation time exception if problems are detected.
- The ability for tag libraries to include event listener classes. The listeners are listed in the tag library descriptor and the JSP container automatically instantiates the listener classes and registers them in a similar way to *web.xml*. Essentially, the mechanism locates the TLDs in the Web Application (either in *WEB-INF/classes* or *WEB-INF/lib*), reads their <listener> elements, and regards them as an extension of those listed in *web.xml*.
- Better specification of the tag handler contract – tag handlers are Java classes that implement specific servlet interfaces. JSP 1.2 introduces some new interfaces and changes some existing interfaces to provide better support.
- Improvements on authoring support – extends tag library functionality.
- Better I18N support – implements the `javax.servlet.ServletResponse.setContentType()` method to provide support for dynamic content-type.
- Fixes the “flush before you include” limitation in JSP 1.1.

For detailed information about JavaServer Pages technology, see the Java software Web site at http://java.sun.com/products/jsp.

Compiling JSPs

When you create a JSP using EAServer Manager, the load during startup option determines if your JSPs are compiled at server start-up or when the JSP is first called. You can also use a command-line utility to compile your JSPs manually. This allows you to debug and test your JSPs without running the server. This section describes the JSP compiler supplied with your EAServer installation.

jspc compiler

The *%JAGUAR%\bin\jspc.bat* (Windows) and *$JAGUAR/bin/jspc.sh* (UNIX) compiler provides you with several options for compiling your JSPs. This section uses *jspc.sh* for demonstration purposes. All options are valid for both systems.

Usage

```
jspc.sh <options> <jsp files>|<jsp dirs>
```

options include:

- **-webapp <*dir*>** The name of the Web application directory, relative to the *$JAGUAR/Repository/WebApplication* directory. For example, *MyWebApp* identifies the *$JAGUAR/Repository/WebApplicationMyWebApp* directory.
- **-uriroot <*dir*>** Complete path name of the Web application directory. For example, *$JAGUAR/Repository/WebApplication/MyWebApp*.
- **-d <*dir*>** The name of the output directory to which the compilation results are stored.
- **-keep** Keep the generated Java source files.

jsp files include any number of:

- **<*file*>** A file to be parsed as a JSP.
- **-jspdir <*dir*>** A directory containing a Web-app, all JSPs are recursively parsed.

Examples

This section provides examples of the various jspc compiler options:

```
jspc.sh -webapp MyWebApp /pets/cat.jsp /pets/dog.jsp
```

The compiler identifies the *$JAGUAR/Repository/WebApplication/MyWebApp* directory as the Web application directory and compiles the *cat.jsp* and *dog.jsp* files located in the *$JAGUAR/Repository/WebApplication/MyWebApp/pets* directory.

```
jspc.sh -uriroot
$JAGUAR/Repository/WebApplicaton/MyWebApp -d
$JAGUAR/work /pets/cat.jsp /pets/dog.jsp
```

The compiler identifies the *$JAGUAR/Repository/WebApplication/MyWebApp* directory as the Web application directory and compiles the *cat.jsp* and *dog.jsp* files, which are located in the *pets* subdirectory, and outputs the results to the *$JAGUAR/work directory*.

```
jspc.sh -uriroot
$JAGUAR/Repository/WebApplicaton/MyWebApp -d
$JAGUAR/work -jspdir
```

```
$JAGUAR/Repository/WebApplicaton/MyWebApp
```

The compiler identifies the *$JAGUAR/Repository/WebApplication/MyWebApp* directory as the Web application directory and compiles all JSP files contained in it and any subdirectories, and outputs the results to the *$JAGUAR/work directory*.

```
jspc.sh -uriroot
$JAGUAR/Repository/WebApplicaton/MyWebApp -d
$JAGUAR/work -keep /pets/dog.jsp /pets/cat.jsp
```

the -keep option notifies the compiler to keep all generated Java files in the output directory, along with the compiled JSP files.

JSP file locations

JSPs are contained within Web applications. JSP source code and class files are stored relative to the Web application to which it belongs.

Saving Java source code

Normally, EAServer deletes the Java source code after compiling a JSP. To keep the generated source code to view or use in a debugger:

1 Display the properties for the Web application in which the JSP is installed.

2 On the Advanced tab, set the com.sybase.jaguar.webapplication.keepgenerated property to “true”.

Source and class file locations

EAServer generates JSP servlet source files and classes in this directory under your EAServer installation:

```
work\server\Servlet\WebApp-WAName
```

Where *server* is the name of your server, and *WAName* is the name of your Web application.

Creating and configuring JSPs in EAServer

JSPs in EAServer must be created in a Web application. If necessary, create the Web application to contain the JSPs as described in Chapter 21, "Creating Web Applications." You can create new JSPs in EAServer Manager or import them from existing JSP source files.

❖ **Creating or importing JSPs**

1 In EAServer Manager, select the Web Application folder.

2 Select an existing Web application to which you are adding a JSP, then import the JSP or create a new one as follows:

- **Creating new JSP files** Select File | New Web Component. Enter the name of the Web component (JSP) and click OK.

  EAServer creates the JSP under the Web applications folder. For example, if you name your JSP *testjsp* and it belongs to the *MyWebApp* application, then the location is *EAServer_home/Repository/WebApplication/MyWebApp/testjsp*.

- **Importing JSP files** If you have existing JSPs that you want to add to your Web application or if you create JSPs with another editing tool, you can copy them to this location, making sure the JSP you copy and the name you enter for the JSP match.

❖ **Configuring the JSP properties**

To configure your JSP, double-click the JSP or highlight the JSP and select File | Web Application Component Properties. Complete the information described below:

1 General properties – select this tab to enter general parameters for your JSP:

- Description – a brief description of the JSP file.
- Web Component Type – select JSP.
- JSP File Name – the name and path of the JSP file. The path is relative to the Web application context. For example, if you enter */work/test.jsp*, the JSP will be placed in *EAServer/Repository/WebApplication/appname/work/test.jsp*, where *EAServer* is the EAServer installation directory and *appname* is the Web application name. The JSP file must include the *.jsp* extension.

- Startup Load Sequence Position – enter a number that indicates when the JSP loads in relation to other JSP files when the Web application starts. This option applies only if Load During Startup is true. The lower the number the earlier it loads; 1 indicates that this JSP loads first. If a JSP is dependent on another JSP that requires time to initialize, specify the JSP that requires additional initialization time to load first. JSPs with a startup load sequence position of 0 loads last.
- Load During Startup – this option compiles and translates the JSP into a servlet at start-up. If you do not select this option, the JSP is compiled when it is first called.

2 Init-Args – select this tab to enter the initialization parameters associated with the JSP. The initialization property values are listed in the `com.sybase.jaguar.servlet.init.args` property of the Advanced tab:

- Add – enter the Initialization Property Name. Add a default value for your parameter in the Property Value window.
- Modify – highlight the argument you want to modify and click Modify. Make your modifications and click OK.
- Delete – highlight the argument you want to delete and click Delete.

3 Advanced – to improve JSP performance, set the value of the `com.sybase.jaguar.webapplicaton.jspc-interval` property, which determines if and when the JSP runtime checks whether a JSP is current. Set the property value to an integer.

- If set to a negative number, the JSP runtime never checks.
- If set to 0, the JSP runtime always checks.
- To specify the number of seconds before the next check, set the value to a number greater than 0. If a request comes in before the time expires, the JSP is not checked.

Complete the rest of the properties as you would for a servlet.

To configure security for your JSP, see Chapter 3, "Using Web Application Security," the *EAServer Security Administration and Programming Guide*.

❖ **Editing the JSP source**

EAServer supplies an editor for creating and modifying your JSP files; however, you can use any text editor to perform the same tasks. To edit the JSP in EAServer Manager:

1 Open the Web Application folder and select the Web application to which the JSP belongs.

2 Highlight the JSP.

3 Select File | Edit JSP. An editor displays where you can view and modify your JSP. Locate other files for editing by selecting File | Open. When you are finished, select File | Save.

❖ **Deleting a JSP**

1 Open the Web Application folder and select the Web application to which the JSP file belongs.

2 Highlight the JSP.

3 Select File | Delete Web Application Component.

Internationalization

EAServer supports international versions of your Web application resources: Servlets, static Web pages, and so on. For more information, see “Localizing Web applications” on page 402.

Mapping JSPs

EAServer supports path mappings as described in the Java Servlet 2.3 specification. Mappings are defined at the Web application level. Refer to Chapter 21, “Creating Web Applications” for information about Request path mappings.

Page caching

EAServer supports page caching, which improves the performance of servlet and JSP requests. When page caching is enabled for a servlet or JSP Web component, the cache is checked before invoking the Web component. For more information, see “Dynamic page caching” in Chapter 5, “Web Application Tuning,” in the *EAServer Performance and Tuning Guide*.

Filters

EAServer supports servlet filters as described in the Java Servlet 2.3 specification. Filters are defined at the Web application-level. For information on creating filters, see Chapter 23, “Using Filters and Event Listeners.”

PART 8

Advanced Features

This part explains how to use advanced features such as connection caching, result sets, messaging, threads, or pseudocomponents.

CHAPTER 25

Sending Result Sets

This chapter describes how component methods can return results to the client that called them.

Overview

Component methods use either Java classes or Server-Library C routines to send rows, as follows:

- Java components send results sets with the classes in the com.sybase.jaguar.sql package, as discussed in "Sending result sets with Java" on page 464.
- PowerBuilder components send result sets with the DataStore, ResultSet, and ResultSets interfaces, as described in "Sending result sets from a PowerBuilder component" on page 469.
- ActiveX components send result sets with the IJagServerResults interface, as described in "Sending result sets from an ActiveX component" on page 470.
- C components call EAServer C-language routines to send rows, as described in "Sending result sets from a C or C++ component" on page 475.

Components that interact with third-tier servers should use EAServer's Connection Management feature to realize improved performance. See Chapter 26, "Using Connection Management" for more information.

Note When you are defining a method in EAServer Manager, be sure to indicate whether the method returns row results.

Sending result sets with Java

Java components send results sets with 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.

The JContext class contains static factory methods to return objects that implement these interfaces.

Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference* contains reference pages for all classes and interfaces.

Note You cannot send a result set unless the IDL definition of the component method returns TabularResults::ResultSet or TabularResults::ResultSets.

Methods in Java components that use Java/IDL datatypes must be declared to return TabularResults.ResultSet or TabularResults.ResultSet if the method returns result sets. However, you can still use the JServerResultSetMetaData and JServerResultSet interfaces to implicitly return results. Just return null as the method's return value. Alternatively, you can construct the equivalent Java datatypes for the IDL TabularResults::ResultSet and TabularResults::ResultSets types. Call the getResultSet method in the class com.sybase.CORBA.jdbc11.IDL to convert a java.sql.ResultSet instance into a TabularResults.ResultSet instance that can be returned by the method.

Forwarding a ResultSet object

You can use the steps below to forward results from a JDBC query directly to the client:

1 Query the remote server. Use java.sql.Statement or one of its extensions; the appropriate method depends on the query being sent.

2 Handle the results of the query. For each ResultSet returned by the query, call JContext.forwardResultSet(ResultSet) to forward the rows to the client.

3 If your component uses IDL/Java datatypes, return null as the method's return value.

For an example that calls JContext.forwardResultSet(ResultSet), see "Java Connection Manager example" on page 487. You can find more examples in the source for the EAServer sample Java components, available in your EAServer installation directory.

Instead of calling JContext.forwardResultSet(ResultSet), Java components that use IDL/Java datatypes can call the IDL.getResultSet(java.sql.ResultSet) method to convert ResultSet object to TabularResults.ResultSet object, then return the converted object as the method's return value.

Sending results row-by-row

Use the sequence of calls below to define and send a result set row-by-row. Use these calls when building a result set from a non-JDBC source, or when the java.sql.ResultSet returned by a database query cannot be sent as-is to the client.

JServerResultSet sequence of calls

Here are the calls to construct a result set and send it row-by-row:

1 Create a JServerResultSetMetaData object by calling JContext.createServerResultSetMetaData().

2 Call the JServerResultSetMetaData methods to define the format of the result rows, as follows:

   a JServerResultSetMetaData.setColumnCount(int) to specify the number of columns in each row.

   b For each column, call JServerResultSetMetaData.setColumnType(int, int) to specify the datatype.

c For columns that have a variable length datatype, call JServerResultSetMetaData.setColumnDisplaySize(int, int) to specify the maximum length for column values.

d Call other JServerResultSetMetaData methods to specify other column attributes as needed.

3 Create a JServerResultSet object by calling JContext.createServerResultSet().

4 Call JServerResultSet.next() to position the result set's cursor at the first row.

5 For each row to be sent:

- For each column, call the appropriate JServerResultSet.set<Object>(int, <Object>) method to set the column value.
- Call JServerResultSet.next() to send the row.

6 If sending a single result set or if using JDBC types, call JServerResultSet.done() to indicate that all rows have been sent in the current result set.

7 If your component uses IDL/Java datatypes, use the com.sybase.CORBA.IdlResultSet class to convert the result set to a TabularResults.ResultSet instance. See Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference* for details.

You can repeat steps 4 to 6 to send or create another result set that has the same metadata using the same JServerResultSet object. Repeat steps 1 to 6 to send or create another result set that requires different metadata.

You cannot return multiple result sets unless the method's IDL definition returns TabularResults::ResultSets.

JServerResultSet example

The example method below sends three rows with three columns each. Note that exceptions are not caught in the example; the server logs any uncaught exceptions that are thrown in a method call:

```
public void send_rows (IntegerHolder ih) throws
      JException, SQLException
{

  // Declare the constant 'pi'
```

```
final double pi = 3.1414;  // Create the metadata object.
JServerResultSetMetaData
     jsrsmd = JContext.createServerResultSetMetaData();

// There will be 3 columns in the result set.
jsrsmd.setColumnCount(3);

// The first column has datatype INTEGER and name 'one'.
jsrsmd.setColumnType(1, Types.INTEGER);
jsrsmd.setColumnName(1, "one");

// The second column has datatype VARCHAR and name 'two'.
jsrsmd.setColumnType(2, Types.VARCHAR);
jsrsmd.setColumnName(2, "two");

// The third column has datatype DOUBLE and name 'three'.
jsrsmd.setColumnType(3, Types.DOUBLE);
jsrsmd.setColumnName(3, "three");

// Create the result set object.
JServerResultSet jsrs = JContext.createServerResultSet(jsrsmd);

// Position the cursor.
jsrs.next();

// First row values: 1, "first", pi
jsrs.setInt(1, 1);
jsrs.setString(2, "first");
jsrs.setDouble(3, pi);

// Send the row.
jsrs.next();

// Second row values: 2, "second", pi * 2
jsrs.setInt(1, 2);
jsrs.setString(2, "second");
jsrs.setDouble(3, pi * 2.0);

// Send the row.
jsrs.next();

// Third row values: 3, "third", pi * 3
jsrs.setInt(1, 3);
jsrs.setString(2, "third");
jsrs.setDouble(3, pi * 3.0);
```

```
  // Send the row.
  jsrs.next();

  // Demarcate the end of the result set by calling done().
  jsrs.done();
}
```

The fragment below shows client-side code to call the stub and print the rows to the console. For more information about coding the client to retrieve result sets from components, see “Return result sets” on page 199.

```
try {
  ih = new IntegerHolder();
  comp.send_rows(ih);

  ResultSet rs = comp.getResultSet();
  ResultSetMetaData rsmd = rs.getMetaData();

  StringBuffer row = new StringBuffer("");
  for (int i = 1; i <= rsmd.getColumnCount(); i++)
  {
    row.append(rsmd.getColumnName(i));
    if (i < rsmd.getColumnCount())
      row.append("\t");
  }

  System.out.println(row);

  while(rs.next())
  {
    row = new StringBuffer("");
    for (int i = 1; i <= rsmd.getColumnCount(); i++)
    {
      row.append(rs.getString(i));
      if (i < rsmd.getColumnCount())
        row.append("\t");
    }
    System.out.println(row);
  }

  // Discard any remaining results.
  while(comp.getMoreResults())
  {
    rs = comp.getResultSet();
  }

}
```

```
catch (Exception e) {
  System.out.println("Exception: " + e.getMessage());
  e.printStackTrace();
}
```

Sending result sets from a PowerBuilder component

This section briefly summarizes how PowerBuilder components return result sets. For detailed instructions, see the *Application Techniques* manual in the PowerBuilder documentation.

ResultSet objects

PowerBuilder components use DataStore objects to store and manipulate result sets. The DataStore works like a DataWindow object, except that it has no visual attributes.

Result sets are returned and retrieved using these PowerBuilder objects:

- **ResultSet** Represents a single result set to be returned to the client. You cannot directly manipulate the contents. Instead, call the DataStore functions discussed below to populate a ResultSet from the contents of a DataStore or vice-versa. Methods in the component interface that return the TabularResults::ResultSet IDL type are represented by PowerBuilder functions that return ResultSet.
- **ResultSets** Holds a list of 0 or more ResultSet instances; the list is stored as the ResultSetList property. Methods in the component interface that return the TabularResults::ResultSets IDL type are represented by PowerBuilder functions that return ResultSets.

Forwarding result sets

You can retrieve result sets from a remote database and forward them to a client as follows:

1. When results are ready to be retrieved, associate the transaction object with a DataStore instance by calling the DataStore.SetTransObject function.
2. Populate the DataStore by calling the Retrieve function.
3. Call the DataStore.GenerateResultSet function and assign the returned object to a ResultSet instance.
4. If the method returns ResultSets, add the result set to the list in the ResultSets.ResultSetList property. Otherwise, return the result set directly.

Sending result sets from an ActiveX component

ActiveX methods return row results using the IJagServerResults interface. Both a custom (native C++) and IDispatch version of the interface are available, as documented in these chapters of the the *EAServer API Reference*:

- Chapter 2, "ActiveX C++ Interface Reference"
- Chapter 3, "ActiveX IDispatch Interface Reference"

To use these methods in your component, you must first register the *jagaxwrap.dll* programmable object on your machine. If you are developing on a machine that has EAServer installed on it, *jagaxwrap.dll* is already registered.

The following sections describe the two methods for returning result sets:

- "Forwarding a result set with ResultsPassthrough" on page 470 describes how to forward an ODBC or Client-Library result set to the client. Use this method when you want to send query results as-is.
- "Sending results row-by-row" on page 470 describes the sequence of calls to define a result set's columns and send the result set row-by-row. Use this method when you must manufacture a result set from scratch. You can also use this method to filter rows or columns from the results of a remote-database query.

Note You cannot send a result set unless the IDL definition of the component method returns TabularResults::ResultSet or TabularResults::ResultSets.

Forwarding a result set with ResultsPassthrough

You can call the ResultsPassthrough method after you have sent a remote-database query with ODBC or Client-Library calls. This method passes either the current result set or all the result sets on the connection (or command). See the ResultsPassthrough reference page for examples.

Sending results row-by-row

The steps below describe the call sequence for sending a result set from scratch.

1 Instantiate an IJagServerResults interface pointer.

2 Call BeginResults, specifying the number of columns in the result set.

3 For each column, call DescribeCol to describe the column number, column name, length of the column name, datatype, precision, and size of the column. If a column represents a cash value, call ColAttributes to set the "COLUMN_MONEY" attribute.

   This step and the next step may be combined; you can describe columns and bind them in the same loop.

4 For each column, call BindCol, specifying the size and location of the variable where the column's data value and length can be read.

5 For each row, update the variables containing the column's data value and length, then call SendData to send the row to the client.

6 Call EndResults after all rows have been sent.

Result sets row-by-row: C++ example

The C++ fragment below shows an ActiveX method implementation that returns a result set:

```
// EAServer includes
#include <stdio.h>
#include <sql.h>
#include <jagctx.h>
#include <JagAxWrap.h>
#include <JagAxWrap_i.c>
#include <jagpublic.h>

STDMETHODIMP CAXRSDemo::SendRows()
{
    HRESULT            hr;
    IJagServerResults  *p_ijsrs;
    CLSID              clsid_jsrs;
    BSTR               colName;
    BSTR               SQLType;
    VARIANTARG         bindVar;
    long               rowCount;
    LONG               intCol;
    short              intColInd = 0;
    BSTR               strCol;
    short              strColInd = 0;
    DOUBLE             doubleCol;
    short              doubleColInd = 0;
```

```
// Create an IJagServerResults interface pointer
hr = CLSIDFromProgID(
         L"Jaguar.JagServerResults.1",
         &clsid_jsrs);
// ... Deleted error checking ...
hr = CoCreateInstance(clsid_jsrs, NULL,
                      CLSCTX_INPROC_SERVER,
                      IID_IJagServerResults,
                      (void**)&p_ijsrs);
// ... Deleted error checking ...

// Result set has three columns.
hr = p_ijsrs->BeginResults(3);
// ... Deleted error checking ...

//
// First column has datatype SQL_INTEGER,
// has name "one", and can be NULL.
//
colName = SysAllocString(L"one");
SQLType = SysAllocString(L"SQL_INTEGER");
hr = p_ijsrs->DescribeCol( 1, colName, SQLType,
                sizeof(intCol), 0, 0,
                VARIANT_TRUE);
// ... Deleted error checking ...

//
// Bind first column to intCol
//
VariantInit(&bindVar);
bindVar.vt = VT_I4 | VT_BYREF;
bindVar.plVal = &intCol;

hr = p_ijsrs->BindCol( 1, bindVar, sizeof(intCol),
                &intColInd);
// ... Deleted error checking ...

//
// Second column has datatype SQL_VARCHAR,
// maximum length 32, name "two", and can
// be null.
//
hr = SysReAllocString(&colName, L"two");
// ... Deleted error checking ...

hr = SysReAllocString(&SQLType, L"SQL_VARCHAR");
```

```
    // ... Deleted error checking ...

    hr = p_ijsrs->DescribeCol( 2, colName, SQLType,
                    32, 0, 0,
                    VARIANT_TRUE);
    // ... Deleted error checking ...

    //
    // Allocate a BSTR and bind the second column
    // to it. Later,
we'll use SysReAllocString() to set
    // values for transfer.
    //
    strCol = SysAllocString(L"");
    VariantInit(&bindVar);
    bindVar.vt = VT_BSTR | VT_BYREF;
    bindVar.pbstrVal = &strCol;
    // ... Deleted error checking ...

    //
    // Third column has datatype SQL_DECIMAL with
    // precision of 5 and scale of 3
    // Column name is "three", and the column can be n
ull.
    //
    hr = SysReAllocString(&colName, L"three");
    // ... Deleted error checking ...

    hr = SysReAllocString(&SQLType, L"SQL_DECIMAL");
    // ... Deleted error checking ...


    hr = p_ijsrs->DescribeCol( 3, colName, SQLType,
                    0, 5, 3,
                    VARIANT_TRUE);
    // ... Deleted error checking ...

    //
    // Bind the third column to doubleCol.
    //
    VariantInit(&bindVar);
    bindVar.vt = VT_R8 | VT_BYREF;
    bindVar.pdblVal = &doubleCol;
    // ... Deleted error checking ...

    //
```

```
    // Now send the rows.
    //
    rowCount = 0;
    // First row: 1, "uno", 3.141
    intCol = 1;
    hr = SysReAllocString(&strCol, L"uno");
    // ... Deleted error checking ...

    doubleCol = 3.141;
    hr = p_ijsrs->SendData();
    // ... Deleted error checking ...

    ++rowCount;
    // Second row: 2, "dos", 6.282
    intCol = 2;
    hr = SysReAllocString(&strCol, L"dos");
    // ... Deleted error checking ...

    doubleCol = 6.282;
    hr = p_ijsrs->SendData();
    // ... Deleted error checking ...

    ++rowCount;
    // Third row: 3, "tres", 9.423
    intCol = 3;
    hr = SysReAllocString(&strCol, L"tres");
    // ... Deleted error checking ...

    doubleCol = 9.423;IJagServerResults
    hr = p_ijsrs->SendData();
    // ... Deleted error checking ...

    ++rowCount;
    //
    // Done sending rows.
    //
    hr = p_ijsrs->EndResults(rowCount);
    // ... Deleted error checking ...

    return S_OK;

}
```

Sending result sets from a C or C++ component

C component methods return row results using the routines listed in Chapter 5, "C Routines Reference," in the *EAServer API Reference*.

This section describes the two different algorithms for returning results sets from C components:

- "Forwarding a result set with JagResultsPassthrough" on page 475 describes how to forward an ODBC or Client-Library result set to the client. Use this method when you want to send query results as-is.
- "Sending results row-by-row" on page 476 describes the sequence of calls to define a result set's columns and send the result set row-by-row. Use this method when you must manufacture a result set from scratch. You can also use this method to filter rows or columns from the results of a remote-database query.

Note You cannot call C routines to send a result set unless the IDL definition of the component method returns TabularResults::ResultSet or TabularResults::ResultSets.

To return result sets from C++ components:

- C++ component methods that return a single result set are defined to return a pointer to a TabularResults::ResultSet structure; use the C routines to return a single result set, as described in "Sending results row-by-row" on page 465, then return NULL in place of the structure pointer.
- C++ component methods that return multiple result sets must populate an array of TabularResults::ResultSet structures. For more information, see the generated IDL documentation for TabularResults::ResultSet under *html/ir/index.html* in your EAServer installation.

Forwarding a result set with JagResultsPassthrough

In your C or C++ component, you can call JagResultsPassthrough after you have sent a remote-database query with ODBC or Client-Library calls. JagResultsPassthrough extracts the results from a Client-Library or ODBC control structure and forwards them to the client. For details, see Chapter 5, "C Routines Reference" in the *EAServer API Reference*.

Sending results row-by-row

The steps below describe the call sequence for sending a result set from scratch.

1 Call JagBeginResults, specifying the number of columns in the result set.

2 For each column, call JagDescribeCol to describe the name, datatype, and size of the column.

JagDescribeCol accepts either ODBC or Client-Library datatypes. If you use ODBC datatypes, and the column represents money, call JagColAttributes to set the column's SQL_COLUMN_MONEY attribute.

This step and the next step may be combined; you can describe columns and bind them in the same loop.

3 For each column, call JagBindCol, specifying the locations of variables where the column's data values and lengths can be read.

4 For each row, update the variables containing the column's data value and length, then call JagSendData to send the row to the client.

5 Call JagEndResults to indicate that all rows have been sent.

Sending a result set using ODBC types

The example code below defines a method that returns a result set, using ODBC datatypes to describe the columns:

```
#define LOG_ERROR(errtext) (CS_VOID)JagLog(JAG_TRUE,\
                           "sendrows_C: rows1(): " errtext "\n")

#define MAX_STRCOL 64

/*
** Component - sendrows_C
** Method - rows1
**
**   Return a small result set to the client.
**
*/
CS_RETCODE rows1(
)
{
     JagStatus jsret;

     SQLINTEGER intcol;
     SQLINTEGER intcol_len = sizeof(SQLINTEGER);
     SQLSMALLINT intcol_ind = 0;
```

```
SQLCHAR    strcol[MAX_STRCOL + 1];
SQLINTEGER strcol_len;
SQLSMALLINT strcol_ind = 0;

SDOUBLE  doublecol;
SQLINTEGER doublecol_len = sizeof(SDOUBLE);
SQLSMALLINT doublecol_ind = 0;

SQLINTEGER rowcount = 0;

/*
** Step 1: JagBeginResults() to begin sending
**    a new result set. Number of columns is 3.
*/
jsret = JagBeginResults(3);
if (jsret != JAG_SUCCEED)
{
    LOG_ERROR("JagBeginResults() failed.");
    return CS_FAIL;
}

/*
** Steps 2 and 3: For each column, describe the
**   column's metadata and bind the column to a
**   variable from which values will be
**   read.
*/

/*
** First column is integer, bound to int_col.
*/
jsret = JagDescribeCol(1, JAG_ODBC_TYPE,
              "First", SQL_INTEGER,
              sizeof(SDWORD), 0, 0, SQL_NULLABLE);
if (jsret != JAG_SUCCEED)
{
    LOG_ERROR("JagDescribeCol(1) failed.");
    return CS_FAIL;
}

intcol_len = sizeof(SDWORD);
jsret = JagBindCol(1, JAG_ODBC_TYPE,
                   SQL_C_SLONG, &intcol,
                   intcol_len, &intcol_len, &intcol_ind);
if (jsret != JAG_SUCCEED)
{
```

```
    LOG_ERROR("JagBindCol(1) failed.");
    return CS_FAIL;
}

/*
** Second column is a string, bound to string_col.
*/
jsret = JagDescribeCol(2, SQL_VARCHAR, "Second",
              MAX_STRCOL, 0, 0, SQL_NULLABLE);
if (jsret != JAG_SUCCEED)
{
    LOG_ERROR("JagDescribeCol(2) failed.");
    return CS_FAIL;
}

/*
** Length specified as MAX_STRCOL + 1 to
** allow space for null-terminator.
*/
jsret = JagBindCol(2, SQL_C_CHAR, &strcol,
          MAX_STRCOL + 1, &strcol_len, &strcol_ind);
if (jsret != JAG_SUCCEED)
{
    LOG_ERROR("JagBindCol(2) failed.");
    return CS_FAIL;
}

/*
** Third column is a SQL_DOUBLE, bound to double_col.
** It does not allow nulls.
*/
jsret = JagDescribeCol(3, SQL_DOUBLE, "Third",
              sizeof(SDOUBLE), 0, 0, SQL_NO_NULLS);
if (jsret != JAG_SUCCEED)
{
    LOG_ERROR("JagDescribeCol(3) failed.");
    return CS_FAIL;
}

doublecol_ind = sizeof(SDOUBLE);
jsret = JagBindCol(3, SQL_C_DOUBLE, &doublecol,
       doublecol_ind, &doublecol_len, &doublecol_ind);
if (jsret != JAG_SUCCEED)
{
    LOG_ERROR("JagBindCol(3) failed.");
    return CS_FAIL;
```

```
}

/*
** Step 4: send the rows.
*/

/*
** Values for row 1:
**    1, "Uno", 1.001
*/
++rowcount;
intcol = 1;

strcpy(strcol, "Uno");
strcol_ind = strlen(strcol);

doublecol = 1.001;

jsret = JagSendData();

/*
** Values for row 2:
**   2, "Dos", 2.002
*/
++rowcount;
intcol = 2;

strcpy(strcol, "Dos");
strcol_ind = strlen(strcol);

doublecol = 2.002;

jsret = JagSendData();

/*
** Values for row 2:
**   3, "Tres", 3.003
*/
++rowcount;
intcol = 3;

strcpy(strcol, "Tres");
strcol_ind = strlen(strcol);

doublecol = 3.003;
```

```
        jsret = JagSendData();
        if (jsret != JAG_SUCCEED)
        {
            LOG_ERROR("JagSendData(1) failed.");
            return CS_FAIL;
        }

        /*
        ** Step 5: Call JagEndResults() to say that we're done.
        */
        jsret = JagEndResults(rowcount);
        if (jsret != JAG_SUCCEED)
        {
            LOG_ERROR("JagEndResults() failed.");
            return CS_FAIL;
        }

        return(CS_SUCCEED);
    }
```

Sending a result set using Client-Library types

The example code below defines a method that returns a result set, using Client-Library datatypes to describe the columns:

```
#include <jagpublic.h>
#include <ctpublic.h>

CS_RETCODE JAG_PUBLIC getResultSet (void
{

JagStatus jsret;

CS_SMALLINT intcol_ind = 0;
CS_INT intcol_len = sizeof (CS_INT) ;
CS_INT intcol     = 0 ;


CS_CHAR     strcol[MAX_STRCOL + 1];
CS_INT  strcol_len = 0;
 CS_INT strcol_ind = 0;


char *data[] = {"one", "two", "three", "for", "five",
                "six", "seven", "eight", "nine", "ten"} ;

  /*
  ** Step 1: JagBeginResults() to begin sending
  ** a new result set. Number of columns is 2
```

```
*/
jsret = JagBeginResults(2);
if (jsret != JAG_SUCCEED)
{
    LOG_ERROR("JagBeginResults() failed.");
    return CS_FAIL;
}


/* type int column*/
jsret = JagDescribeCol(1, JAG_CS_TYPE, "Int Column", CS_INT_TYPE,
             sizeof ( CS_INT) , 0, 0, CS_CANBENULL);
if (jsret != JAG_SUCCEED)
{
   LOG_ERROR("JagDescribeCol(2) failed.");
   return CS_FAIL;
}

/*A columm of type string*/

jsret = JagDescribeCol(2, JAG_CS_TYPE, "StringColumn", CS_CHAR_TYPE,
           MAX_STRCOL, 0, 0, CS_CANBENULL);
if (jsret != JAG_SUCCEED)
{
   LOG_ERROR("JagDescribeCol(2) failed.");
   return CS_FAIL;
}


jsret = JagBindCol(1, JAG_CS_TYPE , CS_INT_TYPE , &intcol ,
            intcol_len, &intcol_len,  &intcol_ind );
if (jsret != JAG_SUCCEED)
{
   LOG_ERROR("JagBindCol(2) failed.");
   return CS_FAIL ;
}

jsret = JagBindCol(2, JAG_CS_TYPE , CS_CHAR_TYPE , strcol ,
              MAX_STRCOL, &strcol_ind , (CS_SMALLINT*)NULL);
if (jsret != JAG_SUCCEED)
{
   LOG_ERROR("JagBindCol(2) failed.");
   return CS_FAIL ;
}

/*
```

```
  ** Step 4: send the rows.
  */

  /*
  ** Values to send
  */

  for (int jj = 0 ; jj < MAXDATA ; jj++)
  {
    intcol = jj + 1  ;
    strcpy((char*)strcol, data[jj]  );
    strcol_ind = strlen((char*)strcol) ;
    jsret = JagSendData();
    if (jsret != JAG_SUCCEED)
    {
       LOG_ERROR("JagSendData(1) failed.");
       return CS_FAIL;
    }
  }

  /*
  ** Step 5: Call JagEndResults() to say that we're done.
  */
  jsret = JagEndResults( MAXDATA );
  if (jsret != JAG_SUCCEED)
  {
      LOG_ERROR("JagEndResults() failed.");
      return CS_FAIL;
  }

  return CS_SUCCEED;
}
```

CHAPTER 26

Using Connection Management

This chapter provides an overview of EAServer's built-in connection management features.

Overview of connection management

Connection Manager controls caches of connections that EAServer components use to interact with third-tier servers. Connection management allows EAServer to service hundreds of clients using only a few third-tier database server connections.

Connection Manager provides Java classes, PowerBuilder objects, and C routines for obtaining and releasing connections, as well as utility routines for monitoring cache use.

When to use Connection Manager

Components that use EAServer transactions must use cached connections to interact with remote databases. Otherwise, work done on the connection is not affected by the outcome of the transaction.

Components that use cached connections can realize improved performance and scalability for the following reasons:

- Improved performance – Connection Manager allows client sessions to share previously opened third-tier 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 third-tier connections, less memory and other resources are required to maintain third-tier 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. Do not let your components 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.

Connection caches and security

Your application may have a potential security hole if Java component implementation classes are deployed under EAServer's *html* directory. An unauthorized user can implement a program that connects to EAServer's HTTP port and downloads the component's implementation classes. The user can then decompile the classes and gain access to potentially sensitive information such as database passwords. To close this security hole, Sybase recommends one of the following approaches:

- Deploy Java component implementation classes under the EAServer *java/classes* subdirectory.
- Code components that retrieve connection caches to use the getCacheByName API rather than the APIs that require a database password.
- Implement your Java components to retrieve potentially sensitive information from a properties file that is not located beneath the EAServer *html* directory.

Defining connection caches

A connection cache is an internal EAServer structure that maintains a pool of available connections to a third-tier server. All connections in the cache must share a common user name and password, all must connect to the same third-tier server, and all must use the same connectivity library.

Use EAServer Manager to define connection caches used by your application, as described in Chapter 4, "Database Access," in the *EAServer System Administration Guide*.

JDBC DataSource lookup

EAServer supports JNDI lookup of JDBC 2.0 DataSources to access ConnectionPoolDataSources and XADataSources, as illustrated in the following example. Only EJB components, Web applications, and application clients can use this feature, and you must define a resource reference to alias the connection cache to a JNDI name. For more information, see:

- "Configuring resource references" on page 133 – for EJB components
- "Resource references" on page 385 – for Web applications
- "Resource references" on page 177 – for application clients

JNDI access to connection caches requires JDBC 2.0 drivers Only connection caches that use a JDBC 2.0 driver can be aliased to JNDI resources. Specifically, the driver must implement the javax.sql.DataSource interface.

The JNDI lookup returns a DataSource interface, regardless of the cache configuration.

```
_cntxtProps = New Properties();
_cntxtProps.put(Context.INITIAL_CONTEXT_FACTORY,
               "com.sybase.ejb.InitialContextFactory");
_cntxtProps.put(javax.naming.Context.PROVIDER_URL,
               "iiop://<host_name>:<port>");

nameContext = new InitalContext(_cntxtProps);

_ds =
(javax.sql.DataSource) nameContext.lookup("java:comp/env/jdbc/myAlias2DB");

_conn = ds.getConnection();
```

```
// use the connection

_conn.close();
```

Application authentication

EAServer provides application authentication by allowing you to get a JDBC connection for a user name and password that you specify in the source code. This feature is supported for JDBC 2.0 ConnectionPoolDataSources only. This example gets a connection:

```
_ds = (javax.sql.DataSource)
   nameCntxt.lookup("java:comp/env/jdbc/myAlias2DB");

_conn = ds.getConnection(user_name, password);

// use the connection

_conn.close();
```

An application authenticated connection acts as a shared connection. Since only a single connection can be enlisted in a transaction, you cannot get two application authenticated connections, with different user name/password combinations in the same transaction. Attempts to do so can lead to unexpected results.

Using Java Connection Manager classes

Java components can use the Java Connection Manager (JCM) classes to take advantage of connection caching. The JCM classes manage JDBC connections.

Classes

The JCM classes are:

- com.sybase.jaguar.jcm.JCMCache, which represents a configured connection cache and provides methods to manage connections in the cache.

- com.sybase.jaguar.jcm.JCM, which provides access to JDBC connection caches defined in EAServer Manager. JCM is a factory for JCMCache instances.

These classes are documented in Chapter 1, "Java Classes and Interfaces," in the *EAServer API Reference*.

Java Connection Manager example

The example below implements a Java component that calls these JCM class methods:

- JCM.getCache(String, String, String) – called in the constructor to obtain a cache reference.
- JCMCache.getConnection(0) – called in the method passthru_query(String) to obtain a connection.
- JCMCache.releaseConnection(Connection) – called by passthru_query(String) to release the connection before the method returns.

Many JDBC programs do not explicitly clean up java.sql.Statement objects. Instead, they rely on the JDBC driver to clean up Statement objects when the connection is closed. This strategy does not work with cached connections; you must explicitly clean up Statement objects before releasing a connection back into the cache. To clean up Statement objects, call Statement.close() and set the Statement reference to null.

Warning! To prevent memory leaks, you must explicitly clean up a connection's Statement objects before releasing the connection back into the cache. Do not release the same connection more than once.

This code also calls JContext.forwardResultSet(ResultSet) to forward result sets from a remote server to the client:

```
import com.sybase.jaguar.sql.*;
import com.sybase.jaguar.server.*;
import com.sybase.jaguar.jcm.*;
import com.sybase.jaguar.util.*;

import java.io.*;
import java.sql.*;
import java.util.*;
```

```
/**
 *  Java class to implement rs_passthru EAServer component.
 */
class rs_passthruImpl {

  JCMCache _cache = null;
  private static final String _user = "dba";
  private static final String _password = "sql";
  private static final String _server_url =
                       "jdbc:odbc:Jaguar SVU Sample";

  /**
   * Default constructor that is called by EAServer
   *  when a component instance is created.
   */
  public rs_passthruImpl() throws JException {

    // Get a JDBC connection cache handle.
    try {
      _cache = JCM.getCache(_user, _password,
                _server_url);
    } catch (Exception e) {
      Jaguar.writeLog(true, "rs_passthru(): getCache() exception"
                      + e.getMessage());
      _cache = null;
    }
    // If we can't get a cache handle here, log an
    // error message then throw an exception.
    if (_cache == null)
    {
      Jaguar.writeLog(true,
        "rs_passthru(): Could not access connection cache.");
      Jaguar.writeLog(false, "rs_passthru(): Cache may not be configured prop
erly in Jaguar Manager.");
      throw new JException(
        "rstest(): Could not create connection cache.");
    }
  } // rs_passthruImpl()
  /**
   * Forward the client's query to the remote server, forward
   * the results back to the client.
   */
  public void passthru_query (String query)
     throws JException, SQLException
  {
    Connection conn = null;
```

```
    Statement stmt = null;
    ResultSet rs = null;
    // Note that this code does not catch exceptions;
    // if an exception is thrown will be caught by EAServer
    // and the method invocation will fail.

    // Call getConnection() to get a connection from the cache.
    while (conn == null)
    {
      conn = _cache.getConnection(0);
    }
    // Create a Statement instance and use it to
    // forward the query.
    stmt = conn.createStatement();
    boolean results = stmt.execute(query);
    int update_count = -1;
    // Process all the results, forwarding each result set
    // to the client.
    do {
      if (results)
      {
        rs = stmt.getResultSet();
        if (rs != null)
          JContext.forwardResultSet(rs);
      }
      else
      {
        update_count = stmt.getUpdateCount();
      }
      results = stmt.getMoreResults();
    } while (results || (update_count != -1));
    //
    // Explicitly release the Statement object.
    // Otherwise, it will linger attached to the cached
    // connection.
    //
    stmt.close()
    stmt = NULL;
    _cache.releaseConnection(conn);

  } // passthru_query (String)

} // class rs_passthruImpl
```

For more Java connection management examples, see the source for the EAServer sample Java components in your installation directory.

Using Connection Manager routines in C, C++, and ActiveX components

ActiveX, C, and C++ components can call the Connection Manager routines to take advantage of connection caching. These routines manage caches of ODBC, Client-Library, or Oracle Call Interface (OCI) connections.

EAServer C routines are documented in Chapter 5, "C Routines Reference," in the *EAServer API Reference*. The Connection Manager routines have names that begin with JagCm.

ODBC connection caches

Header files

The header file *jagpublic.h* declares the Connection Manager routines and data structures; the file is located in the *include* subdirectory of your EAServer installation.

Include required ODBC header files before including *jagpublic.h*, for example:

```
#include <sql.h>
#include <sqlext.h>
#include <jagpublic.h>
```

Data structures

Most Connection Manager routines require the address of a CM_CACHE handle as a parameter. The cache handle allows your code to refer to a specific cache that is defined in EAServer Manager. The routines JagCmGetCachebyName or JagCmGetCachebyUser retrieve cache handles.

ODBC uses a HDBC structure to represent a database connection. The JagCmGetConnection routine returns the address of an HDBC structure.

ODBC example

The following example demonstrates program logic that offers improved performance when a matching cache is available and that still functions when no matching cache has been configured. The example first calls JagCmGetCachebyUser to obtain a cache handle for a cache that has matching values for the user name (“myrtle”), password (“secret”), and server name (“tsingtao”) and that uses ODBC. If such a cache exists, the call sets the *cache* variable to the cache handle.

The example then calls JagCmGetConnection, passing the *cache* value as set by JagCmGetCachebyUser, and passing explicit values for the user name, server name, password, and connectivity library. If the *cache* variable contains a valid cache reference, JagCmGetConnection looks directly in the cache for an available connection. If *cache* was set to NULL or the indicated cache has no available connections, JagCmGetConnection creates and opens a new, uncached connection.

Code that follows the implementation strategy illustrated here can achieve better performance when there are many configured caches. Passing the cache handle explicitly in JagCmGetConnection eliminates repeated internal table searches.

```
/* ODBC includes */
#include <sql.h>
#include <sqlext.h>
/* Connection Manager includes */
#include <jagpublic.h>

SQLRETURN ret;        /* Return code catcher    */
SQLHDBC *hdbc;        /* ODBC connection handle */
JagCmCache cache;     /* Cache handle           */

/*
** Retrieve a cache handle if a matching cache is
configured.
** If not, our cache variable will be set to NULL.
*/
cache = NULL;
ret = JagCmGetCachebyUser (“myrtle”, “secret”,
   “tsingtao”, “ODBC”, &cache);
/*
** Ignore the return code. If the call failed, cache
will be
** NULL and we can keep going.
*/
```

```
/*
** Obtain a connection. If we have a cache handle, the
connection
** will be taken from the cache (if one is available).
Otherwise,
** the call creates a new connection.
*/
ret = JagCmGetConnection (&cache, "myrtle", "secret",
   "tsingtao", "ODBC",(SQLPOINTER *)&hdbc,
   JAG_CM_FORCE);

if (ret != SQL_SUCCESS)
{
   ... log the error ...
}

... code that uses the connection goes here ...

ret = JagCmReleaseConnection (&cache, "myrtle",
   "secret","tsingtao", "ODBC",
   hdbc, JAG_CM_UNUSED);

if (ret != SQL_SUCCESS)
{
   ... log the error ...
}
```

You can call JagCmGetCachebyName rather than JagCmGetCachebyUser. For an example, see the reference page for JagCmGetCachebyName in Chapter 5 of the *EAServer API Reference*.

Single-threading ODBC calls on UNIX

On UNIX platforms, ODBC calls must be single-threaded. Connection Manager provides a cache property JAG_CM_MUTEX to be used for this purpose. The JAG_CM_MUTEX property provides access to an Open Server SRV_OBJID mutex structure. The structure should be obtained with JagCmCacheProps(JAG_CM_MUTEX) and locked with the Server-Library srv_lockmutex routine before performing any of the following calls:

- JagCmGetConnection and JagCmReleaseConnection calls on ODBC caches.
- All ODBC calls.

The lock should be released as soon as the operation is complete.

The sample C components contain code that demonstrates how to single-thread ODBC calls.

This requirement should be temporary. A solution that eliminates the need for single-threading is planned for a future release.

Client-Library connection caches

To support Client-Library connection caches, EAServer includes a native threaded version of Open Client Client-Library using the shared library *libjct_r.sl*. This version supports all features in Open Client 11.1, plus the high availability and failover and wide table features from Open Client 12.5 (varchar/varbinary columns more than 255 bytes long and tables with more than 255 columns). You can use these Opent Client 12.5 features only when connected to Adaptive Server® Enterprise version 12.5 or later.

Header files

Before including *jagpublic.h*, you must include the Client-Library *ctpublic.c* header file, as in the example below:

```
#include <ctpublic.h>
#include <jagpublic.h>
```

Data structures

Most Connection Manager routines require the address of a CM_CACHE handle as a parameter. The cache handle allows your code to refer to a specific cache that is defined in EAServer Manager. The routines JagCmGetCachebyName or JagCmGetCachebyUser retrieve cache handles.

Client-Library uses a CS_CONNECTION structure to represent a database connection. The JagCmGetConnection routine returns the address of a CS_CONNECTION structure.

Client-Library example

The following example calls JagCmGetConnection to obtain a connection that has a user name of “myrtle,” has a password of “secret,” connects to the server “tsingtao,” and uses Client-Library:

```
#include <ctpublic.h>
#include <jagpublic.h>
```

```
CS_RETCODE ret;
CS_CONNECTION *connection;
JagCmCache cache;

/*
** Obtain a connection.
*/
cache = NULL;
ret = JagCmGetConnection (&cache, "myrtle", "secret",
   "tsingtao", "CTLIB_110",(SQLPOINTER *)&connection,
   JAG_CM_FORCE);

if (ret != CS_SUCCEED)
{
   ... log the error ...
}

... code that uses the connection goes here ...

ret = JagCmReleaseConnection (&cache,
                          "myrtle","secret","tsingtao",
                          "CTLIB_110",
                          (SQLPOINTER)connection,
                          JAG_CM_UNUSED);

if (ret != CS_SUCCEED)
{
   ... log the error ...
}
```

In the example, the call to JagCmGetConnection looks for a cache that includes matching values for the user name ("myrtle"), password ("secret"), and server name ("tsingtao") and that uses Client-Library. The last parameter, *opt*, is passed as JAG_CM_FORCE to indicate that the call should open a new, uncached connection if no cached connection is available. JagCmReleaseConnection releases control of the connection: a connection that was taken from a cache is returned to that cache; an uncached connection is closed and deallocated.

Note that JagCmGetConnection attempts to open a connection even when no matching cache is configured. In this case, JagCmGetConnection attempts to create a new, uncached connection using the specified values.

In this example, JagCmGetConnection and JagCmReleaseConnection return Client-Library return codes since both calls use "CTLIB_110" for the *con_lib* parameter.

You can call JagCmGetCachebyName rather than JagCmGetCachebyUser. To see an example, see the reference page for JagCmGetCachebyName in the *EAServer API Reference*.

Maintaining properties, options, and database context

The connection's server name, user name, and password are fixed when the cache is established. However, other connection properties can be changed dynamically when the connection is opened. For example:

- Server-side connection options – ct_options calls, 'set' language commands, or equivalent ODBC calls all affect the server's response to commands sent on the connection.
- Database context – different users of a cached connection may use different databases. You can avoid problems by explicitly changing the database each time a cached connection is used.
- Connection properties – connection properties affect client-side connection behavior.

Follow these guidelines to avoid problems with inconsistent connection state:

- Set any options and properties that your code requires when you obtain a connection.
- If your code may share a cache with other connections, set changed properties and options back to the original values before releasing the connection.
- If your code is the only user of a cache, and no other components use the named cache, then you do not need to set options and properties back to the original values; however, you will have to reset the properties to the original values when you get the connection again.

Client-Library error and message callbacks

EAServer installs default server message and client message callbacks into cached Client-Library connections. The default callbacks write error and message information to the server's log file.

When using Client-Library connections, you can install your own server message and client message callbacks into connections retrieved from JagCmGetConnection. JagCmReleaseConnection reinstalls the default callbacks before placing connections back into the cache.

Oracle connection caches

You can define caches of connections to an Oracle database using OCI 7.x, OCI 8.x, or OCI 9.x.

Oracle autocommit setting

EAServer creates Oracle connections with the default autocommit setting, autocommit off. In non-transactional components, you must explicitly issue a commit command to commit update and insert queries. In transactional components, the EAServer transaction manager issues commit and rollback commands for connections used by the components that participate in an EAServer transaction.

Note In a non-transactional component, if you do not explicitly issue commit or rollback after sending Oracle commands, the commands may be committed when a transactional component uses the same connection. EAServer issues a commit to clear the connection status before passing Oracle connections to a transactional component.

Using OCI 7.x connection caches

Header files

Include Oracle header files after *jagpublic.h*, as in the example below:

```
#include <jagpublic.h>
#include "oratypes.h"
#include "ocidfn.h"
#ifdef __STDC__
    #include "ociapr.h"
#else
    #include "ocikpr.h"
#endif
#include "ocidem.h"
```

Data structures

Most Connection Manager routines require the address of a CM_CACHE handle as a parameter. The cache handle allows your code to refer to a specific cache that is defined in EAServer Manager. The routines JagCmGetCachebyName or JagCmGetCachebyUser retrieve cache handles.

OCI 7.x uses an Lda_Def structure to represent a database connection. The JagCmGetConnection routine returns the address of an Lda_Def structure.

OCI 7.x example

The example below retrieves an Lda_Def structure, executes a statement using the connection, then returns the connection to the cache.

```
#include <jagpublic.h>
#include "oratypes.h"
#include "ocidfn.h"
#ifdef __STDC__
    #include "ociapr.h"
#else
    #include "ocikpr.h"
#endif
#include "ocidem.h"

Cda_Def cda;
Lda_Def *lda;

#define   USERID            "system"
#define   PASSWD          "manager"
#define   DATASOURCE "OCITEST"

    /* Connect to ORACLE. */
    cache = NULL;
    ret = JagCmGetConnection(&cache,
                             USERID, PASSWD, DATASOURCE,
                             "OCI_7",
                             (void*)&lda,
                              JAG_CM_FORCE);

    /* Open a cursor, parse stmt, execute, close cursor */
    oopen(&cda, lda, (text *) 0, -1, -1, (text *) 0, -1);
    oparse(&cda, sql_statement, -1, FALSE, 2);
    ...

    if (oexec(&cda))
        oci_error(&cda);
    ...

    if (oclose(&cda))
        oci_error(&cda);

    /* release connection */
    ret = JagCmReleaseConnection(&cache,
```

```
                                    USERID, PASSWD, DATASOURCE,
                                    "OCI_7",
                                    (Lda_Def *)lda,
                                     JAG_CM_UNUSED);
```

Using OCI 8.x connection caches

Header files

Include *oci.h* before *jagpublic.h*, as in the example below:

```
#include <oci.h>
#include <jagpublic.h>
```

Data structures

Most Connection Manager routines require the address of a CM_CACHE handle as a parameter. The cache handle allows your code to refer to a specific cache that is defined in EAServer Manager. The routines JagCmGetCachebyName or JagCmGetCachebyUser retrieve cache handles.

OCI 8.x uses a OCISvcCtx structure to represent a database connection. The JagCmGetConnection routine returns the address of a OCISvcCtx structure.

OCI 8.x example

The example below retrieves an OCI 8.x connection, executes a statement using the connection, then returns the connection to the cache.

```
#include <jagpublic.h>
#include <oci.h>

#define   USERID            "system"
#define   PASSWD          "manager"
#define   DATASOURCE "OCITEST"

 JagCmCache  cache;
 OCIEnv *envhp;
 OCISvcCtx **svcpp, *svchp;
 OCIError *errhp;
 OCIStmt *stmthp;
 sword ociret;

     /* Connect to ORACLE. */
    cache = NULL;
    ociret = JagCmGetConnection(&cache,
                                USERID, PASSWD, DATASOURCE,
```

```
                                "OCI_8",
                                (void*)&svchp,
                                JAG_CM_FORCE);
...
  /* Initialize an Env, to allocate stmt and error handles */
OCIEnvInit( &envhp, OCI_DEFAULT, (size_t) 0, (dvoid **)0 );
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp,
                OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));
checkerr(errhp, OCIStmtPrepare(stmthp, errhp, sql_statement,
                               (ub4) strlen((char *) sql_statement),
                               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

  /* execute using the service context */
 checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                (CONST OCISnapshot *) NULL,
                                (OCISnapshot *) NULL, OCI_DEFAULT));
.....

  /* free handles */
 OCIHandleFree(stmthp, OCI_HTYPE_STMT);
 OCIHandleFree(errhp, OCI_HTYPE_ERROR);

  /* release connection */
  ret = JagCmReleaseConnection(&cache,
                               USERID, PASSWD, DATASOURCE,
                               "OCI_8",
                               svchp,
                                JAG_CM_UNUSED);
```

Using OCI 9.x connection caches

Using an OCI 9.x connection cache requires the same header files and data structures, as described in “Using OCI 8.x connection caches” on page 498.

OCI 9.x example

This code sample illustrates the changes you need to make to the “OCI 8.x example” on page 498 for an OCI 9.x connection. The only differences are the third arguments that you pass to the JagCmGetConnection and JagCmReleaseConnection methods. Other than these changes, the OCI 8.x example works for OCI 9.x connections.

```
  /* Connect to ORACLE */
 cache = NULL;
```

```
ociret = JagCmGetConnection(&cache,
                            USERID, PASSWD, DATASOURCE,
                            "OCI_9",
                            (void*)&svchp,
                            JAG_CM_FORCE);

 /* Release connection */
 ret = JagCmReleaseConnection(&cache,
                              USERID, PASSWD, DATASOURCE,
                              "OCI_9",
                              svchp,
                               JAG_CM_UNUSED);
```

Using cached connections in PowerBuilder components

This section briefly summarizes how to create PowerBuilder components that interact with the EAServer Connection Manager. For detailed instructions on using cached connections in PowerBuilder, see the *Application Techniques* manual in the PowerBuilder documentation.

To make sure that your PowerBuilder components use a cached connection, first use EAServer Manager to verify that a matching cache is defined.

In your component, you can obtain cached database connections using standard PowerBuilder techniques for opening a connection. When you open a connection, EAServer's PowerBuilder dispatcher checks whether a cache exists with matching values for user name, password, and connectivity library. If a matching cache exists, your component receives a connection from the cache. Likewise, when you use standard PowerBuilder techniques to close the connection, EAServer places it back in the cache for reuse.

For finer control over the use of connection caches, you can set the following DBParm settings before opening a connection:

- Set the CacheName DBParm if you wish to identify a cache by name. This setting causes EAServer to retrieve connections from the cache with that name rather than looking for matching values for user name, password, and connectivity library. You cannot use this option if the "Enable cache by Name Access" option is not set for the cache in EAServer Manager.
- Set the GetConnectionOption DBParm to control what happens if all connections in the cache are in use.

- Set the ReleaseConnectionOption to control whether the released connection is closed and deallocated or placed back in the cache for reuse.

For information on these options, see the DBParm documentation in the PowerBuilder online help.

Connection Manager guidelines

This section explains Connection Manager guidelines.

Avoiding results-pending errors

You must be careful not to release a connection that has unprocessed command results associated with it. Any time you send a command using a cached connection, you must completely process the results of the command before releasing the connection for reuse. Failure to process all results will cause errors in the next component that uses the connection.

Connections and cache handles

Never release a connection into a cache other than the one in which it was created. If you follow the coding conventions illustrated in the examples, this issue should not be a problem.

Do not release a connection twice—this can cause unexpected problems.

CHAPTER 27

Creating Entity Components

An entity component is an EJB entity bean or a component of another type that implements the CtsComponents::ObjectControl interface, implemented to represent a database row. EAServer supports the standard EJB entity bean model, and provides a similar model for components of other types.

Implementing entity components

An entity component is an EJB entity bean or a component of another type that implements the CtsComponents::ObjectControl interface. 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 can be EJB entity Beans implemented according to the EJB 2.0, 1.1, or 1.0 standard (see Chapter 7, “Creating Enterprise JavaBeans Components”). You can also implement CORBA entity components by following these requirements:

- Use CtsComponents::ObjectControl as the component's control interface. See "Configuring a control interface" on page 72.
- Define a primary key type for the component. See "Defining the primary key type" on page 126 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" on page 127 for more information.

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

- **Manual persistence** 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 (BMP) model defined by the EJB 2.0 and 1.1 specifications, but has been extended to support other component types.
- **Automatic persistence** EAServer generates skeleton code to manage the storage and retrieval of persistent data. This model can be used by entity components that are not EJBs.
- **Generated class (EJB CMP)** For EJB CMP entity beans, EAServer generates a class that manages the interaction with the remote database to load and store the values of container-managed fields.

Coding to support manual persistence

To use component-managed persistence, you must configure the component's persistence properties and implement the required methods from the EntityBean or CtsComponents::ObjectControl interfaces. Display the Component Properties window in EAServer Manager and configure the following fields on the Persistence tab:

- **Persistence** Choose Component Class.
- **Primary Key** Enter the name of the primary key type (see "Defining the primary key type" on page 126).

In most cases, no other persistence settings are required. You can delegate to EAServer'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 545). Delegation requires that you use the CtsComponents::DataStream and CtsComponents::Storage interfaces. See the generated Interface Repository documentation in your EAServer installation (in the *html/ir* subdirectory) for descriptions of these interfaces.

Improve performance: identify read-only methods
For best performance when using component-managed persistence, mark all remote interface methods that do not modify data as read-only. To do so, select the Read Only check box in the Method Properties dialog box. When this property is enabled, the components ejbStore or ctsStore method is not invoked after the business method returns.

Understanding the automatic persistence architecture

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

- **Using mapped fields** In the mapped field model, you define a mapping from a database table to fields in your component implementation class. When a write to the database is required, the server reads the field values; after reading new data from the database, the server assigns new field values for each mapped database column. This model corresponds to the container-managed Persistence (CMP) model required by the EJB 2.0 and 1.1 specifications, but has been extended to support other component types.
- **Using binary storage** In this model, you define state-accessor methods and an IDL state type. The 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`.

The automatic persistence architecture includes:

- As for any component, the component's *implementation class* and *skeleton*. The skeleton acts as the interface between EAServer and the implementation class.

  For EJB 2.0 entity beans, your implementation class must be an abstract Java class, with abstract accessor methods for the bean's container-managed fields and abstract declarations of the bean's ejbSelect methods. For example, if *firstName* is a container-managed field of type String, you must declare these abstract accessor methods:

  ```
  public abstract String getFirstName();
  public abstract void setFirstName(String value);
  ```

  EAServer generates the implementation class that executes at runtime. This class extends your abstract EntityBean class and implements the accessor methods for container-managed fields.

- A *storage component*, which stores instance field data to a remote database. The storage component manages all data storage and retrieval, including concurrency control to prevent overlapping updates of the same rows.

- The component's *state datatype*, which is an IDL structure that contains the data that is to be stored in the database. The state datatype is required to exchange data between the component's skeleton and the storage component.

- The component's *state accessor methods*, which the component or skeleton implements to interact with the storage component. When the client calls a business method, the instance fields must be loaded with up-to-date data, so the storage component calls the *set* method to provide the data. When the business method completes, the storage component calls the *get* method to obtain and save the changed data.

  For EJB entity beans, EAServer generates a skeleton with state accessor methods. For entity components of other types, you must implement accessor methods and specify their names in component properties as described below.

- The *object cache*, which allows in-memory caching of entity instance data to avoid unnecessary database reads to load instance state.
- The *query cache*, which allows in-memory caching of finder and ejbSelect query results to avoid unnecessary database reads to execute finder and ejbSelect methods.
- The component's *abstract persistence schema*, which defines the names and types of container-managed fields, and (for EJB 2.0 entity beans) container-managed relationships between entity beans. In EAServer, the abstract persistence schema is configured by the component properties, specifically:
  - The primary key and state datatypes.
  - *Field-mapping properties*, to bind component fields to database columns.
  - *Query-mapping properties*, to specify the queries required to run finder methods and ejbSelect methods. For EJB 2.0 entity beans, EAServer supports query mappings defined in standard EJB Query Language (EJB-QL).
  - *Table-mapping properties*, to further fine tune the database access. For example, table-mapping properties can be defined to allow the use of stored procedures for all database access.
  - *Relationship components*, which manage EJB 2.0 entity bean relationships to allow one bean to contain instances of another in a container-managed field. Relationship components are themselves EJB 2.0 entity beans generated entirely by EAServer.

  When you import an EJB CMP entity bean from an EJB-JAR file, the importer configures almost all EAServer Manager properties based on the abstract persistence schema defined in the deployment descriptor.

Configuring automatic or EJB CMP persistence

If you are developing EJB entity beans, use an EJB development tool to create an EJB-JAR file that defines the components' CMP fields and container-managed relationships. You can define CMP entity beans in EAServer Manager, but it may be easier with a dedicated EJB development tool.

EAServer includes a sample EJB 2.0 CMP entity bean, in the installation subdirectory *html/classes/Sample/cmp20sample*. See the *readme.txt* file in this directory for instructions on deploying and running the sample.

When you have CMP entity beans defined in an EJB-JAR file, import the EJB-JAR file into EAServer as described in Chapter 9, "Importing and Exporting Application Components," in the *EAServer System Administration Guide*. The JAR importer configures appropriate defaults for almost all settings. After importing, review the persistence settings described below and verify that they are correct.

To use automatic persistence for non-EJB entity components, you must configure the persistence properties in EAServer Manager.

Component properties for automatic persistence are configured on the Persistence tab in the Component Properties dialog box. This tab has subtabs to display the categorized properties.

❖ **Configuring automatic persistence**

1 Specifying the CMP version for EJB 2.0 entity beans.

2 Setting Persistence/General subtab properties.

3 Enabling automatic key generation if your entity component uses a table with an automatically generated key or your EJB entity bean uses java.lang.Object as the primary key class.

4 Creating database tables.

5 Configuring concurrency control.

6 Setting field-mapping properties.

7 Specifying finder- and ejbSelect-method queries.

8 Configuring table-mapping properties.

9 Using relationship components if using EJB 2.0 container-managed relationships.

After you have verified the deployment is working, you can optimize your component's performance by configuring in-memory caching of instance data and query results. For more information, see "Entity instance and query caching" in the *EAServer Performance and Tuning Guide*.

Specifying the CMP version for EJB 2.0 entity beans

For EJB 2.0 entity beans, choose a CMP version on the General tab in the Component Properties dialog box. Choose 2.0 to match the EJB 2.0 specified behavior. Choose 1.1 to match the EJB 1.1 persistence model. Use 2.0 CMP for new development, and 1.1 for existing beans that have code that requires the 1.1 model.

For EJB 1.1 entity beans, you can not set the CMP version.

Setting Persistence/General subtab properties

Display the Component Properties window in EAServer Manager and configure the following fields on the Persistence/General subtab:

- **Persistence** For an EJB CMP entity bean, choose Generated Class. For components of other types, choose Automatic Persistent State.
- **Generated Class Name** For an EJB CMP entity bean, optionally enter a class name for the generated subclass. If you do not specify a class name, the default is:

  ```
  java-package._ps_package_component
  ```

  Where *java-package* is the Java package of the implementation class, *package* is the EAServer package name, and *component* is the component name.
- **Primary Key** Enter the name of the primary key type (see “Defining the primary key type” on page 126). If you have imported an EJB entity bean, the primary key has been defined already.

  An EJB 2.0 entity bean that uses CMP and specifies java.lang.Object as the primary key class requires automatic key generation when deployed to EAServer. Components of other types can use this feature as well. See “Enabling automatic key generation” on page 513.
- **State** If you have imported an EJB CMP entity bean from an EJB-JAR file, the state has been defined already. Otherwise, define the state type as described in “Defining the state datatype” on page 511.

- **State Methods** If you are creating an EJB entity bean, enter "default". If you are creating an entity component of another type, specify the names of the state accessor methods implemented by your component. See "Defining the state methods" on page 512.
- **Storage Component** Specify the name of the storage component, as described in "Storage components" on page 545.
- **Connection Cache** Enter the name of a JDBC connection cache that connects to the database. The default for new components is `JavaCache`. For components that are imported from an EJB-JAR file, the default is the value of the com.sybase.jaguar.server.defaultStorageCache server property. The cache must:
  - Allow by-name access (configured on the Cache tab in the Connection Cache Properties dialog box).
  - Specify a database type. This property defines database-specific information required by the storage component, for example, the commands to verify a table exists and create new tables. Several predefined configurations are provided for popular databases, and you can create your own. For more information, see "Configuring connection caches" in the *EAServer System Administration Guide*.
  - Have a user name specified for the property com.sybase.jaguar.conncache.ssa.systemid if Set-Proxy support is enabled (com.sybase.jaguar.conncache.ssa is set to true). In an EJB CMP entity bean, the client user name is not available to set proxy to since the persistence engine runs as the system user.

  For EJB CMP entity beans, EAServer supplys JDBC wrapper drivers that can improve performance through the use of deferred updates and stored procedures. For more information, see "Using CMP JDBC wrapper drivers" in the *EAServer Performance and Tuning Guide*.
- **Table** If using mapped table fields, enter:

  `map:`*`table`*

  Where *table* is the database table name. "Creating database tables" on page 516 describes how tables are created and made accessible to the storage component.
- **Timestamp** Optionally specify a database timestamp column that EAServer uses for concurrency control. Using timestamps for concurrency control yields the best performance in most cases. See "Configuring concurrency control" on page 517 for more information.

- **Create Database Triggers** Applies if you have configured instance or query caching as described in "Entity instance and query caching" in the *EAServer Performance and Tuning Guide*, and you have enabled database change notification as described in that section. This option enables automatic creation of database triggers to notify the EAServer cache manager when the table data changes.
- **Select With Lock** Configures data locking to enable pessimistic concurrency control. See "Configuring concurrency control" on page 517 for more information.

Warning! Carefully evaluate your concurrency control model before deploying your application. Concurrency control greatly affects performance as well as the integrity of your back-end database. See "Configuring concurrency control" on page 517 for more information.

Defining the state datatype

For automatic persistence, you must define a *state datatype* to be used for exchange of data between the component's skeleton and the storage component.

State types for mapped fields If using the mapped-fields storage model, the state datatype must be an IDL structure. Enter the structure name in the in the State field on the Persistence/General subtab in the Component Properties Dialog box. For example:

```
MyPackage::CustomerState
```

You can enter the name of an IDL structure that does not exist; EAServer Manager creates it when you click Ok in the Component Properties dialog box. Afterwards, navigate to the IDL definition and edit the structure as described in "Editing IDL types, exceptions, and interfaces" on page 88.

Define the structure field names and types as follows:

- For EJB 2.0 entity beans, specify one field for each container-managed field that is not part of the primary key and not a container-managed relationship field. Use the same name as the container-managed field. Choose the IDL type that matches the Java datatype, as listed in Table 29-2 on page 547.
- For EJB 1.1 entity beans, specify one field for each container-managed field that is not part of the primary key. Use the same name as the container-managed field. Choose the IDL type that matches the Java datatype, as listed in Table 29-2 on page 547.

- For non-EJB entity components, specify one field for each component field that is to be stored in the database table, excluding fields that are part of the primary key. Choose the IDL type that matches the field's datatype in the implementation class, as listed in Table 29-2 on page 547. If using timestamps for concurrency control, do not include an IDL field for the timestamp column. Your implementation class must contain state accessor methods to apply the field values to implementation fields, and populate the state type from instance field data. Specify the state accessor method names on the State Methods field on the Persistence/General subtab, as described in "Defining the state methods" on page 512.

State types for binary storage If you are using the binary storage model, enter the name of an IDL structure or serializable Java class. Your state accessor methods must contain code to get and set this data to and from the current instance, as described in "Defining the state methods" on page 512.

Defining the state methods

For automatic persistence in non-EJB components or in EJB components using the binary-storage persistence model, your component implementation must contain state accessor methods to read state data from the current instance and apply state data to the current instance. Specify the names of these methods on the State Methods field in the Persistence/General subtab. If you specify no value, the default is `getState,setState`.

Your component implementation must contain these methods, but they should not be listed in the component's client interfaces. The getState method returns an instance of the type specified by the State field, and the setState method accepts a parameter of this type. For example, if the State type is ShoppingCartState, the getState and setState methods might be defined as follows in Java:

```
private ShoppingCartState data;

ShoppingCartState getState()
{
    return data;
}

void setState(ShoppingCartState state)
{
    data = state;
}
```

Enabling automatic key generation

If automatic key generation is enabled, keys are created automatically for every row inserted in the table. If you are mapping container-managed fields to multiple tables, automatic key generation applies only to the main table, specified on the Persistence/General subtab.

❖ **Specifying the key type for EJB CMP entity beans**

There are two options for the primary key type when using automatic key generation in EJB and Java components:

- **java.lang.Object** The EJB 2.0 specification requires this type for entity beans that have automatically generated keys. However, using java.lang.Object makes client coding difficult, particularly if the home interface has finder methods that take key values as input. In this case, you do not know what the actual Java key type is until after deploying the component.
- **java.lang.Integer or other integer types** EAServer allows you to use an integer type with automatic key generation configured. You can also use other integer types, as long as you specify the wrapper class name, such as java.lang.Long.

Specify the key type in the EAServer Manager Component Properties dialog box as follows:

1 On the Persistence/General subtab, set the Primary Key field to the Java class name used in your code. EAServer Manager saves the setting as the IDL type, and you will see the IDL type after you close and reopen the Component Properties dialog box.

Java key type	IDL key type
java.lang.Object	XDT::Integer
java.lang.Integer	CtsComponents::LongValue
Other integral types	The corresponding CtsComponents value structure

2 Regenerate stubs and skeletons for the component if you have changed the primary key type.

❖ **Choosing the key generation mechanism**

- EAServer supports three mechanisms for key generation:

- **Using the Sybase identity datatype** If you are using Sybase Adaptive Server Enterprise or Adaptive Server Anywhere, the main table uses the identity datatype for the primary key. The database manages the creation of new keys. To configure this mechanism, follow the procedure "Configuring key generation to use the Sybase identity datatype" on page 514. Alternatively, use a key table as described in "Overriding the use of native identity types" on page 516.
- **Using the Oracle sequence datatype** If you are using an Oracle database, the main table uses an Oracle sequence for the primary key. The database manages the creation of new keys. To configure this mechanism, follow the procedure "Configuring key generation using the Oracle sequence datatype" on page 514. Alternatively, use a key table as described in "Overriding the use of native identity types" on page 516.
- **Using a key lookup table** For any SQL database, you can use a single-row, single-integer-column table to generate key values. EAServer increments the key lookup value to generate new keys. If other processes or applications insert to the table, they must also use the key lookup table. To configure this mechanism, follow the procedure "Configuring key generation to use a key lookup table" on page 515.

❖ **Configuring key generation to use the Sybase identity datatype**

1 Use the Advanced tab to set the property com.sybase.jaguar.component.generateKey to `true`. The default is `false`.

2 In the Persistence/Field Mappings tab, verify that the key field is mapped to the Sybase identity type or a compatible type.

3 Use the Advanced tab to verify that the property com.sybase.jaguar.component.db.sequence is not set, or set to no value.

❖ **Configuring key generation using the Oracle sequence datatype**

1 Use the Advanced tab to set the property com.sybase.jaguar.component.generateKey to `true`. The default is `false`.

2 In the Persistence/Field Mappings tab, verify that the key field is mapped to the Oracle sequence type or a compatible type.

3 If using an existing table or a table that you create yourself (rather than relying on autocreation by EAServer), use the Advanced tab to set the component property com.sybase.jaguar.component.db.sequence. This property specifies the name of the Oracle sequence to use. The default is:

```
table_keys
```

Where *table* is the main table name specified on the Persistence/General subtab.

❖ **Configuring key generation to use a key lookup table**

1 Use the Advanced tab to set the property com.sybase.jaguar.component.generateKey to `true`. The default is `false`.

2 In the Persistence/Field Mappings tab, verify that the key field is mapped to an integer type that is compatible with the type used in the lookup table– see "Setting field-mapping properties" on page 521.

3 Use the Advanced tab to set the component property com.sybase.jaguar.component.db.sequence to a value:

```
table.column += key_use_rate
```

or

```
table += key_use_rate
```

Where:

- *table* is the name of the key lookup table. The table must contain a single row with a single integer column. The column datatype cannot have precision greater than 64 bits. The default key lookup table name is the main table name appended with `_keys`; for example, if the main table is phone, the key lookup table is phone_keys.
- *column* is the column name, which must be an integer column or another integral datatype. If you specify only a table name, the table must contain a column named next_key.
- *key_use_rate* is the number of keys that are reserved at once. To prevent different threads from creating duplicate keys, EAServer uses a semaphore to synchronize the key increment operation. Each thread reserves *key_use_rate* key values per increment. The key use rate can be tuned to reduce inter-thread contention for locks on the key table. The default of 100 results in good performance for most applications. Very large values can result in large gaps between key values. Gaps in the key sequence are possible if the key use rate is greater than 1.

4 Create the key lookup table if it does not exist in the database.

❖ **Overriding the use of native identity types**

- If you are using Sybase or Oracle, you can force the use of a key lookup table rather than the Sybase identity or Oracle sequence datatypes. To do so, set the com.sybase.jaguar.component.db.sequence. property and specify a key use rate as described in “Configuring key generation to use a key lookup table” on page 515. If the value of this property contains “+=”, EAServer uses a key lookup table regardless of the database type.

Creating database tables

The entity component’s database table must be specified on the Table Name setting in the Persistence/General subtab.

Tables for binary storage

If you are using binary storage, simply enter the table name. If you have specified a valid database type in the connection cache properties, EAServer creates the table if it does not exist. “Table schema for binary storage” on page 549 describes the required table schema.

Tables for mapped-fields storage

If you are using mapped-field storage, you can use actual database table names, or logical table names that do not necessarily exist in the database. You can map fields to several tables, in which case the table named on the Persistence/General tab becomes the “main” table.

Table access is configured by the component’s table mapping properties. The default mappings require that the tables exist and are accessed by standard SQL commands.

Use logical names if you must use stored procedures for data access, or are using a non-SQL database. You must configure the data access operations for logical tables, as described in “Configuring table-mapping properties” on page 526.

If you have specified a valid database type in the connection cache properties, EAServer creates database tables if they do not exist. Otherwise, you or your database administrator (DBA) must create tables manually.

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

The table column types must agree with the mapped fields (see “Setting field-mapping properties” on page 521), and contain a timestamp column if timestamps are used for concurrency control. “Supported Java, IDL, and JDBC/SQL types” on page 547 lists the supported JDBC/SQL types and the corresponding Java and IDL field types.

Configuring concurrency control

When using mapped fields, you must also choose a concurrency control model. Concurrency control prevents overlapping updates from entity instances running in different threads or different servers, or from applications running outside of EAServer. There are two approaches for concurrency control:

- In the *Pessimistic concurrency control (PCC)* model, data rows are locked when read, for the duration of the EAServer transaction. This method can introduce database deadlocks and usually reduces the scalability of the application.
- In the *Optimistic concurrency control (OCC)* model, data rows are not locked when read. Timestamps are used for concurrency control; the timestamp can be a timestamp column in the database that is updated every time the row is modified, or it can be the row data itself. At the end of the transaction, the in-memory timestamp value is compared to the timestamp value in the database, and the transaction rolls back if the values do not match.

OCC allows greater scalability than PCC, however, when using OCC, client applications must be coded to retry rejected updates, or you must enable automatic transaction retry for the application components as described below.

When using OCC, each update statement contains SQL logic that determines if the last-read timestamp matches the stored value, and rolls back the transaction if the timestamp does not match. In other words, updates based on stale data are rejected. There are several options for using timestamps:

- Use a timestamp column: each table contains a timestamp column, which can be a database timestamp type (if supported) or an integer column that is incremented for every update. This option provides good performance if your database and table schema can support it.

- Use all-values comparison: on update, all row values are compared to the last-read values to detect update collisions. OCC with all-values comparison is the default concurrency control model. Performance with this option is worse than when using a single timestamp column, particularly if the table contains many columns or wide columns (such as Sybase text or image columns). Whenever possible, the use of a timestamp column is recommended in these cases.
- Use a table-level timestamp: the timestamp is a single integer counter that is incremented for every update, insert, or delete in the main table. This option provides the best performance for CMP entity beans that are mapped to read-mostly (or read-only) tables when verified results are required to meet transaction isolation requirements. For best results, use table-level timestamps with a Sybase CMP wrapper driver to allow verification queries to be batched with other deferred operations. For more information, see "Using CMP JDBC wrapper drivers" in the *EAServer Performance and Tuning Guide*.

❖ **Enabling optimistic concurrency control**

1 Configure the Timestamp field on the Persistence/General subtab. Table 27-1 describes the allowable values. For best performance when using tables with many columns or large column values (such as Sybase text or image columns), specify a timestamp column as described in Table 27-1.

If multiple tables are used and you specify a timestamp column, all tables must contain a column with the same name and datatype.

Table 27-1: Timestamp field values

To configure	Set the timestamp value to
A timestamp column	The name of a single column in each table that serves as the timestamp to detect update collisions. If the component uses multiple tables, each must contain a timestamp column with this name. The column type can be: • A 4-byte integer – this is the default timestamp column type. All processes that update the table(s) must increment the timestamp with each update, or your DBA can create an update trigger to increment the timestamp automatically. • The database timestamp type – you can use the timestamp datatype if using Sybase Adaptive Server Enterprise or Adaptive Server Anywhere version 7.0 or later. You must also define a field mapping property to specify the timestamp datatype as described in "Setting field-mapping properties" on page 521. For example, if the column name is ts, specify the mapping as: ts[dbts not null] dbts is a logical type name mapped to the timestamp type in the Sybase_ASE and Sybase_ASA database types. If the database does not support timestamps, a 4-byte integer counter is used instead.

To configure	Set the timestamp value to
A table level timestamp	A table and column name, in the form *ts_table*.*ts_column*, where *ts_table* specifies the timestamp table and *ts_column* specifies the name of the timestamp column in the timestamp table. The specified timestamp table must be separate from the main table. The timestamp tables can contain multiple columns, to allow use of one timestamp table by multiple entity beans. Timestamp tables are automatically created if they do not exist. A timestamp table can be shared among multiple components even when only one column is present in the timestamp table. In other words, a single timestamp value can be shared by multiple tables. This helps further improve performance for a group of read-mostly tables. However, any insert, delete, or update on any of the tables results in all cache entries being discarded. When using a timestamp table, database triggers are required to increment the timestamp for each update, delete, or insert to tables that are mapped to the component or components that require the timestamp. You can set the component property com.sybase.jaguar.component.ts.triggers property so EAServer creates triggers, create triggers yourself, or add code to existing triggers.
All values comparison	Leave blank.

2 Optionally enable auto-retry for the application components so that EAServer replays EJB CMP transactions that fail due to conflicting updates. Auto-retry must be configured for the component that initiates the transaction, which is typically a session bean in EJB applications. Auto-retry works only for intercomponent calls, not for direct invocations of entity beans from the Web tier or base clients. Configure auto-retry as follows:

- In the properties of the components that initiate the transactions to be retried, use the Advanced tab to set the property com.sybase.jaguar.component.tx_retry. A value of true enables auto retry. A value of false disables auto-retry. If this property is not set, the value of the server property com.sybase.jaguar.server.tx_retry is used. If neither the component property or server property is set, the default is false.
- In server properties, use the Advanced tab to set the property com.sybase.jaguar.server.tx_retry. The default of false disables auto-retry for all components for which auto-retry is not explicitly enabled. Specify true to enable auto-retry for components for which auto-retry is not explicitly set to false.

Auto-retry is not appropriate for all applications. For example, an end user may want to cancel a purchase if the item price has risen. If auto-retry is disabled, clients must be coded to retry or abort transactions that fail because of stale data. The exception thrown is CORBA::TRANSIENT (for EJB clients, this exception is the root cause of the java.rmi.RemoteException thrown by the EJB stub).

3 For EJB CMP entity beans, configure an effective transaction isolation level, as described in "Configuring CMP isolation level" in the *EAServer Performance and Tuning Guide*.

4 Verify that the Select With Lock option on the Persistence/General subtab in the Component Properties dialog box is disabled. On the Advanced tab, verify that com.sybase.jaguar.component.selectForUpdate is not set or set to false.

❖ **Enabling pessimistic concurrency control**

1 Configure a locking mechanism. You can do one of the following:

- Enable the Select With Lock option on the Persistence/General subtab. When using jagtool or XML configuration files, set com.sybase.jaguar.component.selectWithLock to true.
- Enable the Select for Update option by setting the com.sybase.jaguar.component.selectForUpdate property to true on the Advanced tab in the Component Properties dialog box or by using jagtool or an XML configuration file. This setting requests an exclusive database lock be obtained at select time to avoid deadlocks during lock promotion. Also consider configuring the database table for row-level locking.
- Configure the table-mapping select queries and add "holdlock" or the appropriate lock syntax for your database. See "Configuring table-mapping properties" on page 526 for more information.

2 Make sure that OCC is disabled by setting the Timestamp field to "none" on the Persistence/General subtab in the Component Properties dialog box.

Setting field-mapping properties

Field-mapping properties specify which table columns correspond to the component's container-managed fields, the primary key (which may map to one or several columns), and the timestamp (if used for concurrency control). Before configuring field mappings, make sure that:

- You have specified the primary key type, and defined the IDL structure fields if using a multi-column key (or this has been done by the EJB-JAR import process).
- You have defined the state type, or it has been defined by the EJB-JAR import process. See "Defining the state datatype" on page 511.
- If using timestamp columns for concurrency control, you have specified the timestamp column name on the Persistence/General subtab.

Configure field-mapping properties in the Component Properties dialog box, on the Persistence/Field Mapping subtab. This subtab displays a mapping for each container-managed field (based on the state datatype structure fields), the key fields, and the timestamp column (if specified). The initial mappings use default values which you may need to adjust.

Refreshing the field-mapping properties

If you do not see mappings for all fields:

1 Verify that the state type, primary key, and timestamp have been configured.

2 Click Ok in the Component Properties dialog box to save the properties.

3 Reopen the Component Properties dialog box.

Field mapping format

The mapping for each field has the form:

column[*typespec*]

Where:

- *column* is the database column name. You can use a table prefix, which is required if the table is not the main table (named in the Table field on the Persistence/General tab). For example, `custinfo.address` specifies the address column in the custinfo table.

  All fields that are in the primary key must be mapped to the main table. If you use the default queries in your table mappings, other tables must have key columns with the same name and type as the main table key.

The table name can be a logical table name that does not exist in the database. For example, your database may allow only stored procedure access. In this case, you must define table mappings that describe how to access the data represented by the logical table name. “Configuring table-mapping properties” on page 526 describes how.

- *type-spec* is the column’s database type, for example:

  string(255) not null

  or:

  binary(255) null

  The specified datatype is not necessarily the type used in the table schema. It can be redefined in the database properties. For example, `binary(255)` maps to `varbinary(255)` for the database type Sybase_ASE, and to `raw(255)` for Oracle8i. Similarly, `string(`***`length`***`)` maps to the appropriate type to define variable or fixed-length character columns of the specified length. Using the logical type names rather than actual database types allows you to more easily run the same configuration against databases of different types. For more information on these definitions, see Logical column type definitions in Appendix B, “Repository Properties Reference,” in the *EAServer System Administration Guide*.

  If the column is fixed-length binary or character, use one of these types, where *length* is the field length:

  binary(*length*) fixed length null
  binary(*length*) fixed length not null
  string(*length*) fixed length
  string(*length*) fixed length not null

Overriding the default database column names

For entity beans imported from an EJB-JAR file, the default field mappings use quoted database column names to avoid conflicts with database reserved words. In some cases, you may find that the quoted names exceed the maximum allowed for column names in the database. In this case, you can modify the column names after deploying, or add a *sybase-easerver-config.xml* file to your EJB-JAR file to set the field mapping properties before deploying. If you use quoted column names in the XML file, be sure to use the quote entity (`&quot;`) in place of quotes in property value strings.

For information on creating an *sybase-easerver-config.xml* file, see “Using EAServer configuration files in J2EE archives” in Chapter 9, “Importing and Exporting Application Components,” in the *EAServer System Administration Guide*.

Specifying finder- and ejbSelect-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. ejbSelect methods in an EJB 2.0 entity bean also require query mappings.

There are two ways to specify query mappings:

- **Using EJB-QL** EJB 2.0 entity beans can use EJB Query Language (EJB-QL) in the EJB-JAR deployment descriptor. EJB-QL allows portability among EJB 2.0-compliant servers. EAServer translates EJB-QL to SQL at runtime. You can configure additional EAServer query mapping properties to enable caching of the query results for improved performance.
- **Using extended SQL** If you cannot use EJB-QL, you must specify the query mappings using the EAServer extended SQL mapping language. This language is based on standard SQL, with extensions to allow substitution of method parameters and invocation of stored procedures. In some cases, no mapping is required: EAServer can correctly infer the query required to execute the findByPrimaryKey method. EAServer can also infer some queries based on finder-method naming patterns, for example:
  - findAll to return keys for all rows.
  - findByField where *field* is the name of a container-managed field in the component, to return the rows that match the specified *field* value.
  - findLikeField where *field* is the name of a container-managed field in the component, to return rows where the column mapped to *field* contains the specified *field* value

If you have deployed EJB CMP entity beans from an EJB-JAR file
If you have deployed EJB CMP entity beans from an EJB-JAR file, the special query mapping value `[unknown]` indicates that you must specify a query. The special value `[default]` means that EAServer can infer the query based on the method's name pattern. The special value `ejbQuery:` indicates the query uses EJB-QL that was specified in the deployment descriptor.

❖ **Configuring queries for methods**

1 Display the Persistence/Query Mapping subtab in the Component Properties dialog box. One mapping displays for each finder method in the component's home interface and for each ejbSelect method.

2 To change the query mapping, highlight the method name or query, then click Modify. Edit the value as follows:

Using EJB-QL: The special EAServer query `ejbQuery:` indicates that an EJB-QL query is defined for the finder or ejbSelect method. You can add additional parameters to configure caching of the query results, as described in "Entity instance and query caching" in the *EAServer Performance and Tuning Guide*, for example:

```
ejbQuery:[cache]
```

If the special mapping ejbQuery: is specified for the method, the EJB-QL query can be specified using a second query mapping for the method. The EJB-QL query must follow the syntax described by the EJB 2.0 specification.

In EJB-QL, each entity bean is represented by a schema name. EAServer defines the EJB-QL schema names in the properties of the package that contains the component. To map schema names to components in the package, specify a package property of the form:

```
schema:name=package/component
```

Where *name* is the schema name, *package* is the EAServer package name and *component* is the component name. If the package properties do not specify a schema name for the component, the default is the component name. Schemas used in queries for a component can only refer to components in the same EAServer package.

Using EAServer extended SQL: Enter one of the following for the query:

- `[default]` if EJB-QL is not used and EAServer can correctly infer the query.
- A SQL query appropriate for the semantics of the finder method, which can be a query filter or the syntax to call a stored procedure. For ordinary select queries that select only from the main table, you can omit the select keyword and column list, and specify only a where clause. Use the placeholders described in Table 27-2 to represent column and table names and parameter values.

Table 27-2: Finder query placeholders

Placeholder	To indicate
[key]	The table's primary key (which can consist of multiple columns).
[table]	The name of the main table, specified in the Table field on the Persistence/General subtab.
@param	Reference the value of parameter *param* in the finder method's IDL signature. **Note** If the component was imported from an EJB-JAR file, the IDL parameter names do not match the original Java implementation. The IDL parameters are named p0, p1, and so forth.
@param.fieldName	If method parameter *param* is not a simple type, reference the value of field *fieldName*.
[cache *cache-props*]	When appended to the query, configures query caching. See "Entity instance and query caching" in the *EAServer Performance and Tuning Guide*.

Note select queries for ejbSelect methods can return one column only.

Extended SQL examples

Use the syntax of these examples if you are using database tables that can be accessed directly with standard SQL select statements.

For simple queries that select only from the main table, you can omit the select keyword and column list as in this example. This query uses the value of the *expiryDate* parameter to filter a range of closingDate column values. Note the select keyword, column list, and from clause are omitted:

```
where closingDate < @expiryDate
```

When you omit the select keyword for a finder query, all columns are selected. If you are using query caching, selecting all columns allows EAServer to preload the query cache when the finder method executes.

If you include the select keyword and a column list in the query, you must use binding syntax (pseudoassignments) to bind the selected columns to the container-managed fields unless you are using the `[key]` placeholder for the table's primary key, as in:

```
select @field1 = col1, @field2 = (select col2 from t2
where t2.a = t1.a) from table1 t1, table2 t2
```

In this example, `field1` and `field2` are the names of the container-managed fields that are bound to the columns returned in the result set.

If you specify a column list and you are using entity object caching, specify a select list that returns all columns so that the object cache can be populated from the query results. Omit the select keyword, the column list, and the from clause to avoid this complication.

The query must be a complete select statement or omit the select keyword, column list, and from clause.

Calling stored procedures in extended SQL

You can call stored procedures to return the results required to execute a finder or ejbSelect query. To specify a stored procedure call, use the syntax:

select *column-list* from {call *proc param-list*}

Where:

- *column-list* contains placeholders for the field values and specifies their positions in the row returned by the procedure. Use the following syntax to indicate fields:
  - `@fieldName` to specify the value of field *fieldName*.
  - `@fieldName.subfield` to specify the value of subfield *subField* where *fieldName* is a container-managed field that takes structured types.

  If you are using entity object caching, the stored procedure should return values for all columns, so that the object cache can be populated when the procedure is called.
- *proc* is the stored procedure name.
- *param-list* is a parameter list that contains the parameters required by the stored procedure. Use the placeholder syntax described in Table 27-2.

For example, you might use this query for a findByPrimaryKey method:

```
select @firstName, @lastName, @ts from {call
sp_select_CustomerProcs @primaryKey}
```

Configuring table-mapping properties

Table mapping allows you to customize the DBMS queries and DML statements used to support ejbCreate/ctsCreate, ejbLoad/ctsLoad, ejbStore/ctsStore and ejbRemove/ctsRemove.

The default table mappings suffice for direct access to tables in standard SQL databases. You can customize the default SQL commands. For example, you might optimize the select query to force the use of an index by adding proprietary DBMS keywords.

You must configure explicit table access commands if you use stored procedures for data access or a non-SQL database.

To configure table mapping for an entity component, display the Persistence/Table Mapping subtab in the Component Properties window. Mapping properties display for the main table, specified on the Persistence/General tab, and any other table referenced in field mapping properties. Each table has mapping properties for the operations listed in Table 27-3.

Table 27-3: Table mapping operation names

Operation	Specifies
`select`	The database command for ejbLoad/ctsLoad operations.
`update`	The database command for ejbStore/ctsStore operations.
`insert`	The database command for ejbCreate/ctsCreate operations.
`delete`	The database command for ejbRemove/ctsRemove operations.
`notify`	When you are using object caching and have enabled database change notification, this property specifies the message service topic name used to notify the object cache of table changes. The default is the unqualified table name. See "Entity instance and query caching" in the *EAServer Performance and Tuning Guide* for more information.

For select, update, insert, and delete operations, the mapping can be `[default]`, to specify that standard SQL commands be used, a stored procedure call, or alternate query text.

Configuring stored procedure invocations

For update operations Specify the stored procedure call in the form:

{call *update-proc param-list*}

Where:

- *update-proc* is the stored procedure name. The procedure must perform the update given the supplied input parameters, and return no data. EAServer checks the JDBC row count to determine whether the update succeeded.

- *param-list* is the parameter list, which must include all container-managed fields. If a timestamp or version counter is used for concurrency control, it must also be in the parameter list. You can format parameters as:
  - `@fieldName` to specify the value of field *fieldName*.
  - `@fieldName.subfield` to specify the value of subfield *subField* where *fieldName* is a container-managed field that takes structured types.
  - `@old.fieldName` or `@new.fieldName` to specify the old (last read) or new (updated) value for field *fieldName*. If no `old` or `new` prefix is used, the new value is assumed. The `old` and `new` prefixes cannot be applied to primary key fields, because an instance is not allowed to change the primary key.

For example:

```
{call sp_update_CustomerProcs @primaryKey, @firstName,
@lastName, @old.ts}
```

The update procedure must contain logic to perform concurrency control. If using a timestamp column, make sure the timestamp value is passed to the procedure and used in the update statement. If using OCC with value comparisons, make sure the procedure accepts all old values as well as new values, and contains the value comparison logic.

For delete operations Specify the stored procedure call in the form:

{call *delete-proc param-list*}

Where:

- *delete-proc* is the stored procedure name.
- *param-list* is the parameter list, which must contain parameter values for the primary key columns, using placeholders as described for update procedure.

For example:

```
{call sp_delete_CustomerProcs @primaryKey}
```

For insert operations Specify the stored procedure call in the form:

{call *insert-proc param-list*}

Where:

- *insert-proc* is the stored procedure name. If the component uses generated primary keys, the procedure must return a result set containing the new key. Otherwise, the procedure must not return any data.

- *param-list* is the parameter list, which must contain all values for the new row, unless using automatic key generation. If keys are generated, omit the key from the parameter list. The parameter format is the same as for the update procedure, except that the `old` and `new` prefixes are not supported.

For example:

```
{call sp_insert_CustomerProcs @primaryKey, @firstName, @lastName}
```

For select operations Specify the stored procedure call in the form:

select *read-param-list* from {call *select-proc key-param-list*}

Where:

- *read-param-list* contains placeholders for the field values and specifies their positions in the row returned by the procedure. The parameter format is the same as for the update procedure, except that the `old` and `new` prefixes are not supported. The stored procedure does not need to return the key value.
- *select-proc* is the stored procedure name.
- *key-param-list* is a parameter list that specifies all the primary key columns. The parameter format is the same as for the update procedure, except that the `old` and `new` prefixes are not supported.

For example:

```
select @firstName, @lastName, @ts from {call
sp_select_CustomerProcs @primaryKey}
```

Specifying alternate query text

You can enter query text for the insert, delete, update, and select operations, using the same parameter placeholder format as used for stored procedures.

Special syntax is required also for ordinary SQL select statements to specify the mapping of fields to expected result set columns. For example, if storing customer names in a separate table and not using stored procedures, you might specify the select operation as:

```
select=select @firstName = firstName, @lastName = lastName from
TestCMP_Customer where primaryKey = @primaryKey
```

When this query is issued to the JDBC driver, it will be in the form of a JDBC prepared statement, such as:

```
select firstName, lastName from TestCMP_Customer where primaryKey
= ?
```

The persistence engine removes the "@field" references from the query, allowing the proprietary syntax of the target DBMS to be used effectively. For example, the query could be modified to force the use of an index by using proprietary DBMS keywords.

Using relationship components

EAServer uses relationship components to manage relationships between EJB 2.0 CMP entity beans.

EBJ 2.0 CMP entity beans can have container-managed relationships. A relationship allows an entity component to have a container-managed field that contains instances of another (or the same) entity component. For example, an *Order* component may have an *items* field that consists of a collection of *Inventory* objects representing the items being purchased. Or, an *Employee* component may be related to itself, with *manager* and *employees* fields that contain Employee instances.

A relationship can be unidirectional or bidirectional. For example, the Employee-Manager relationship is typically bidirectional: it's convenient to know who works for a particular employee as well as who reports to that employee. An Order-Inventory relationship is typically unidirectional: it would not be practical to track every order that a line item is added to.

The relationship component name contains one or more hyphens. Only relationship components may have names that contain hyphens. Relationship components created by deploying EJB-JAR files have names of the form:

component1-component2

Or:

`component-field`

Where *component1* and *component2* are names of two related components. The ***`component-field`*** form is used when a component is related to itself (such as the employee-manager relationship). In this case,. *field* is the field name used for maintaining one side of the relationship. For example, an Employee component may have a Manager field, resulting in the relationship component *Employee-Manager*.

Relationship components are supported only for EJB 2.0 CMP entity beans. Relationship components are themselves CMP entity beans, and must have the relationship properties described below. Related components must be in the same EAServer package. The implementation of a relationship component is generated when you generate skeletons for the components in the relationship.

Maintaining the relationship

The tables represented by related components must be related in the database. There are two techniques to maintain the table relationship:

- **Using foreign keys** The source table stores the key of the related table as a column.
- **Using a join table** A separate table relates keys between the two tables.

Foreign keys offer the best performance, but can be used only when one destination instance relates to a given source instance. For example, you can use foreign keys to manage the Employee-to-Manager relationship (an employee has one manager), but must use a join table for the Manager-to-Employee relationship (a manager has many employees). In bidirectional relationships, you must configure the technique for each direction of the relationship.

The Table field in the Persistence/General properties specifies the name of the join table. When configuring components to work with existing database tables, the join table must exist and contain the key pairs to describe the relationship.

Cascading deletes

In some cases, you may want an ejbRemove operation to delete "through" a relation. For example, an Order instance represents an online purchase, and is related to LineItem instances that represent items included in the order. In this case, removal of an Order instance should remove LineItem instances contained in the order.

You can configure the Cascade Delete properties for each side of a one-to-one relationship, and on the "one" side of many-to-one and one-to-many relationships.

Configuring relationship component properties

To configure relationship properties, display the Component Properties for the relationship component, then display the Persistence/Relationship subtab and configure the Relationship Type settings and Relationship Name settings, described in Table 27-4 and Table 27-5, respectively.

Regenerating the relationship component
After editing relationship component properties, regenerate the skeletons for the package and refresh the package to ensure the changes take effect.

Table 27-4: Relationship Type properties

Property	Specifies
Type	The cardinality of the relationship. Allowable values are: • One to One – one *from* component instance is related to one *to* instance. • One to Many – one *from* instance is related to many *to* instances. • Many to One – many *from* instances are related to one *to* instance. • Many to Many – many *from* instances are related to many *to* instances.
From Component	The name of the *from* component in the relationship, in the form: *package*/*component* The *from* component contains a container-managed field that contains instances of the *to* component specified by the To Component property.
From Field	The name of the container-managed field in the *from* component that contains related *to* component instance references.
From Field Type	For one-to-many and many-to-many relationships, specifies the Java type used in the getter and setter methods of the *from* component. Allowable values are Collection and Set. For single-valued fields, no value is required. You can set this property by setting the `from-field-type` property on the Advanced tab.
From Query	When a join table is used, the name of a query used to select the *to* component's primary keys that are required to populate the *from* component field indicated by the From Field setting. This query must be defined by a query mapping property, as described in "Specifying finder- and ejbSelect-method queries" on page 523.
From Role	Matches the name of the corresponding `ejb-relationship-role` element in the EJB-JAR deployment descriptor.
Use Foreign Key	Whether to use a join table or foreign keys to maintain the *to-from* relationship. A value of true indicates that foreign keys must be used. You can use foreign keys only on the "one" side of a relationship, as described in "Maintaining the relationship" on page 531.
Cascade Delete	Applies only when the relationship-type is one-to-one or one-to-many. Specifies whether deletion of a an instance on the singleton side of the relation causes deletion of the related instance.

Table 27-5: Relationship Name properties

Property	Specifies
Name	The name of the relationship. Matches the corresponding `ejb-relation-name` element in the EJB-JAR deployment descriptor.

Property	Specifies
To Component	The name of the *to* component in the relationship, in the form: *package*/*component* For bidirectional relationships, you must also specify values for the To Field, To Field Type, and To Query properties.
To Field	The name of the container-managed field in the *to* component that contains related *from* component instance references.
To Field Type	If a *to* instance can be related to multiple *from* instances, specifies the Java type used in the getter and setter methods of the *from* component. Allowable values are Collection and Set. For single-valued fields, no value is required. You can set this property by setting the `to-field-type` property on the Advanced tab.
To Query	When a join table is used, the name of a query used to select the *from* component's primary keys that are required to populate the *to* component field indicated by the To Field setting. This query must be defined by a query mapping property, as described in "Specifying finder- and ejbSelect-method queries" on page 523.
To Role	Matches the name of the corresponding `ejb-relationship-role` element in the EJB-JAR deployment descriptor.
Use Foreign Key	Whether to use a join table or foreign keys to maintain the *from-to* relationship. Enable the option if foreign keys must be used. You can use foreign keys only on the "one" side of a relationship, as described in "Maintaining the relationship" on page 531.
Cascade Delete	Applies only when the relationship-type is one-to-one or many-to-one. Specifies whether deletion of a an instance on the singleton side of the relation causes deletion of the related instance.

Example properties for a many-to-many bidirectional relationship

The components here are *TestCMP/Customer* and *TestCMP/Account*:

```
relationship-type=many-to-many
relationship-name=Customer-Account
relationship-from=TestCMP/Customer
relationship-to=TestCMP/Account
from-field=accounts
from-field-type=Collection
from-foreign-key=true
from-query=findByCustomer
from-role=customer-has-accounts
to-field-type=Collection
to-field=customers
to-foreign-key=true
to-query=findByAccount
to-role=account-has-customers
cascade-delete=false
```

Example properties for a recursive, bidirectional, many-to-one relationship

The component here, *TestCMP/Employee*, is related to itself:

```
relationship-type=many-to-one
relationship-name=Employee-Manager
relationship-from=TestCMP/Employee
relationship-to=TestCMP/Employee
from-field=managerField
from-foreign-key=true
from-query=findByEmployees
from-role=employees-has-manager
to-field=employeesField
to-field-type=Collection
to-foreign-key=true
to-query=findByManager
to-role=manager-has-employees
cascade-delete=false
```

CHAPTER 28

Configuring Persistence for Stateful Session 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. However, instances of a component coded this way can run on one server only, and cannot support load balancing or failover. Using persistent state storage allows your component to participate in failover and load balancing. EAServer also uses database storage to support the EJB session bean passivation and activation mechanism.

How it works

State data can be stored either in memory or to a persistent data store:

- *In-memory storage* uses a mirror-pair model where data is replicated between pairs of servers running in the cluster. In-memory storage offers better performance than persistent storage, but each client session has two points of failure (the originating server, and its mirror-pair twin). In-memory storage uses the EAServer message service to replicate data between servers in a mirror pair. See “Requirements for in-memory stateful failover” on page 549 for more information.

- *Persistent storage* uses a remote database to store component state. A component instance can failover to any other server in the cluster where the component is installed. Persistent storage requires a highly available database, otherwise the database itself can become a single point of failure.

The stateful failover architecture includes:

- A *storage component*, which stores state data to a remote database or, for in-memory storage, calls the message service to replicate the data to the other server in the mirror pair. EAServer provides several storage components for database storage and for in-memory replication. You can also provide your own, custom implementation.
- The component's *state datatype*, which is an IDL structure or serializable class that allows transfer of state data from the stateful component instance to the storage component. For EJB stateful session beans, the state type is the implementation class (which is serialized to save the state).
- The component's *state accessor methods*, which the component or skeleton implements to interact with the storage component. When the client calls a business method, new state may be created in the component. After the business method returns, the storage component calls the *get* method to obtain the state data, passed as an instance of the state datatype. If the component fails over to another server, a new instance is created and the storage component calls the new instance's *set* method, providing saved state data to initialize the new instance with the client's session data.

  For EJB stateful session beans, EAServer generates a skeleton with state accessor methods that get and set instance state using the standard EJB passivation and reactivation protocol. For components of other types, you must implement accessor methods and specify their names in component properties as described below

The server takes care of converting the specified state data type to and from a form suitable for persistent storage or in-memory replication. This feature is very powerful because any IDL datatype (or serializable Java class) can be used as the state type. In many cases, the most suitable state type is an IDL struct type, which can be created in the IDL editor (required when using C++ or COM components), or generated automatically from a PowerBuilder structure type by the PowerBuilder deployment process.

Supported component implementations

To use persistent state management, a stateful component must be an EJB stateful session bean or a component of another type that uses the control interface CtsComponents::ObjectControl. (See “Configuring a control interface” on page 72.)

You can manage persistence using these techniques:

- Using EJB activation and passivation

  This model can be used only in EJB stateful session Beans. To save persistent state, the state accessor methods in the component skeleton serialize the component class instance and saves the binary data to the database. To restore state, the saved data is deserialized.

- Using automatic persistence

  Use this model for non-EJB components. In this model, you define a state datatype in IDL or Java and implement component methods to receive state data and return state data. The server calls your state access methods, and manages interaction with the database.

Using EJB activation and passivation

This stateful persistence model is how EAServer implements the standard EJB passivation and activation protocol. This model can be used only by EJB stateful session beans. In EJB terminology, *passivation* is the process of removing an instance’s data from memory and saving it to a database. *Activation* is the process of restoring the state and applying it to an instance of the component. You can configure passivization for single-server and clustered-server deployments as described in Table 28-1.

Table 28-1: EJB stateful session bean passivation options

Option	Description
No passivation	This is the default configuration for EJB stateful session beans. EAServer never performs passivation. If you configure an instance timeout property for the component, instances that time out are destroyed and removed from memory. Subsequent use of the instance handle by a client results in an error.

Option	Description
Serialization with support for load balancing and failover	For clustered (multiserver) deployments. EAServer serializes instance state and saves it to a remote database after every remote interface method invocation. This option can be used to support failover and load balancing in an EAServer cluster and also supports passivation as required by the EJB specification. “Configuring stateful session beans to support failover” on page 538 describes how to configure this option.
Serialization after timeout	For single-server deployments. EAServer serializes instance data when the instance times out, saving the data to a remote database. This option allows you to support EJB passivation without incurring the overhead of a database write after every method invocation. You can also tune the timeout setting to balance memory use versus response time. “Configuring passivation after timeout” on page 539 describes how to configure this option.

Configuring stateful session beans to support failover

These settings configure EAServer to save in a remote database a copy of the the session bean’s serialized data after every method invocation. When running in a cluster, the instance data can be restored on any server in which the component is installed, permitting load balancing and failover of the component. Configure the settings in Table 28-2, using the Persistence Tab in the Component Properties dialog box:

Table 28-2: EJB stateful session failover options

Field	Value
Persistence	Automatic Persistent State.
State	The name of the component’s Java class.
Primary Key	None (leave blank).
State Methods	Enter `default`.
Storage components	The name of the storage component. See “Storage components” on page 545 for more information.
Connection cache	If you are using database storage, enter the name of a JDBC connection cache that connects to the database. The cache must have by-name access enabled.
Table	If you are using database storage, enter the name of a database table where the serialized data is to be stored. If you use Adaptive Server Enterprise or Adaptive Server Anywhere, EAServer creates the table if it does not exist. When you are using another data server, you or your database administrator (DBA) must create the table manually. The table must have the schema described in “Table schema for binary storage” on page 549.

Configuring passivation after timeout

For single-server deployments, these settings configure EAServer to passivate instances when the instance has been idle for the period specified by the Instance Timeout property. EAServer serializes the instance, saves the data in a remote database, and removes the instance from memory. When using this option, you can tune the timeout setting to balance your application's memory use against the average response time.

The following component properties that must be set in addition to the default EJB stateful session bean component settings:

- com.sybase.jaguar.component.timeout – Enter the instance time out period, in seconds.
- com.sybase.jaguar.component.ps – Set to `serialize`.
- com.sybase.jaguar.component.state.gs – Set to `default`.
- com.sybase.jaguar.component.transient – Set to `false`
- com.sybase.jaguar.component.storage – Set to:

`CtsComponents/JdbcStorage(cache=`*`c_name`*`,table=`*`t_name`*`,transient)`

Where *c_name* is the name of a JDBC connection cache that connects to the target database and *t_name* is the name of the table used. If you use Adaptive Server Enterprise or Adaptive Server Anywhere, EAServer creates the table if it does not exist. When you are using another data server, you or your database administrator (DBA) must create the table manually. The table must have the schema described in "Table schema for binary storage" on page 549.

In EAServer Manager, use the Advanced tab in the Component Properties dialog box to configure these properties. You can also use jagtool or jagant to configure components that use this option, or add an EAServer configuration file to your EJB-JAR file that configures the component properties. See the following references in the *EAServer System Administration Guide* for more information:

- Chapter 12, "Using jagtool and jagant"
- "Using EAServer configuration files in J2EE archives" in Chapter 9, "Importing and Exporting Application Components"

Using automatic persistence

To use automatic persistence, you must define a state datatype to hold your component's instance state, implement accessor methods, and choose a storage component.

❖ **Configuring a stateful component to use automatic persistence**

1 Display the component's properties, then click the Persistence tab. Configure the Persistence/General tab settings as follows:

Setting	Value
Persistence	Choose Automatic Persistent State.
State	Enter the name of an IDL structure that contains your component's state data, for example: `TheCart::CartState` "Defining the IDL state type" on page 541 describes how to create this structure.
State methods	Enter the names of the component methods that retrieve and apply an instance's state data. If you specify no value, the default is `getState,setState`. Your component implementation must contain these methods, but they should not be listed in the component's client interfaces. "Accessing the state data in the implementation" on page 542 describes how to implement these methods.
Storage component	Specify the name of the storage component, as described in "Storage components" on page 545.
Connection cache	If using database storage, enter the name of a JDBC connection cache that connects to the database. The cache must have by-name access enabled.
Table	If using database storage, enter the name of a database table where the serialized data is to be stored. If you use Sybase Adaptive Server Enterprise or Adaptive Server Anywhere, EAServer creates the table if it does not exist. When using another data server, you or your database administrator (DBA) must create the table manually. The table must have the schema described in "Table schema for binary storage" on page 549.

2 Define the IDL state type as described below.

3 Add code to retrieve, apply, and modify the state data as described below.

4 Regenerate stubs and skeletons for the component—this step will generate the Java classes for the IDL state types.

5 If using in-memory storage, configure the component's Mirror Cache properties and cluster to support in-memory failover, as described in "Requirements for in-memory stateful failover" on page 549.

Defining the IDL state type

The IDL state type is a structure that must hold all the session data for the bean. The following IDL module shows example types used for a shopping cart component:

```
module TheCart
{
  // This is the state type, an IDL structure that holds customer data
  // and the collection (sequence) of items in the cart.
  struct CartState
  {
    string name;
    string address;
    string phone;
    ::TheCart::ShoppingCartItems items;
  };

  // "ShoppingCartItems" is the sequence to hold items in the cart:
  typedef sequence < ::TestInMemoryFailvoer::ShoppingCartItem >
ShoppingCartItems;

  // "ShoppingCartItem" holds the data for one item in the cart:
  struct ShoppingCartItem
  {
    string item;
    long quantity;
  }
}
```

In the State field on the Persistence/General tab, you can enter the name of an IDL structure that does not exist. EAServer Manager creates the structure when you close the Component Properties dialog box. Afterwards, navigate to the module definition under the top-level IDL folder and edit the structure definition. For information on editing IDL in EAServer Manager, see Chapter 5, "Defining Component Interfaces."

Accessing the state data in the implementation

Make the following changes to the component implementation:

- **Add the state accessor methods** Add methods with the names specified in the State Methods field in the Persistence/General tab in the Component Properties dialog box. Also add an instance variable of the generated state type. For example:

```
import TheCart.CartState;
private CartState data = new CartState();
private int lastItem = 0;

TheCart.CartState getState()
{
    return data;
}

void setState(TheCart.CartState state)
{
    data = state;
}
```

- **Add code to initialize the state type** Add code to your comonents' ctsCreate method to initialize the state type. For example:

```
public void ctsCreate(
    name, address, phone, capacity)
{
  data.name = name;
  data.address = address;
  data.phone = phone;
  data.items = new ShoppingCartItem[capacity];
}
```

  These assignments correspond to the IDL structure fields in the example above. The *items* IDL sequence is represented by an array in Java.

- **Add code to access the state data from business methods** Where appropriate, the bean's business methods should read and write from the state type. For example, these methods add an item to the shopping cart and return the customer's name, respectively:

```
public void addItem(String item, int quantity)
{
  ....
  lastItem++;
  data.items[lastItem]
      = new ShoppingCartItem(item, quantity);
```

```
  ....
}

public String getCartName()
{
  return data.name;
}
```

CHAPTER 29

Configuring Persistence Mechanisms

You can code components to store state information in a remote database rather than in memory. Doing so offers several advantages, such as enabling failover and load balancing for stateful components, or allowing you to map relational data to a component interface using the EJB entity bean model. The EAServer persistent state model is based on standard EJB interfaces, but you can use it in components of other types. For more information, see these chapters:

- Chapter 27, “Creating Entity Components”
- Chapter 28, “Configuring Persistence for Stateful Session Components”

The remainder of this chapter contains reference material that is useful in configuring stateful session components and entity components.

Storage components

A storage component read and writes component state information from a remote database server or replicates state information to other servers for in-memory stateful failover. If your component uses automatic persistence, Java serialization, or component-managed persistence with an implementation that delegates to EAServer’s built-in storage component, you must specify the storage component used to interact with the persistent data store.

When using persistent (database) storage, the storage component uses the connection cache and database table identified on the Persistence/General subtab in the Component Properties dialog box.

For EJB stateful session beans, *HeapStorageReqNew* or *JdbcStorageReqNew* are the recommended storage components. For EJB CMP entity beans, *JdbcStorage* or *JdbcStorageReqNew* are recommended, and you cannot use in-memory storage. Table 29-1 describes the storage components available in EAServer.

Table 29-1: EAServer storage components

Storage component	Characteristics
CtsComponents/ JdbcStorage	Uses a JDBC connection cache to provide persistent storage of component state applied in the context of the current EAServer transaction. This component has the Requires transaction attribute. The component's state is saved in the context of any existing transaction associated with the component. Consequently, a transaction rollback rolls back changes to the state data.
CtsComponents/ JdbcStorageReqNew	Database storage using a JDBC connection cache, applied in the context of a new transaction. A transaction rollback does not affect the storage of state data.
CtsComponents/ HeapStorage	For stateful components only, not for entity components. Supports in-memory failover using a mirror-pair model to replicate state between pairs of servers. Changes are applied in the context of the current EAServer transaction. A transaction rollback rolls back changes to the state data. Additional configuration is required to use in-memory storage, as described in "Requirements for in-memory stateful failover" on page 549.
CtsComponents/ HeapStorageReqNew	For stateful components only, not for entity components. Supports in-memory failover using a mirror-pair model to replicate state between pairs of servers. In-memory storage applied in the context of a new transaction. A transaction rollback does not affect the storage of state data. Additional configuration is required to use in-memory storage, as described in "Requirements for in-memory stateful failover" on page 549.

Storage component	Characteristics
A custom storage component.	You can specify the name of your custom storage implementation, entered as *package*/*component*, where *package* and *component* are the package and component names, respectively, as displayed in EAServer Manager. The package must be installed on all servers where your component is installed. Customers and partners can implement custom storage components. The component must implement the CtsComponents::Storage interface and must have the Bind Object option enabled on the Instances tab. Due to the thread-safe instance requirement for the Bind Object option, only C++ and Java are suitable for coding storage components.

Supported Java, IDL, and JDBC/SQL types

Table 29-2 lists the Java, IDL, and JDBC/SQL types that EAServer 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 29-2: 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
(N/A)	unsigned short	SMALLINT
int	long	INTEGER
(N/A)	unsigned long	INTEGER
long	long long	BIGINT
(N/A)	unsigned long long	BIGINT
float	float	REAL
double	double	FLOAT
(N/A)	string	VARCHAR

Java field type	CORBA IDL field type	JDBC/SQL column type
(N/A)	BCD::Binary	VARBINARY
(N/A)	BCD::Decimal	DECIMAL
(N/A)	BCD::Money	DECIMAL
(N/A)	MJD::Date	DATE
(N/A)	MJD::Time	TIME
(N/A)	MJD::Timestamp	TIMESTAMP
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
(N/A)	CtsComponents::UShortValue	SMALLINT
java.lang.Integer	CtsComponents::LongValue	INTEGER
(N/A)	CtsComponents::ULongValue	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
(N/A)	CtsComponents::MoneyValue	DECIMAL
java.sql.Date	CtsComponents::DateValue	DATE
java.sql.Time	CtsComponents::TimeValue	TIME
java.sql.Timestamp	CtsComponents::TimestampValue	TIMESTAMP
java.lang.Object (as primary key)	Xdt::IntegerValue	IDENTITY
Serializable Java object	(N/A)	VARBINARY

Values that can be null

If a field can contain nulls, do not use a primitive type. Instead, use the CtsComponents::*Type*Value 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 the schema described in Table 29-3.

Table 29-3: Table schema for binary storage

Column	Data format
ps_key (primary key)	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): • 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 *any other* primary key type, including java.lang.String, ps_key must be variable length binary, 255-byte maximum length. This column cannot be null.
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.

Requirements for in-memory stateful failover

You can use in-memory storage to support failover for stateful components by choosing *CtsComponents/HeapStorage* as the storage component. This feature allows component state to be maintained on a pair of servers, without incurring the overhead of using a remote database to store component state.

The in-memory failover implementation is based on *mirror pairs*. A mirror pair consists of two servers that are members of a cluster. The servers use the EAServer message service to synchronize component session state held in memory. If one server in a mirror pair goes offline, the other remains to serve client sessions for the mirrored components. You can configure multiple mirror pairs within a cluster, but each server can be a member of only one mirror pair.

In-memory failover requires the following:

- A cluster with mirror pairs configured as described in "Cluster configuration for in-memory failover" on page 550.
- Component Mirror Cache configuration as described in "Mirror Cache tab component properties" on page 551.
- A working message service on each server that is in a mirror pair. Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide* describes how to configure the message service.

Cluster configuration for in-memory failover

Chapter 6, "Clusters and Synchronization," in the *EAServer System Administration Guide* describes how to configure a cluster. To support in-memory failover, you must define mirror pairs within the cluster.

A mirror pair consists of two servers within the cluster that use the EAServer message service to replicate state information for session components hosted on those servers. Servers in a mirror pair should have the same set of stateful components installed. One server cannot be a member of more than one mirror pair.

❖ **Configuring mirror pairs**

1 Display the Mirror Groups tab in the Cluster Properties dialog box.

2 Click Add.

3 Enter the IIOP URLs for the two servers in the mirror pair, separated by a comma. For example:

```
iiop://mypc:9000,iiop://yourpc:9100
```

4 Repeat to add as many mirror pairs as required.

❖ **Configuring server session cache size**

- For each logical server in a mirror pair, configure the server property `com.sybase.jaguar.server.ps.cache.size`. This property specifies the maximum size of the memory cache used to hold session data for components running on the server. The value has the same syntax as the Cache Size property described in "Mirror Cache tab component properties" on page 551.

Mirror Cache tab component properties

On the Mirror Cache tab, configure the following:

- **Cache Size** Specifies the maximum size of the cache used to hold session state for instances of this component. Specify the size in megabytes, kilobytes, or bytes with the syntax shown in the following table:

Cache size value syntax	To indicate
*n*M or *n*m	*n* megabytes, for example: `512M`
*n*K or *n*k	*n* kilobytes, for example: `1024K`
n	*n* bytes, for example: `536870912`

  The component's cache size cannot be greater than the size of the server's in-memory session cache (specified by the `com.sybase.jaguar.server.ps.cache.size` server property). If the cache is not large enough, clients may experience cache overflow errors. When this happens, the least recently accessed instance is removed from the cache. If a client attempts to invoke an instance, the client receives a CORBA::OBJECT_NOT_EXIST exception.

- **Synchronization** Specify "mirror".
- **Timeout** Specify a cache timeout value as a positive integer. This value is the number of seconds that cached state data remains valid. The cache timeout must be less than the Instance Timeout setting, and in most cases should be the same. The default is 10000 seconds.

CHAPTER 30

Configuring Custom Java Class Lists

Understanding how the class loader works

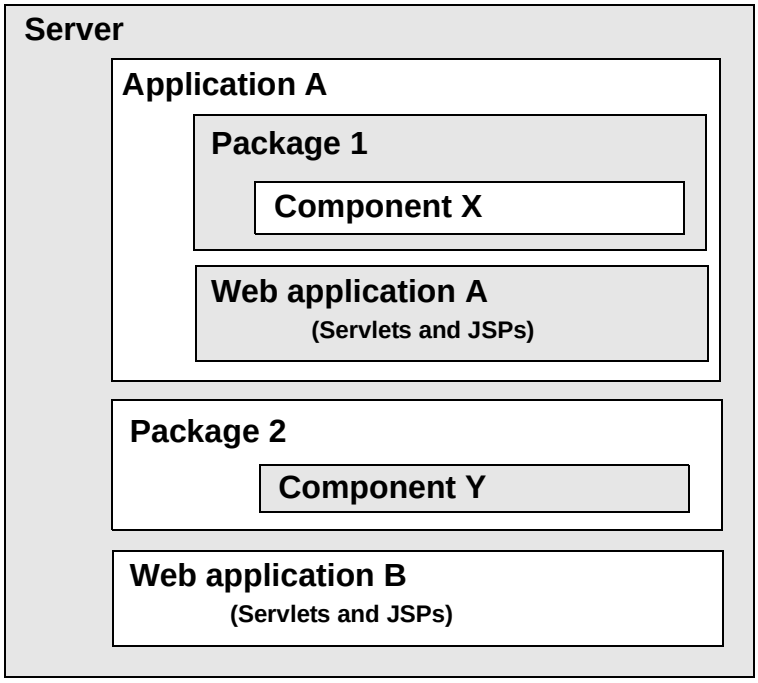
In Java, a *class loader* loads the Java classes used by an application. Most applications use the Java system class loader, which loads classes from the directories and JAR files specified by the CLASSPATH environment variable. In the normal Java program configuration, you must restart a program or server to begin using updated Java classes. EAServer uses customized Java class loaders to allow *hot refresh* of Web application classes and Java components without restarting the server. EAServer provides custom class loaders for these entity types:

- **Component** For CORBA/Java and EJB components, allows you to define the list of classes that must be custom loaded in addition to the component implementation class.
- **Package** Allows you to define a custom class list shared by all Java components that are installed in the package.
- **Web application** Allows you to specify classes and JAR files that are not in the Web application's *WEB-INF/lib* or *WEB-INF/classes* directories, but must be custom loaded for the Web application.
- **Application** Allows you to define a custom class list shared by all components and Web applications that are installed in the J2EE Application.

- **Server** Allows you to define a custom class list shared by all components, servlets, and Web applications running on the server.
- **Servlet** For servlets that are not installed in a Web application, but are installed directly in a server, allows you define the list of classes that must be custom loaded in addition to the servlet implementation class.

Each server, component, Web application, application, and package has a custom class loader associated with it. The EAServer class loaders have a a parent-child hierarchy. Every class loader except the server class loader has a parent. This relationship is shown in the following figure:

Figure 30-1: EAServer class loader hierarchy



The package, application, and server custom class lists allow you to configure sharing of common utility classes used by components and Web applications. By custom loading these classes at a higher level, you avoid the overhead incurred by custom-loading many copies of the same class.

When you specify the same class at multiple levels, it is always loaded at the highest possible level. For example, if you specify Java package X in a component's class list, the class list of the package where X is installed, and the class list of the server where the package is installed, the server class loader loads classes in package X.

Minimize Refresh in production servers
Refreshing components loads additional copies of all implementation classes. EAServer leaves the previous implementation in memory for use by existing client sessions. In effect, refresh introduces a controlled memory leak. For this reason, it is best to restart your production server after deploying a large number of new implementation classes.

The system class loader

Classes not loaded by the custom class loader must be loaded by the system class loader, based on the search order specified by the server class path setting. These classes cannot be refreshed without restarting the server. It can be more efficient configure system class loaders for classes that are used server wide, as long as all components that use them require the same class versions, and you do not need to refresh the classes without restarting the server. (For classes used server-wide that may be updated, you can configure sharing of the classes at the server class loader level, as described in "Custom class lists for packages, applications, or servers" on page 560.)

By default, the server class path includes these entries:

- Class files deployed in the class trees rooted in the EAServer *java/classes* and *html/classes* subdirectories.
- All JAR files in the *java/lib* directory that exist when the server starts, exclusive of *jagtool.jar*, *jdmkrt.jar*, and *jdkmktk.jar*.

You can add classes or JAR files to the server's class path using any of these techniques:

- By placing a JAR file that contains the classes in the EAServer *java/lib* subdirectory. At startup, EAServer automatically configures the CLASSPATH and BOOTCLASSPATH environment variables to include JAR files that you have placed in this location.

- By modifying the CLASSPATH and BOOTCLASSPATH environment variables in the shell where you start the server. You can do this by adding settings to the *bin/user_setenv.bat* (Windows) or *bin/user_setenv.sh* (UNIX) scripts. EAServer applies these settings automatically when you start the server. However, they are also applied in the environment for other tools, such as EAServer Manager and jagtool.
- By modifying the following server properties, using the advanced tab in the EAServer Manager server properties dialog box:
  - com.sybase.jaguar.server.jvm.classpath to configure the CLASSPATH setting.
  - com.sybase.jaguar.server.jvm.classpath.jars to specify JAR files in the *java/lib* directory to add to the CLASSPATH setting.
  - com.sybase.jaguar.server.jvm.bootclasspath to configure the BOOTCLASSPATH setting.
  - com.sybase.jaguar.server.jvm.bootclasspath.jars to specify JAR files in the *java/lib* directory to add to the BOOTCLASSPATH setting.

  For syntax information, see the reference pages in Appendix B, "Repository Properties Reference," in the *EAServer System Administration Guide*.

Deciding which classes to add to the custom list

Since system loaded classes require a server restart to update, use system loading only for classes that never change. For classes that might be refreshed with a component or Web application, add the classes to the custom list. The following sections give more specific guidelines for each entity type.

Custom class lists for Java and EJB components

The standard locations for deploying Java and EJB component class files is the *java/classes* directory. EAServer also generates the component's stubs and skeleton classes under this directory.

A Java component's implementation class and stub classes are automatically part of the custom class list for the component. Add the following additional classes to the custom list:

- Stub classes used for intercomponent calls. For EJB local interface calls, you must also configure sharing of the class instances as specified in "Calling local interface methods" on page 150.
- 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.
- For EJB components, classes that extend javax.naming.InitialContext or other javax.naming classes and that are called by your component.
- Additional classes that must be reloaded when the component is refreshed. For example, if you are debugging utility classes used by the component, add these classes to the custom list.

Component-specific JAR files should be deployed in the EAServer *java/classes* subdirectory and added to the custom class list.

To configure the custom list, follow the instructions in "Configuring an entity's custom class list" on page 562.

JNDI classes EAServer never custom loads the standard Java Naming and Directory (JNDI) packages. In EAServer versions prior to 4.0, an EJB component required the following in the custom class list:

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

In EAServer 4.0 and later, these classes and interfaces are ignored when listed in the custom class list.

Class loading order for components

When loading classes listed in a Java or EJB component custom list, the component class loader searches for classes in this order:

1 Any JAR file that is listed in the component's custom class list property and deployed in the EAServer *java/classes* subdirectory.

2 Classes listed in the component's custom class list and deployed under the EAServer *java/classes* subdirectory.

3 Classes listed in the component's custom class list and deployed under the EAServer *html/classes* subdirectory.

4 Any JAR file that is listed in the component's custom class list property and deployed in the EAServer *java/lib* subdirectory.

5 If the class is not loaded at this point, it is loaded by the system loader using the search order specified in the server class path—see “The system class loader” on page 555.

Custom class lists for Web applications

A Web application’s custom class list must include any classes that must be reloaded when the Web application is refreshed. For example, servlet implementation classes, utility classes called by servlets and JSPs, and stub classes for component invocations should be in the custom class list.

If you deploy classes and JAR files that the standard Web application deployment locations, you do not need to explicitly list them in the custom class list. The standard Web application deployment locations are under your Web application’s context root, the EAServer subdirectory:

```
Repository/WebApplication/WebApp
```

Where *WebApp* is the Web application name. Deploy classes and JAR files to these subdirectories of the context root:

- *WEB-INF/classes* for class files.
- *WEB-INF/lib* for JAR files that contain archived class files.

EAServer automatically adds classes and JAR files that are deployed in the standard locations to the custom class list. Use these locations to deploy classes and JAR files that are specific to your Web application. These directories are equivalent to the like-named directories in a J2EE WAR file.

For classes that are also used in EJB or Java components or other Web applications, you can deploy classes or JAR files in one of the locations below. For example, if your servlet calls an EJB component, you may want the servlet and component to use the same copies of the component stub classes. You must explicitly list these classes or JAR files in your Web application’s custom class list:

- The EAServer *java/classes* or *html/classes* directory, for class files that are also used by components.
- The EAServer *java/classes* or *java/lib* directory, for JAR files that are also used by components.

- The EAServer *extensions* subdirectory for JAR files used by multiple Web applications, as described in "Using Java extensions" on page 399. These JAR files can be used by any Web application that lists the JAR file name in the custom class list.

Though a component and Web application may custom load the same classes, to enable sharing of one copy of each class, you must configure the same custom class list entries in parent entities up to a common ancestor entity, as described in "Custom class lists for packages, applications, or servers" on page 560.

Class loading order for Web applications

EAServer loads classes listed in the Web application custom class list by searching the following code bases, in the order specified. All of these locations are subdirectories of your EAServer installation directory, *app_name* represents the name of your Web application, and *server* is the server where the Web application is installed. If the custom class loader cannot locate a class file, the server attempts to load it using the system class loader.

1 Class files in *work/server/Servlet/WebApp-app_name*

   This directory contains generated servlet classes for compiled JSPs.

2 Class files in *Repository/WebApplication/app_name/WEB-INF/classes*

   This is the standard directory for servlet, filter, tag-library, and utility classes used by the Web application. If utility classes are shared with Java components running on the same server, you may wish to move them to the EAServer *java/classes* subdirectory and add them to the custom class list for your application or server.

3 JAR files in *Repository/WebApplication/app_name/WEB-INF/lib*

   This is the standard location for JAR files that contain classes to be used only by this Web application. EAServer searches the JAR files in the order specified by the com.sybase.jaguar.webapplication.jarlist property. If you do not JAR files in this property, EAServer searches in directory order; that is, the order that would be returned in a directory listing.

4 JAR files in *extensions*

   This directory contains JAR files that have been installed as Web application extensions—see "Using Java extensions" on page 399. EAServer places these JAR files in the Web application's default custom class list, in directory order.

5 JAR files in the *java/classes* directory that are listed in the custom class list for the servlet, JSP, or Web application.

6 Class files in the *java/classes* and *html/classes* directory that are specified in the custom class list for the servlet, JSP, or Web application.

7 JAR files in the *java/lib* directory that are listed in the custom class list for the servlet, JSP, or Web application.

8 If the class has not been found, the server attempts to load the class using the system class loader, using the search order in the server class path—see "The system class loader" on page 555.

Custom class lists for servlets installed directly in the server

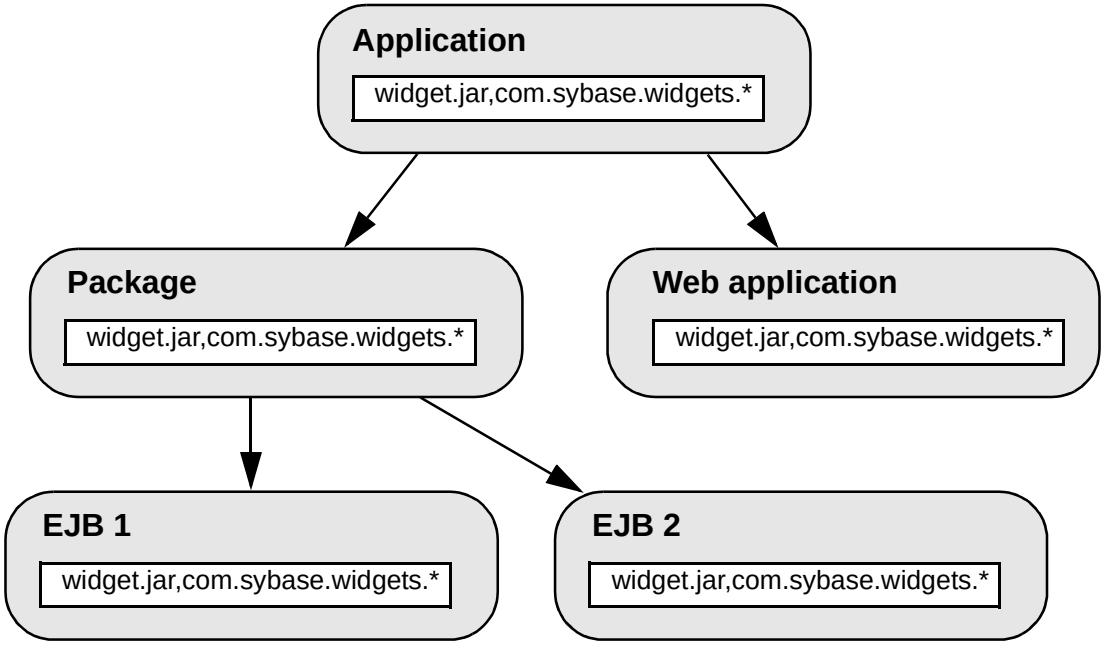
For servlets that are installed directly in a server, and not in a Web application, you must define the custom class list in the servlet properties or the properties of the host server. Classes and JAR files for these servlets should be deployed in the EAServer *java/classes* subdirectory.

Custom class lists for packages, applications, or servers

You can configure the package, application, or server level class lists to configure sharing of class instances between child entities installed in the package, application, or server. Doing so can decrease server memory use by eliminating copies of custom loaded classes. However, you must also refresh the classes at the level where they are loaded. For example, classes loaded at the server level require a refresh of the entire server.

To configure sharing of custom-loaded classes, first configure the class lists for the Web applications and components that use the class directly, then configure the same entries in the parent entries up to a common ancestor. Figure 30-2 shows the entries required to share classes in the JAR file *widget.jar* and the Java package *com.sybase.widgets*, loading one copy of the classes at the application level.

Figure 30-2: Example of sharing custom loaded classes



Deploy classes that are shared by multiple entities in one of these common locations:

- The *java/classes* subdirectory, for JAR files that are always custom-loaded (unless you have added the JAR file to the server CLASSPATH setting by listing it in the com.sybase.jaguar.server.jvm.classpath or com.sybase.jaguar.server.jvm.bootclasspath server property).
- The *java/lib* subdirectory, for JAR files that may be system loaded or custom loaded. If an entity's custom class list contains the JAR file name, the classes are custom loaded. Otherwise, the classes are system loaded, because EAServer adds JAR files in this location to the default server class path at server start-up time.
- The class tree rooted in the *java/classes* or *html/classes* subdirectory, for class files such as generated EJB stub classes. Use *java/classes* for server-side code. Use *html/classes* if the classes are shared with applet clients that load class files directly rather than using a JAR file.

To configure the server, application, or package class list, follow the instructions in "Configuring an entity's custom class list" on page 562.

Configuring an entity's custom class list

You can configure custom class lists for your servers, applications, Web applications, packages, and components, and so forth using the Java Classes tab in the EAServer Manager properties for the entity. If using jagtool or an EAServer XML configuration file, configure the class list by setting the entity property that ends in .java.classes, for example the application property name is com.sybase.jaguar.application.java.classes.

❖ **Configuring the custom class list for a server, application, Web application, component, or Web component in EAServer Manager**

1 Display the properties dialog for the server, application, Web application, component, package, or servlet.

2 Display the Java Classes tab.

3 Edit the class list as follows:

- To add a class, click Add and edit the name of the class, package, or JAR file, following the value syntax rules described below.
- To rearrange the order of listed items, highlight the item to be moved in the list and click Move Up or Move Down.
- To remove an item, click Remove.

Value syntax for Java class lists Enter a comma-separated list of Java classes, packages, and JAR files. 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
```

JAR files must be deployed in the EAServer *java/classes* subdirectory.

Troubleshooting class loader configuration issues

This section presents additional information that can be useful when diagnosing class loader configuration problems.

Commonly encountered problems

Common problems encountered in the custom class list configuration include:

- **Class cast exceptions** In Java, classes loaded by different class loaders are considered different types. You cannot assign a class loaded by one class loader to a reference loaded by another class loader. This restriction must be accounted for when specifying the custom class list, or when deciding at what level a class should be loaded at. Otherwise, the invocation may fail and you see one these Java exceptions in the server log file:
  - java.lang.ClassCastException
  - java.lang.LinkageError
  - java.lang.NoClassDefFoundError
  - java.lang.IncompatibleClassChangeError

  There are two variations of this issue:
  - When using EJB local interfaces, the calling entity and the caller must share the same instance of classes that are passed as method parameters or return values. In this case, fix the problem by copying the relevant custom class list entries to parent entities, up to a common ancestor. See “Custom class lists for packages, applications, or servers” on page 560 for details.
  - For other Java or EJB component calls, the entity that calls the component uses stubs that are system loaded. This call fails because stubs in the component are custom loaded, and Java considers classes that are loaded by different class loaders to be different types, even when the classes have the same name and deployment location. To fix this problem, add the called component’s stub classes to the custom class list for the component or Web application that makes the call.
- **Refreshing classes** Classes must be refreshed at the level they were loaded at. For example, if you configure an application class loader to share some class instances between components and Web applications, you must refresh the application to reload new versions of these classes. It will not do to refresh the components, package, or Web application directly.

Custom class loader tracing

To troubleshoot class loader problems, you can enable custom class loader tracing by setting the server property com.sybase.jaguar.server.classloader.debug to true using the Advanced tab in the Server Properties dialog box.

JAR file locking and copying

JAR files that are in the server's CLASSPATH setting are locked while in use by the system class loader. Consequently, on some platforms such as Windows, you cannot update or overwrite the JAR file while the server is running. To verify the server's CLASSPATH setting, connect to the server with EAServer Manager and check the value in the General tab of the Server Properties dialog box, or connect to the server with jagtool and check the value of the server property com.sybase.jaguar.server.jvm.classpath.

To allow refresh of JAR files that are custom loaded, each class loader instance works with a copy of the JAR files that it has loaded. Copies are created in subdirectories of the EAServer *work/server-name/tempjars* subdirectory, where *server-name* is the name of your server. EAServer deletes these directories and files when you restart the server.

CHAPTER 31

Using the Message Service

This chapter describes the messaging service feature for EAServer version 4.1, which supports the Java Messaging Service (JMS) 1.0.2 specification. You can use the message service to send messages to specific destinations and publish messages to be delivered to interested consumers. You can receive scheduled messages and message topic notifications, as well as asynchronously receive messages using message listeners.

To develop messaging service applications, you can use either the JMS API or the EAServer CORBA API. To ensure that your code is portable, Sybase recommends that you use the JMS API when performing nonadministrative tasks from Java clients and components. Use the EAServer CORBA API for your clients and components:

- For programming languages other than Java
- To automate administrative tasks
- To assign a single message listener to multiple message queues

For information on configuring and administering the message service, see Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide*.

Overview

The message service allows you to send or publish messages to a queue or topic, where they are stored until they can be delivered to either a client or a component. The message service supports two messaging models, point-to-point and publish/subscribe. Use point-to-point messaging to send a message to one consumer. Use the publish/subscribe model to publish messages that are available to all consumers who subscribe to the message topic. The message service provides transient and persistent message storage for message consumers.

A messaging service provides a highly flexible solution for many business needs. As a practical example, consider a business-to-business (B2B) scenario in which a supplier needs to communicate prices to a number of retailers, and the retailers may want to place orders for items at updated prices.

You would create a listener to respond to all incoming orders, and you would manage the list of retailers, as well as add new ones. You would ensure that each party received and processed all the transactions that are sent its way. If one of the retailers is offline, or network routing to its server fails, your application design must continue trying to establish communications until the transaction can be successfully sent. You would provide persistence for critical transactions until all recipients acknowledge them. You would also want to ensure that all parties are granted access and are who they say they are, and that transactions are secure during transmission.

Many B2B transactions take place in an environment such as this, where connectivity cannot be guaranteed and transactions require security. Inserting a messaging service between business nodes in a B2B network insulates your application code from these issues.

The key features of the EAServer message service include:

- High availability and load balancing
- Message security
- Reliable delivery
- Scalable notification
- Transaction management

High availability and load balancing

The message service uses server clustering to provide high availability and load balancing. For complete information about using server clustering, see Chapter 6, "Clusters and Synchronization," in the *EAServer System Administration Guide*.

Message security

The message service provides role-based security for message queues and message topics. The message service operations and the access categories required to use them are:

Access category	Message service operations
Consumer	Adding, removing, or acquiring access to listeners and message selectors, retrieving message queue and topic statistics
Producer	Sending and publishing messages
Security	Adding, configuring, and removing access roles

You need to assign permission for each access category separately.

Reliable delivery

To ensure reliable message delivery, the message service provides:

- IIOP/IIOPS connections for client notification.
- HTTP tunneling of IIOP connections. Messages can be delivered through client-side firewalls that accept only HTTP/HTTPS.
- Component notification that is usually performed in-process, which reduces the risk of partial failure.
- Persistent messages that guarantee message delivery, subject to the reliability of the persistent store.

Scalable notification

When a thread pool is used for client notification, the message queue object is implemented with a specialized server IIOP handler that uses only a few waiting threads to handle blocking receive calls, so it avoids waking large numbers of threads for client notification.

Transaction management

A transactional operation runs in the caller's transaction, or if the caller is not enlisted in a transaction, in a new transaction. These operations can be transactional:

- **publish** A message producer can publish a message within a transaction.
- **send** A message producer can send a message within a transaction.
- **acknowledge** A client can acknowledge and process a message in the same transaction.
- **onMessage** A listener component can process a message within the same transaction as the automatic acknowledgment, which occurs when onMessage returns.
- **move** A message can be moved from one queue to another only within a transaction.

Developing JMS applications

You can create a JMS application using either the point-to-point or the publish/subscribe messaging model. Both models support applications that are capable of:

- Creating a JMS InitialContext object
- Looking up a ConnectionFactory object
- Creating permanent destinations
- Creating connections
- Creating sessions
- Creating message producers

- Creating message consumers
- Implementing and installing message listeners
- Creating messages
- Sending messages
- Publishing messages
- Receiving messages
- Browsing messages
- Enabling JMS tracing
- Closing connections, sessions, consumers, and producers

You can download a copy of the JMS 1.0.2 API documentation from the Java Web site at http://java.sun.com/products/jms/index.html.

Creating a JMS InitialContext object

A JMS client application must instantiate a JMS InitialContext object and set these properties:

- InitialContext factory – set `java.naming.factory.initial` to `"com.sybase.jms.InitialContextFactory"`.
- URL – set `java.naming.provider.url` to the location where the client can connect to EAServer.
- User name – valid for a connection with EAServer.
- Password – valid for the user name.

When instantiating a JMS InitialContext from a base client, specify the user name with the SECURITY_PRINCIPAL property, and specify the password with the SECURITY_CREDENTIALS property. The default for both properties is an empty string.

This example runs the JMS client application JMSClientClass and sets the InitialContext factory, URL, user name, and password properties at runtime:

```
java -D java.naming.factory.initial=com.sybase.jms.InitialContextFactory
-Djava.naming.provider.url=iiop://stack:9000
-DSECURITY_PRINCIPAL="jagadmin" -DSECURITY_CREDENTIALS="sybase"
JMSClientClass
```

Looking up a ConnectionFactory object

A connection factory allows you to create JMS connections and specify a set of configuration properties that define the connections. EAServer provides two types of connection factories, one to create queue connections and one to create topic connections. Queue connections allow you to send and receive messages using the PTP messaging model. Topic connections allow you to publish and receive messages using the Pub/Sub messaging model. EAServer provides two preconfigured connection factories that you can use javax.jms.QueueConnectionFactory and javax.jms.TopicConnectionFactory. To create connection factories, use EAServer Manager—see Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide* for details. Once you have created a connection factory, client applications must look up a ConnectionFactory object using JNDI, as in this example:

```
// Look up a queue connection factory

QueueConnectionFactory queueCF =
 (QueueConnectionFactory) ctx.lookup("MyTestQueueCF");

// Look up a topic connection factory

TopicConnectionFactory topicCF =
 (TopicConnectionFactory) ctx.lookup("MyTestTopicCF");
```

If the connection factory cannot be found, EAServer throws a javax.naming.NamingException.

Creating permanent destinations

Permanent destinations are message queues or topics whose configuration properties are stored in a database. You can create permanent destinations two ways. The recommended way is to use EAServer Manager to create and configure message queues and topics. See Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide* for information on how to do this. You can also create permanent destinations using the JMS APIs createQueue and createTopic. Sybase suggests that you use this option only when client applications need the ability to dynamically change the provider-specific destination name; applications using this technique are not portable.

When you create message queues and topics using EAServer Manager, client applications can use their InitialContext object to look up the destinations; for example:

```
// Look up a message queue

javax.jms.Queue queue =
   (javax.jms.Queue) ctx.lookup("MyQueue");

// Look up a topic

javax.jms.Topic topic =
   (javax.jms.Topic) ctx.lookup("MyTopic");
```

To create permanent destinations using the JMS APIs, you must first create a session; see "Creating sessions" on page 573 for details. Once you have access to a session object, use this syntax to create a destination:

```
javax.jms.Queue queue =
   queueSession.createQueue("MyQueue");

javax.jms.Topic topic =
   topicSession.createTopic("MyTopic");
```

Temporary destinations

You can also create temporary destination objects that are active only during the lifetime of a session. When the session is closed, temporary destination objects and their associated messages are discarded. These two lines illustrate how to create temporary message queue and topic destinations:

```
// Create a temporary queue

javax.jms.Queue queue =
   queueSession.createTemporaryQueue();

// Create a temporary topic

javax.jms.Topic topic =
   topicSession.createTemporaryTopic();
```

By default, temporary message queues time out after 60 seconds of inactivity. To increase this value, you can do one of two things:

- Set the connection factory's CONFIG_QUEUE property to the name of a message queue with a reasonably high timeout value. Subsequently, each temporary message queue you create that uses this connection factory inherits the properties of the queue assigned to CONFIG_QUEUE.
- Set the value at the global level so all temporary message queues use the same timeout value. To do this, edit the *MessageServiceConfig.props* file, located in the EAServer */Repository/Component/CtsComponents* directory, and set the `session.timeout` property to the appropriate number of seconds.

Chapter 8, “Setting up the Message Service,” in the *EAServer System Administration Guide* describes these settings in detail.

Creating connections

JMS provides two types of connections, one for message queues and one for topics. To create a connection from a JMS client application to EAServer, you must have access to a ConnectionFactory object. See “Looking up a ConnectionFactory object” on page 570. Once you have created a connection, you must explicitly start it before EAServer can deliver messages on the connection. To avoid message delivery before a client has finished setting up, you may want to delay starting the connection. This code fragment creates a QueueConnection object and starts the connection:

```
QueueConnection queueConn =
   queueCF.createQueueConnection();

// other setup procedures

queueConn.start();
```

This sample creates a connection object for a topic and starts the connection:

```
TopicConnection topicConn =
   topicCF.createTopicConnection();

// other setup procedures

topicConn.start();
```

You can also stop delivery of messages using the QueueConnection.stop method, then use start to resume delivery. While a connection is stopped, receive calls do not return with a message, and messages are not delivered to message listeners. Any calls to receive or message listeners that are in progress when stop is called, complete before the stop method returns.

With a single connection to EAServer, the message service can send and receive multiple messages. Therefore, a JMS client usually only needs one connection.

Setting the client ID

A connection for a durable topic subscriber must have a client ID associated with it so that EAServer can uniquely identify a client if it disconnects and later reconnects. You can use EAServer Manager to set the client ID when you create the topic connection factory; see Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide*. If you do not set the client ID in the connection factory, you must set it immediately after creating the connection and before performing any other action on the connection. After this point, attempting to set the client ID throws an IllegalStateException. This code fragment illustrates how to set a topic connection's client ID:

```
topicConn.setClientID("Client ID String");
```

For more information about topic subscribers, see "Creating message consumers" on page 575.

ExceptionListener

To enable EAServer to asynchronously notify clients of serious connection problems, create and register an ExceptionListener. The javax.jms.ExceptionListener must implement this method:

```
void onException(JMSException exception);
```

To register a listener, call the Connection::setExceptionListener method, for example:

```
queueConn.setExceptionListener(MyExceptionListener);
```

If an exception occurs and a listener has been registered, EAServer calls the onException method and passes the JMSException, which describes the problem.

Creating sessions

Once a client has established a connection with EAServer, it needs to create one or more sessions. A session serves as a factory for creating message producers, message consumers, and temporary destinations. JMS provides two types of session objects, one for queue connections and one for topic connections. To create a QueueSession object, use a previously created QueueConnection object as follows:

```
QueueSession queueSession =
  queueConn.createQueueSession(true,
                               Session.AUTO_ACKNOWLEDGE);
```

To create a TopicSession object, use a previously created TopicConnection object as follows:

```
TopicSession topicSession =
```

```
topicConn.createTopicSession(true,
                                Session.AUTO_ACKNOWLEDGE);
```

When you create a session, set the first parameter to true if you want a **transacted session**. When you publish or send a message in a transacted session, the transaction begins automatically. Once a transacted session starts, all messages published or sent in the session become part of the transaction until the transaction is committed or rolled back. When a transaction is committed, all published or sent messages are delivered. If a transaction is rolled back, any messages produced in the session are destroyed, and any consumed messages are recovered. When a transacted session is committed or rolled back, the current transaction ends and the next transaction begins. See Chapter 2, "Understanding Transactions and Component Lifecycles" for more information about transactions.

Set the first parameter to false when you do not want a transacted session. If a client has an active transaction context, it can still determine transactional behavior, even if it does not create a transacted session.

The second parameter indicates whether the message producer or the consumer will acknowledge messages. This parameter is valid only for nontransacted sessions. In transacted sessions, acknowledgment is determined by the outcome of the transaction.

Acknowledgment mode	Description
AUTO_ACKNOWLEDGE	The session automatically acknowledges messages.
CLIENT_ACKNOWLEDGE	The client explicitly acknowledges a message, which automatically acknowledges all messages delivered in the session.
DUPS_OK_ACKNOWLEDGE	EAServer implements this the same as AUTO_ACKNOWLEDGE.

Session::recover

To stop message delivery within a session and redeliver all the unacknowledged messages, you can use the Session.recover method. When you call recover for a session, the message service:

- Stops message delivery within the session.
- Marks all unacknowledged messages "redelivered", including those that have been delivered.
- Restarts sending all unacknowledged messages, beginning with the oldest message.

The method can be called only by a non-transacted session; it throws an IllegalStateException if it is called by a transacted session.

Creating message producers

Create message producers for sending and publishing messages. JMS defines two message producer objects, a QueueSender, used to send messages, and a TopicPublisher, used to publish messages. To create a QueueSender object, use a previously created QueueSession object and this syntax:

```
javax.jms.QueueSender sender = queueSession.createSender(queue);
```

To create a TopicPublisher object, use a previously created TopicSession object and this syntax:

```
javax.jms.TopicPublisher publisher = topicSession.createPublisher(topic)
```

Creating message consumers

Message consumers can be either QueueReceiver or TopicSubscriber objects. Create a QueueReceiver to retrieve messages that are sent using the PTP messaging model. Use previously created Queue and QueueSession objects, as follows:

```
javax.jms.QueueReceiver receiver =
  queueSession.createReceiver(queue);
```

A TopicSubscriber object receives published messages and can be either durable or nondurable. A nondurable subscriber can only receive published messages while it is connected to EAServer. If you want guaranteed message delivery, make the subscriber durable. For example, if you create a durable subscription on a topic, EAServer saves all published messages for the topic in a database. If a durable subscriber is temporarily disconnected from EAServer, its messages will be delivered when the subscriber is reconnected. The messages are deleted from the database only after they are delivered to the durable subscriber.

This example illustrates how to create both durable and nondurable topic subscribers. In both cases, you need to reference previously created Topic and TopicSession objects:

```
// Create a durable subscriber

javax.jms.TopicSubscriber subscriber =
   topicSession.createDurableSubscriber(topic,
                                        "subscriptionName")

// Create a non-durable subscriber

javax.jms.TopicScuscriber subscriber =
```

```
   topicSession.createSubscriber(topic);
```

To remove a durable topic subscription, call the TopicSession.unsubscribe method, and pass in the subscription name; for example:

```
topicSession.unsubscribe("subscriptionName");
```

Filtering messages using selectors

You can use selectors to specify which messages you want delivered to a message queue. Once you add a selector to a queue, the message service delivers only those messages whose message topic matches the selector. You can create message selectors using EAServer Manager—see Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide*. You can also create message selectors programmatically. This example illustrates how to create a message selector and use it when you are creating a new QueueReceiver:

```
// Create a selector to receive only Text messages
// whose value property equals 100.

String selector = "value = 100 and Type = TextMessage";

// Create a queue receiver using the selector.

QueueReceiver receiver =
   queueSession.createReceiver(queue, selector);
```

This code sample sends two messages to the message queue we just created. The properties of the first message match those of the message queue's selector. The properties of the second message do not.

```
// Create and send a message whose properties match
// the message queue selector.

TextMessage textMsg =
   queueSession.createTextMessage("Text Message");
textMsg.setIntProperty("Value", 100);
textMsg.setStringProperty("Type", "TextMessage");
sender.send(textMsg);

// Create and send a Bytes message, whose value
// property equals 200.

BytesMessage bytesMsg =
   queueSession.createBytesMessage();
bytesMsg.setIntProperty("Value", 200);
```

```
bytesMsg.setStringProperty("Type", "BytesMessage");
sender.send(bytesMsg);
```

When we retrieve messages from the message queue, the Text message will be returned but the Bytes message will not.

Implementing and installing message listeners

Message listeners allow you to receive messages asynchronously. Once you have implemented a listener, install it on a message consumer. When a message is delivered to the message consumer, the listener can send the message to other consumers or notify one or more components.

JMS message listeners implement the javax.jms.MessageLintener interface and can be either client-side listener objects or EJB 2.0 message-driven beans (MDB). The MessageListener interface contains only the onMessage method. This example illustrates the skeleton code for a message listener:

```
class QueueMessageListener implements MessageListener
{
   public void onMessage(javax.jms.Message msg)
   {
      // process message, notify component
   }
}
```

You can use EAServer Manager to install a message listener on a message queue or topic—see “Installing and configuring an MDB” on page 578. You can also install a message listener within your application. First create a message consumer, see “Creating message consumers” on page 575, then install the listener, using this syntax:

```
receiver.setMessageListener(new QueueMessageListener());
```

Message-driven beans

An MDB is a type of Enterprise JavaBean (EJB) specifically designed as a JMS message consumer. You can install an MDB as a message listener on a message queue or topic.

When an MDB is installed as a listener on a message consumer and a JMS message arrives, EAServer instantiates the MDB to process the message. An MDB must implement the MessageDrivenBean interface, which consists of these methods:

Method name	Description
ejbCreate	Creates an instance of an MDB.
setMessageDrivenContext	Associates an MDB instance with its context, which EAServer maintains. This provides the MDB instance access to the MessageDrivenContext interface.
ejbRemove	Notifies the MDB instance that it is being removed and should release its resources.

An MDB instance with container-managed transactions can call these MessageDrivenContext interface methods:

Method name	Description
setRollbackOnly	To specify that the current transaction must be rolled back.
getRollbackOnly	To test whether the current transaction has been marked to roll back.
getUserTransaction	Returns the javax.transaction.UserTransaction interface, with which the MDB can set and obtain transaction status.

Unlike other EJBs, message-driven Beans do not have a home or remote interface, and clients cannot directly access an MDB. EAServer calls the onMessage method of the javax.jms.MessageListener interface to notify an MDB that a message has been delivered to the queue or topic on which it is installed. To prevent client access to the onMessage method, this component property is set automatically when you install and configure an MDB:

```
com.sybase.jaguar.component.roles = onMessage(security-roles=ServiceControl)
```

❖ **Installing and configuring an MDB**

1 Create a new EJB component, as described in Chapter 7, “Creating Enterprise JavaBeans Components.”

Note You do not need to define remote or local interfaces.

2 On the General tab, supply these values:

- Description – summarize the bean’s purpose.
- Component Type – select EJB - Message Driven Bean from the drop-down list.
- EJB Version – choose 2.0.
- MDB Class – enter the name of the Java class that implements the MDB; for example com.sybase.jaguar.myPkg.MyBeanImpl.

3 On the MDB Type tab, enter:

- Destination Type – choose either Queue or Topic.
- Name – enter the name of the destination queue or topic.
- Listener – enter the package and component name for the MDB listener; for example, `MyPkg/MyBeanImpl`. To specify a thread pool, append the thread pool name in square brackets, for example, `MyPkg/MyBeanImpl[MyThreadPool]`.

  You can create thread pools in EAServer Manager as described in Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide*. The thread pool must have one or more worker threads. A thread pool with multiple worker threads enables the message listener to process multiple messages at the same time. If you do not specify the name of a thread pool, the message listener uses the <system> default thread pool, which has a single worker thread.
- Acknowledge Mode – choose either Auto or Dups-ok. For an explanation of the acknowledgment modes, see "Creating sessions" on page 573.
- Message Selector – if the destination type is a queue, enter a message selector to filter incoming messages; for example, to receive all published messages with the topic "StockPrice.SY", enter:

  ```
  topic = 'StockPrice.SY'
  ```
- Subscription/Durability – if the destination type is a topic, select either Durable or Non-durable. For a description of these options, see "Creating message consumers" on page 575.

4 On the Transactions tab:

- Select one of Not Supported, Required, or Bean Managed.
- Optionally, select Automatic Failover.

"Component properties: Transactions" on page 58 describes the options on this tab.

5 On the Run-As Mode tab, define the identity properties used for intercomponent calls. "Component properties: Run-As Mode" on page 68 describes this tab in detail. If you do not specify a Run As Mode, the default value for an MDB is "System."

6 Optionally specify a retry timeout. If your MDB throws an exception while processing a message, EAServer can retry delivery for the time period you specify. By default, EAServer does not retry delivery. To configure this setting:

   a Enable the Automatic Failover option on the Transactions tab.

   b On the Advanced tab, set the property com.sybase.jaguar.component.retry.timeout to the number of seconds that EAServer should retry delivery. EAServer retries at intervals that increase by about one second after each failure; that is, after one second, again after 3 seconds, again after 6 seconds, and so forth.

7 Similar to other EJBs, you can enter information on these tabs, but it is not required. For more information, see:

Tab	Description link
Instances	"Component properties: Instances" on page 61
Resources	"Component properties: Resources" on page 65
EJB References	"EJB references" on page 383
EJB Local References	"EJB local references" on page 384
Environment	"Environment properties" on page 387
Resource References	"Resource references" on page 385
Resource Environment References	"Resource environment references" on page 386
Role References	"Configuring role references and method permissions" on page 133
Java Classes	Chapter 30, "Configuring Custom Java Class Lists"
Additional Files	"Component properties: Additional Files" on page 69

Running the sample JMS client and MDB

EAServer includes a sample JMS client and MDB listener. For more information, see "Using message-driven beans and JMS" in Chapter 5, "Using the EAServer Samples," in the *EAServer Cookbook*.

Related chapters

For information about Enterprise JavaBeans, see these chapters:

- Chapter 6, "Enterprise JavaBeans Overview"
- Chapter 7, "Creating Enterprise JavaBeans Components"

Creating messages

To create a message, you must first create an instance of either a queue or topic session. See “Creating sessions” on page 573 for details. To create a text message, use this syntax:

```
// Create a text message using a QueueSession

javax.jms.TextMessage queueTextMsg =
   queueSession.createTextMessage(“Text message”);

// Create a text message using a TopicSession

javax.jms.TextMessage topicTextMsg =
   topicSession.createTextMessage(“Text message”);
```

EAServer supports six message types that a message producer can send or publish. Table 31-1 describes the message types and the javax.jms.Session interface APIs used to create instances of each.

Table 31-1: JMS message types

Message type	Create message API	Comments
Plain	Session.createMessage	Creates a message without a message body.
Text	Session.createTextMessage	Creates a message that can contain a string in the message body.
Object	Session.createObjectMessage	Creates a message that can contain a serializable Java object in the message body.
Stream	Session.createStreamMessage	Creates a message that can contain a stream of Java primitives in the message body. Fill and read the message sequentially.
Map	Session.createMapMessage	Creates a message whose body can contain a set of name-value pairs where names are Strings and values are Java primitive types.
Bytes	Session.createBytesMessage	Creates a message that can contain a stream of uninterpreted bytes in the message body.

To improve interoperability with non-Java clients or components and to improve message receivers' ability to filter messages, Sybase recommends that you use either plain messages or text messages. Selectors allow you to filter messages based on the text in the message properties. You cannot filter messages based on the text in the message body. Therefore, message text in the message properties, instead of the message body, enables the message receivers to filter messages more efficiently.

For more information about the message types, see the JMS API documentation at http://java.sun.com/products/jms/javadoc-102a/index.html.

Sending messages

To send a message, you must specify the destination message queue. The message service notifies listeners that are registered for the queue and the message remains in the queue until it is received and acknowledged.

Figure 31-1: Send message flow

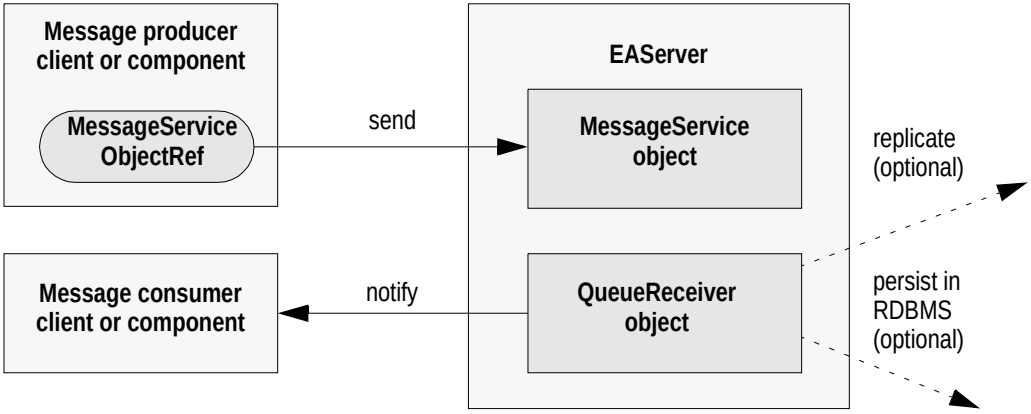


Figure 31-1illustrates the message flow that occurs when a client or component sends a message.

This example notifies a client of a completed order; it creates a new message, constructs the message text, and sends the message to the client's queue:

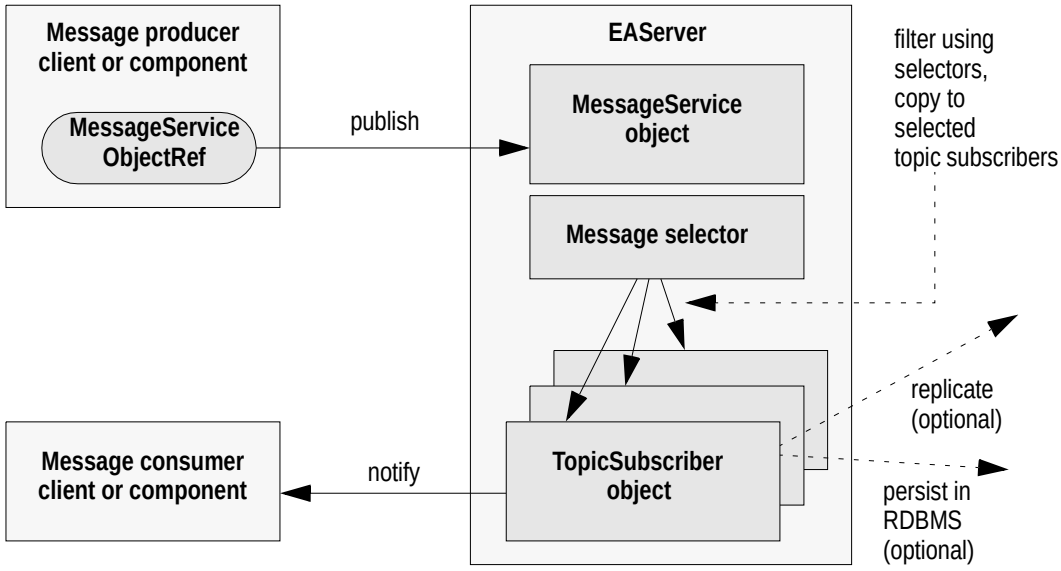
```
public void notifyOrder(QueueSession qSession,
                        Queue queue,
                        int orderNo,
                        String product)
```

```
{
   String time = new java.util.Date().toString();
   String text = "Order " + orderNo + " for product " + product +
      " was completed at " + time;

   javax.jms.QueueSender sender = qSession.createSender(queue);
   javax.jms.TextMessage textMsg = qSession.createTextMessage(text);

   textMsg.setStringProperty("ProductDescription", product);
   textMsg.setIntProperty("OrderNumber", orderNo);

   sender.send(textMsg);
}
```

Publishing messages

When you publish a message, a copy is sent to all topic subscribers that have a message selector registered with the specified topic. Figure 31-2 illustrates the message flow when a client or component publishes a message.

Figure 31-2: Publish message flow



This example publishes a message that notifies clients of changes in a stock value; it creates the message text, creates a TopicPublisher and the message using the TopicSession object, and publishes the message:

```
public void notifyStockValue(TopicSession tSession,
                             Topic topic,
                             String stock,
                             double value)
{
   String time = new java.util.Date().toString();
   String text = time + ": The stock " + stock + " has value " + value;

   // Create the publisher and message objects.

   javax.jms.TopicPublisher publisher = tSession.createPublisher(topic);
   javax.jms.TextMessage textMsg = tSession.createTextMessage(text);


   // Publish a non-persistent message with a priority of 9 and a
   // lifetime of 5000 milliseconds (5 seconds)

   publisher.publish(textMsg, DeliveryMode.NON_PERSISTENT, 9, 5000);
}
```

To publish a persistent message using the default priority (4) and timeout (never expires) values, use this syntax:

```
publisher.publish(textMsg);
```

Receiving messages

You can receive messages either synchronously or asynchronously. To receive messages synchronously (get all the messages at one time), call the receive method for the message consumer. The following code samples illustrate how to receive all the messages from a queue, using three different timeout options:

```
// Get all the messages from the queue. If none exists,
// wait until a message arrives.

javax.jms.TextMessage queueTextMsg =
   (javax.jms.TextMessage) receiver.receive();

// Get all the messages from the queue. If none exists,
// wait 5000 milliseconds (5 seconds) or until a message
// arrives, whichever comes first.
```

```
javax.jms.TextMessage queueTextMsg =
   (javax.jms.TextMessage) receiver.receive(5000);

// Get all the messages from the queue. If none exists,
// return NULL.

javax.jms.TextMessage queueTextMsg =
   (javax.jms.TextMessage) receiver.receiveNoWait();
```

To receive messages asynchronously, implement a message listener and install it on the message consumer, either a topic or a queue. See “Implementing and installing message listeners” on page 577.

For information about creating message queues and topics, see “Creating message consumers” on page 575.

Browsing messages

You can look at messages in a queue without removing them using the QueueBrowser interface. You can browse through all the messages in a queue, or through a subset of the messages. To browse through a queue’s messages, create an instance of a QueueBrowser object using a previously created QueueSession object. To create a browser for viewing all the messages in a queue, call createBrowser and pass the message queue:

```
javax.jms.QueueBrowser qbrowser =
  queueSession.createBrowser(queue);
```

To create a browser for viewing a subset of the messages in a queue, call createBrowser and pass the message queue and a message selector string:

```
javax.jms.QueueBrowser qbrowser =
  queueSession.createBrowser(queue, selector);
```

For information about message selectors, see “Filtering messages using selectors” on page 576.

Once you have access to the QueueBrowser object, call getEnumeration, which returns an Enumeration that allows you to view the messages in the order that they would be received:

```
java.util.Enumeration enum = qbrowser.getEnumeration();
```

Enabling JMS tracing

To help debug your JMS client application, you can enable tracing by setting the `com.sybase.jms.debug` property to true in the InitialContext object. When you enable tracing, diagnostic messages are printed in the console window. By default, tracing is disabled. This code sample illustrates how to set the tracing property:

```
Properties prop = new Properties();

prop.put("com.sybase.jms.debug", "true");
javax.naming.Context ctx =
   new javax.naming.InitialContext(prop);
```

Closing connections, sessions, consumers, and producers

The JMS server allocates resources for each of these objects: connections, sessions, message consumers, and message producers. When you no longer need one of these objects, you should close it to release its resources and help the application run more efficiently. To release each object's resources, EAServer provides these methods:

- QueueConnection.close
- TopicConnection.close
- QueueSession.close
- TopicSession.close
- QueueReceiver.close
- TopicSubscriber.close
- QueueSender.close
- TopicPublisher.close

JMS interfaces not supported

EAServer does not support these JMS interface methods:

JMS interface	Method
javax.jms.Session	• run • setMessageListener • getMessageListener
javax.jms.QueueConnection	• createConnectionConsumer
javax.jms.TopicConnection	• createConnectionConsumer • createDurableConnectionConsumer

In addition, EAServer does not support these JMS interfaces:

- javax.jms.XAQueueConnection
- javax.jms.XATopicConnection
- javax.jms.XAQueueConnectionFactory
- javax.jms.XATopicConnectionFactory
- javax.jms.XASession
- javax.jms.XAQueueSession
- javax.jms.XATopicSession
- javax.jms.XAConnection
- javax.jms.XAConnectionFactory
- javax.jms.ServerSession
- javax.jms.ServerSessionPool
- javax.jms.ConnectionConsumer

Note EAServer supports the XA interfaces that are required to support two-phase commit and the XA transaction coordinator for JMS clients and components.

Developing EAServer messaging service applications

To develop an EAServer messaging service application, use the EAServer CORBA APIs, which enable you to configure and use the message service within a client application or EAServer component. See “EAServer message service CORBA API” on page 595 for more information. Creating message service applications can involve these steps:

- Obtaining CtsComponents::MessageService object references
- Creating message consumers
- Creating message selectors
- Creating thread pools programmatically
- Implementing and installing message listeners
- Sending messages
- Publishing messages
- Receiving messages
- Subscribing to scheduled messages

Obtaining CtsComponents::MessageService object references

Before a CORBA client can send, publish, or receive messages, it must obtain a MessageService object reference. This code sample performs the setup required for a message service client application:

```
org.omg.CORBA.*;
import java.util.*;
import SessionManager.*;
import CtsComponents.*;
import java.lang.Object;

public class ReceiveTest
{
   public static void main(String[] args)
   {
      new ReceiveTest().test(args);
   }

   public void test(String[] args)
   {
      Properties props = new Properties();
```

```
            props.put("org.omg.CORBA.ORBClass",
                      "com.sybase.CORBA.ORB");

            ORB orb = ORB.init((String[])null, props);

            Manager manager =
               ManagerHelper.narrow(orb.string_to_object(
                                    "iiop://localhost:9000"));

            Session session =
               manager.createSession("jagadmin", "");

            MessageService cms =
               MessageServiceHelper.narrow(session1.create(
                           "CtsComponents/MessageService"));

            MessageQueue mq =
               cms.getMessageQueue("test", "",
                                   REQUIRES_ACKNOWLEDGE.value);
...
```

Creating message consumers

Message consumers can be either a message queue or topic. You can create message consumers two ways. The recommended way is to use EAServer Manager to create and configure message queues and topics, so their configuration properties are stored in a database. See Chapter 8, “Setting up the Message Service,” in the *EAServer System Administration Guide* for information on how to do this. You can also create message queues and topics programmatically; for example:

```
void create(MESSAGE_QUEUE, “QueueName”);
void create(MESSAGE_TOPIC, “TopicName”);
```

See *$JAGUAR/html/ir/CtsComponents__MessageService.html* for more information on configuring queues and topics within your application.

Creating message selectors

You can use selectors to specify which messages you want delivered to a message queue. Once you add a selector to a queue, the message service delivers only those messages whose message topic matches the selector. You can create message selectors using EAServer Manager—see Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide*. You can also create message selectors programmatically. This example illustrates how to add a message selector to MyQueue to request notification of a new stock value:

```
MessageService cms = getMessageService();
cms.addSelector("MyQueue",
   "topic = 'stock.SY' AND value > 50");
```

Creating thread pools programmatically

The threads in a thread pool provide asynchronous client and component notification. You can create thread pools using EAServer Manager—see Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide*. You can also create a thread pool within your application. This example creates a thread pool and sets the thread values of readers, writers, and workers to "0". If the thread pool already exists, its configuration is unchanged.

void create (THREAD_POOL, "*Thread_Pool_Name*");

To use a thread pool for client notification, set the thread values of readers to "3", writers to "2", and workers to "0". To use a thread pool for component notification, set the thread values of both readers and writers to "0", and set the value of workers to "1". If you want to enable parallel message processing for component notification, set workers to a value greater than "1".

This code fragment sets the value of *workers* in the system.tp1 thread pool to "1":

```
props = _cms.getProperties(THREAD_POOL.value,
                           "system.tp1");

for (int i = 0; i < props.length; i++)
{
   Property p = props[i];
   if (p.name.equals("workers"))
   {
      if (p.value.longValue() != 1)
```

```
            {
               p.value.longValue(1);
               _cms.setProperties(THREAD_POOL.value,
                                  "system.tp1", props);
               break;
            }
         }
      }
```

Implementing and installing message listeners

Message listeners allow you to receive messages asynchronously. Once you have implemented a listener, install it on a message consumer. When a message is delivered to the message consumer, the listener can send the message to other consumers or notify one or more components.

Message listeners are EAServer components that implement the CtsComponents::MessageListener interface, which contains only this onMessage method:

```
void onMessage(in CtsComponents::Message msg);
```

You can install message listeners two ways. The recommended way is to use EAServer Manager to install a message listener on a message queue or topic—see Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide*. You can also install a message listener within your application.

This example installs a message listener implemented by the EAServer component "MyPackage/MailService" on the message queue "MyClient:email":

```
MessageService cms = getMessageService();
cms.addListener("MyClient:email",
                "MyPackage/MailService");
```

When you create a message listener, you can optionally provide the name of a thread pool. The thread pool must have one or more worker threads. A thread pool with multiple worker threads enables the message listener to process multiple messages at the same time. If you do not specify the name of a thread pool, the message listener uses the system default thread pool, which has a single worker thread.

This code sample adds a message listener and specifies "MailPool" as the name of the thread pool:

```
cms.addListener("MyClient:email",
                "MyPackage/MailService[MailPool]");
```

Sending messages

To send a message, you must specify the destination message queue. The message service notifies listeners that are registered for the queue and the message remains in the queue until it is received and acknowledged.

Figure 31-1 on page 582 illustrates the message flow that occurs when a client or component sends a message.

In this example, we notify a client of a completed order by creating a new message, constructing the message text, and sending the message to the client's queue:

```
public void notifyOrder(MessageService cms,
                        String queue,
                        int orderNo,
                        String product)
{
  String time = new java.util.Date().toString();
  String text = "Order " + orderNo + " for product "
                + product + " was completed at " + time;

  Message msg = new Message();
  msg.key = cms.getMessageKey();
  msg.props = new Property[2];
  msg.props[0] = new Property("orderNo",
                              new PropertyValue());
  msg.props[0].value.longValue(orderNo);
  msg.props[1] = new Property("product",
                              new PropertyValue());
  msg.props[1].value.stringValue(product);
  msg.replyTo = "";
  msg.text = text;
  cms.send(queue, msg, PERSISTENT.value);
}
```

Publishing messages

When you publish a message, a copy is sent to all topic subscribers that have a message selector registered with the specified topic. Figure 31-2 on page 583 illustrates the message flow when a client or component publishes a message.

This example illustrates how to publish a message that notifies clients of changes in a stock value. Set the message topic, define the message text, set the message key, define and set message properties, and call publish:

```
public void notifyStockValue(MessageService cms,
                             String stock,
                             double value)
{
   String topic = "StockValue." + stock;
   String time = new java.util.Date().toString();
   String text = time + ": The stock " + stock +
                        " has value " + value;

   Message msg = new Message();
   msg.key = cms.getMessageKey();
   msg.props = new Property[2];
   msg.props[0] = new Property("stock",
                               new PropertyValue());
   msg.props[0].value.stringValue(stock);
   msg.props[1] = new Property("value",
                               new PropertyValue());
   msg.props[1].value.doubleValue(value);
   msg.replyTo = "";
   msg.text = text;
   cms.publish(topic, msg, 0);
}
```

To publish a persistent message using the default priority (4) and timeout (never expires) values, use this syntax:

```
publisher.publish(textMsg);
```

Receiving messages

You can receive messages either synchronously or asynchronously. To receive messages synchronously (get all the messages at one time), call the receive method for the message consumer. This code sample gets all the messages from the queue, then, for each message, it prints a message receipt and acknowledges the message:

```
Message[] seq = mq.receive(0, DEFAULT_TIMEOUT.value);

for (int m = 0; m < seq.lengeth; m++;)
{
   Message msg = seq[m];
   System.out.println("Received message: " + msg.text);
   mq.acknowledge(msg.key);
}
```

To receive messages asynchronously, implement a message listener and install it on the message consumer, either a topic or a queue. See "Implementing and installing message listeners" on page 591.

Subscribing to scheduled messages

The message service can generate and send regularly scheduled messages to message queues. You can subscribe to scheduled messages by installing a listener on a queue and subscribing to a topic that defines the times you want to be notified. In this example, we add a listener to "MyQueue" and subscribe to the topic "second:30", which causes the message service to send a message to MyQueue at 30 seconds past each minute:

```
cms.addListener("MyQueue", "MyPackage/MyComp");
cms.addSelector("MyQueue", "topic = '<second:30>'");
```

To request a message be sent to MyQueue at 15 and 45 minutes past each hour, use this syntax:

```
cms.addSelector("MyQueue", "topic = '<minute:15>'");
cms.addSelector("MyQueue", "topic = '<minute:45>'");
```

For information on how to add selectors using EAServer Manager, see Chapter 8, "Setting up the Message Service," in the *EAServer System Administration Guide*.

Scheduled messages are delivered to queues with the appropriate selectors within milliseconds of the specified time. The time at which a component receives a message from the queue, however, depends on the number of unprocessed messages in the queue.

Scheduling variables

Scheduled message topic names can be either *'<minute:NN>'* or *'<second:NN>'*. Additional constraints must include a variable name and value. You can use these variables to define the message topic subscriptions:

Variable	Definition
MINUTE	Number of minutes past the hour

Variable	Definition
SECOND	Number of seconds past the minute
HOUR_OF_DAY	To specify 5 PM, “HOUR_OF_DAY = 17”
HOUR	Twice a day, at 6 AM and 6 PM, “HOUR = 6”
YEAR	Four-digit year, for example, 2000
MONTH	The name of the month, for example, January, February, and so forth
DATE	Date of the month, 1-31
DAY_OF_MONTH	Same as DATE
DAY_OF_WEEK	The name of the day of the week, for example, Monday, Tuesday, and so forth.
DAY_OF_WEEK_IN_MONTH	To specify the first Sunday in October, MONTH = October and DAY_OF_WEEK = Sunday and DAY_OF_WEEK_IN_MONTH = 1
DAY_OF_YEAR	To specify February 1, “DAY_OF_YEAR = 32”
WEEK_OF_MONTH	The week number within the current month
WEEK_OF_YEAR	The week number within the current year

The variable names are not case sensitive; minute and MINUTE are equivalent. You can find documentation for the variables, whose names correspond to the constants in the Java class java.util.Calendar, in the Java API Specification at http://java.sun.com/products//jdk/1.2/docs/api/java/util/Calendar.html.

Scheduled messages are not saved to persistent storage and they are not replicated.

A scheduled message includes two properties that indicate the message creation time, which can be accessed by the component:

Property name	Datatype	Format
@t	double	Number of milliseconds since 1 January 1970
ts	string	YYYY-MM-DD HH:MM:SS

You can find the message structure definition in *$JAGUAR/html/ir/CtsComponents.html*.

EAServer message service CORBA API

The EAServer CORBA API includes:

- **MessageService** The CtsComponents::MessageService interface allows EAServer clients and components to send and publish messages, register for message topic notification, and manage message queue and message topic properties. See *$JAGUAR/html/ir/CtsComponents__MessageService.html* for the API definitions and examples of how to use the MessageService interface.
- **MessageQueue** The CtsComponents::MessageQueue interface allows clients to receive messages from a queue, get a list of messages in a queue, acknowledge the receipt of messages, close a message queue object, and recover messages that have been received but not acknowledged. See *$JAGUAR/html/ir/CtsComponents__MessageQueue.html* for the API definitions and examples of how to use the MessageQueue interface.
- **MessageListener** The CtsComponents::MessageListener interface allows an application component to be notified when new messages are sent or published to its message queue or topic. See *$JAGUAR/html/ir/CtsComponents__MessageListener.html* for the API definition and an example of how to use the MessageListener interface.

CHAPTER 32

Using the Thread Manager

The Thread Manager allows you to start threads from EAServer components to perform asynchronous processing.

About the Thread Manager

The Thread Manager allows you to run EAServer component instances in threads that execute independently of client method invocations. You can use threads spawned by the Thread Manager to perform any processing that must occur asynchronously with respect to user interaction. For example, you might have a component method that begins a lengthy file indexing operation. The method could call the Thread Manager to start the processing in a new thread, then return immediately.

The Thread Manager and service components

You can use the Thread Manager as an alternative to creating a service component to handle repetitive processing. You may find the Thread Manager interface allows more design flexibility. For example, you can suspend processing in services run by the Thread Manager, and you can start threads at any time rather than only at server start-up.

The Thread Manager is the recommended way to spawn threads in Java or C++ components. In C++, using the Thread Manager avoids system-level thread calls that may affect portability. In Java and C++, components running in the Thread Manager can make in-memory intercomponent calls, whereas components running in user-spawned threads must make intercomponent calls through the network.

You can use the Thread Manager and service components together. For example, you might code a simple service component that spawns threads in the start or run method, and stops them in the stop method.

PowerBuilder developers can use the Thread Manager to develop more robust services. Since PowerBuilder components cannot support sharing and concurrency, you cannot develop a service that can be stopped or refreshed without using the Thread Manager. In the services start or run method, spawn threads that do the service's processing. In the service's stop method, call the Thread Manager stop method to halt the threads. For more information, see the *Application Techniques* manual in the PowerBuilder documentation.

The Thread Manager and the message service

If you are using threads to implement a provider/consumer algorithm, or an asynchronous notification system, consider using the EAServer message service implementation described in Chapter 31, "Using the Message Service." The message service provides a ready-made infrastructure for solving these classes of problems. The message service uses the Thread Manager in its own implementation.

Using the scheduling facilities provided by the message service, you can restrict background processing to a server's off-peak hours. For example, you may have threads running that index the text content of a Web site. Using a scheduled component and the Thread Manager, you can suspend processing at the beginning of the server's peak use period, then resume processing at the end. "Subscribing to scheduled messages" on page 594 describes how to subscribe to scheduled topics in the message service.

Thread Manager interface documentation

This chapter briefly discusses how to use the Thread Manager methods. For reference documentation, see the generated HTML documentation for the CtsComponents::ThreadManager IDL interface. You can view this documentation in the *html/ir* subdirectory of your EAServer installation. Using a Web browser, load the URL:

```
http://host:port/ir/index.html
```

Where *host* is your server's host name, and *port* is the HTTP port number.

Using the Thread Manager

The Thread Manager is a built-in EAServer component. You can create a proxy and execute methods the same way that you would call any other component. Each thread executes a run method in an EAServer component that you specify.

The thread manager is designed primarily for use in server-side code. However, it is possible to call thread manager methods from base clients or Web applications. For example, you can create an administrative client that stops threads created by your application.

Before you start

Before running components in the Thread Manager, make sure you understand how the component must be prepared, how threads are run in thread groups, and the effect of a thread group's run interval.

Adapting components to be run by the Thread Manager

Each thread runs an EAServer component instance. To be run by the Thread Manager, the component must have a run method with this IDL signature:

```
void run ( );
```

The Thread Manager calls the run method one or more times, depending on how you configure the run interval (described below).

For EJB components, the run method must be in a remote interface or an additional interface that is neither an EJB remote or local interface. To add such an interface, follow the procedure "Adding interfaces" on page 74. You can use the predefined CtsComponents::ThreadBase. Regenerate the component skeleton after adding interfaces.

The Thread Manager is itself an EAServer component, and runs your component using intercomponent calls. All component properties, including transaction attributes, are in effect when your component is run by the Thread Manager. The Thread Manager executes with the system identity, as does your component's run method.

Understanding thread groups

Threads are associated with a thread group. To start, stop, suspend, or manage the run interval of threads, you must specify the group name. These operations affect all threads in the specified group. The group name is simply a string. Group names have a scope limited to one server; that is, you cannot have two like-named groups in the same server. If two applications use the same group name, their Thread Manager calls affect threads in both applications. You can run different components in one thread group.

Naming conventions for thread groups
To avoid collisions between thread groups used by different applications, use the reverse-domain naming convention for group names, as used in Java package names. For example, “com.foo.mythreadgroup”.

Understanding the run interval

Each thread group has a run interval, which determines how often the Thread Manager calls the run method. The run interval can be:

Run interval	Meaning
A positive integer *n*	The Thread Manager calls run repeatedly, waiting approximately *n* seconds after each time the run method returns. The actual time can vary depending on scheduling of calls to other methods and the server’s processing load.
0	The Thread Manager calls run repeatedly, with no waiting between invocations.
-1 (the default)	The Thread Manager calls run only once.

To allow threads to be stopped or suspended, you must configure a positive or 0-length run interval and code each component’s run method to perform a repetitive task, then return. The run interval has no effect if your run method never returns.

If the run interval is positive or 0, you can change the run interval after threads have been started in the group, the change takes for each thread when it returns from the run method. You cannot change the interval to -1, and changing the interval does not affect threads started with the interval set to -1. In these cases, calling setRunInterval has no effect.

You can use a run interval to schedule periodic tasks, such as refreshing a cached copy of a database query result. You can also tune how much CPU time your component consumes if it performs CPU-intensive tasks such as lengthy calculations; such tuning also requires that you adjust the amount of work done in each invocation of the run method.

You can also use the Message Service to schedule periodic background processing. For example, you can configure a run interval of -1 (so Thread Manager calls run once only) and schedule another component to start threads at the desired interval. See "Subscribing to scheduled messages" on page 594 for more information.

Understanding the thread count

Each thread group has a thread count, which determines how many threads can run simultaneously. The count can be:

Run interval	Meaning
-1 (the default)	There is no limit.
A positive integer *n*	*n* threads can execute.
0	No threads can execute..

To change the thread count, call the ThreadManager::setThreadCount method. The change takes effect after threads return from the run method. Thread counts are useful if threads run repeatedly (run interval is positive or 0). For example, if 6 threads are running, and you change the count to 5, the next thread that returns from its run method will not be restarted. The thread count provides a means to throttle the number of running threads, without stopping or suspending all threads.

Instantiating the Thread Manager

Other than restricted access, the Thread Manager can be instantiated as you would instantiate any other component.

Obtaining authorized access

To instantiate the Thread Manager, your client or component must execute with with the system identity or an identity that is in the ThreadManager role. These are the recommended ways to satisfy this constraint:

- Start threads from a service component and create the Thread Manager proxy in the service's start or run method. These methods execute with the system identity.
- For a component that is pooled or shared, create the Thread Manager proxy in the component's class constructor, the setSessionContext or setEntityContext method (for EJB components), or the setObjectContext method (for CORBA components). All of these methods execute with the system identity.
- For a component that is not a service and not pooled or shared:
  - Delegate Thread Manager operations to another component that is pooled or shared, or
  - Run the component with an identity that is in the ThreadManager role.
- For a base client, connect to EAServer with a user name that is a member of the ThreadManager role.

ThreadManager privileges can be dangerous
User accounts with ThreadManager role membership can use the Thread Manager to implement denial of service attacks or to stop thread groups. Treat ThreadManager role accounts with the same care as you would Admin role accounts.

Instantiating a proxy

Use the standard technique for your component model to instantiate the Thread Manager proxy.

CORBA (C++ and Java), ActiveX, and PowerBuilder components must declare a stub for the CtsComponents::ThreadManager IDL interface, then instantiate the component named *CtsComponents/ThreadManager*.

EJB components must use the home interface com.sybase.ejb.cts.ThreadManagerHome to create a stub for the remote interface com.sybase.ejb.cts.ThreadManager. Look up the name *CtsComponents/ThreadManager* to obtain the home interface.

Starting threads

To start threads:

1 Optionally, configure a run interval by calling the setRunInterval method, specifying the group name.

2 If necessary, create proxies for the components that will run in the thread group. For stateless or shared-instance components, you can use one proxy instance to run the component on multiple threads. For stateful components, create a proxy for each component instance and initialize the instance state as necessary.

3 Start the desired number of threads by calling the start method once per thread. In each call, specify the group name and pass a proxy for the component that is to run in the thread.

If you have set a thread count, and try to start more threads than the thread count, the behavior depends on the run interval. If the run interval is -1, all threads are started and run once. If the run interval is 0 or positive, the start method does not create additional threads after the count is reached.

Suspending and resuming execution

To suspend the threads in a group, call the ThreadManager::suspend method, specifying the group name. Each thread is suspended when it next returns from its run method.

To resume execution, call the ThreadManager:resume method.

Stopping threads

You can only stop threads that return from their run method. The Thread Manager stops each thread the next time it returns from its run method.

You can stop threads in two ways:

- **By decreasing the thread count** Call the ThreadManager::setThreadCount method to reduce the number of threads executing in the thread group. This technique is useful when you want to throttle the execution of the task. For example, during a Web site's peak usage hours, you can reduce the thread count for background processing to give user threads more CPU time. During off hours, you can reset the thread count and start new threads to raise the thread count again.

- **By stopping all threads in the group** Call the ThreadManager::stop method to stop all threads in the group. This method is equivalent to calling ThreadManager::setThreadCount to reduce the thread count to zero.

If you stop all threads by calling ThreadManager::stop or setting the thread count to 0, you must reset the thread count to a positive value or -1 (meaning infinity) before starting more threads.

CHAPTER 33

Creating Service Components

This chapter describes how to create service components. Service components are loaded and initialized when EAServer starts and have a run method that executes perpetually, independent of any client interaction.

You can use service components to perform background processing or to provide common services for EAServer clients and other EAServer components.

Introduction

Service components perform background processing or provide common services for EAServer clients and other EAServer components. For example, you might create service components to perform the following tasks:

- Maintain cached copies of commonly used database tables
- Move or replicate data between data sources during server idle time
- Manage application-specific log files

What are service components?

Service components are like any other EAServer component, except that:

- They must implement the methods in the CtsServices::GenericService interface.
- Instances are loaded and initialized when the host server starts.
- They can run independently of client interaction.

The Thread Manager and service components
You can use the Thread Manager as an alternative to creating a service component to handle repetitive processing. You may find the Thread Manager interface allows more design flexibility. For example, you can suspend processing in services run by the Thread Manager, and you can start threads at any time rather than only at server start-up. Chapter 32, "Using the Thread Manager" describes how to use the thread manager.

PowerBuilder developers can use the Thread Manager to develop more robust services. Since PowerBuilder components cannot support sharing and concurrency, you cannot develop a service that can be stopped or refreshed without using the Thread Manager. For more information, see the *Application Techniques* manual in the PowerBuilder documentation.

The GenericService interface

Your component implementation must implement all the methods in the CtsServices::GenericService interface. Your implementation does not need to explicitly implement the interface (that is, list it in the implements clause of the class declaration), and you do not need to list the interface in the component's Interfaces folder in EAServer Manager.

EAServer calls the CtsServices::GenericService methods to indicate transitions in the service's state:

- **start()** Called to initialize the component when the server starts or when the component has been refreshed from EAServer Manager. This method typically initializes data structures and resources that the service requires. For example:
  - A service that writes to log files would open each file and cache each file handle as a class instance variable.
  - A service that caches tabular data from a remote database would open a connection to the database and create the data structures required to store tabular data in memory.

Service components that make intercomponent calls
If your start method makes intercomponent calls, check the `com.sybase.jaguar.server.bindrefresh` property for the servers where your component is installed. Use the Advanced tab in the Server Properties dialog box to view and change this property. This property must be set to "start" to allow name service lookups in the start method of service components. The default setting is "run".

- **run()** Called after the first invocation of start() returns. run() can loop and perform repetitive tasks as an EAServer background process. If the component does not perform background processing, run() can return immediately.

  For services that perform background processing, run() should loop continuously while performing the service task. run() must periodically suspend its own execution by calling the Java java.lang.Thread.sleep() method, one of the Java Object.wait() methods, or the EAServer JagSleep C routine. These APIs suspend the current thread for a specified duration so that other threads may execute. run() should return after the server invokes the stop() method.

  If you configure your service to run in multiple threads, EAServer calls run() concurrently in the specified number of threads.

  Warning! Your run() method must either return immediately or call one of the Object.wait() Java methods, the EAServer JagSleep C routine, or some other thread-aware implementation of sleep. Do not call the sleep system routine or any other routine that suspends process (and not thread) execution. If coding service components in PowerBuilder, code your component to call the JagSleep C routine; do not use the PowerBuilder timer event, which may suspend the EAServer process.

- **stop()** Called when the server is shutting down or when the component has been refreshed from EAServer Manager (refresh stops the service and reloads it). EAServer calls the stop() method on a different thread than the run() method. Code in the stop() method should set a flag that indicates the the run() method should return.

  stop() should also wake up sleeping run() threads if the language allows this. For example, in Java, call the Object.notifyAll() method to wake threads that called Object.wait() on the same monitor object. In languages that do not allow you to wake up sleeping threads, keep your sleep interval reasonably short. The service cannot be refreshed until all running threads return from the run() method; that is, if your sleep interval is one hour, it can take that long to refresh the service unless you add code to wake up sleeping threads.

Implementing other interfaces

Your component can implement additional interfaces. EAServer clients, servlets, and other components can execute a service component's methods like those of any other component, with one exception: Clients cannot invoke methods on the service component until the start() method has returned. This restriction allows you to perform required initialization in start() without worrying about thread synchronization issues.

After start() returns, EAServer calls the run() method in its own thread. Client method invocations may arrive at this time as well. There is no guarantee that run() will have been called when a client method invocation occurs; the first client invocations may arrive before EAServer calls the run() method.

Creating service components

Follow the steps below to create a service component:

1 "Define the component interface and properties" on page 608

2 "Implement GenericService interface methods" on page 610

3 "Implement other required methods" on page 614

4 "Install the component as an EAServer service" on page 614

Define the component interface and properties

Except for a few special requirements described here, you define a service component's interface and properties in EAServer Manager as you would do for any component. Chapter 4, "Defining Components" describes how to define components in EAServer Manager.

Service component properties

Service components require these EAServer Manager settings in the Component Properties window:

- **IDL Interface** Your service component must implement the CtsServices::GenericService interface. You can define additional methods in one or more additional IDL interfaces if necessary.

- **Concurrency and Bind Thread Options** For best performance, you must enable the Concurrency option on the Instances tab, and disable the Bind Thread option.

  Selecting the Concurrency option allows multiple method invocations to occur simultaneously. Concurrent access can decrease client response time. Also, if your component has a run() method that executes indefinitely, you must enable the Concurrency option or no clients will be able to invoke methods. To support concurrency, you must ensure that access to read/write instance variables is synchronized in your component.

  Disabling the Bind Thread option allows EAServer to run the component on any available thread. This option is only required by ActiveX components (where it is on implicitly) and by components that use thread-local storage. It should be disabled in any other case.

- **Sharing Option** For simplified implementation, select the Sharing option on the instances tab. This option ensures that only a single instance of your component is created. One instance serves all client requests. With Sharing enabled, the component can store data in class instance variables. If Sharing is disabled, you must coordinate access to shared data among multiple instances of the component (typically, data shared by multiple instances is stored in static class variables, in a database, or in EAServer shared properties).

- **Transaction Attribute** Do not create service components that are transactional. On the Transactions tab, choose Not Supported. If you require EAServer's transaction semantics, implement a component to perform the transaction-created work and call this component from your service component

Service components cannot be transactional
EAServer-managed transactions require a component lifecycle that allows component deactivation, and a service component is never deactivated.

- **Automatic Demarcation/Deactivation** If a service component is installed to run in multiple service threads, the component must be stateless.You must enable this option if the multiple instances will run as service components. See "Install the component as an EAServer service" on page 614 for information on configuring the component so multiple instances run as services.

Required client roles

You can assign the role ServiceControl to service components so that base clients and other components cannot create instances of the component and call the start and stop methods. No users can be added to this role. To assign this role to a component, display the Advanced tab in the Component Properties dialog and modify the `com.sybase.jaguar.component.roles` property. Add "ServiceControl" to the list of comma-separated role names.

Implement GenericService interface methods

Each service component must implement the CtsServices::GenericService interface. Your component can implement additional interfaces if necessary. This section describes how to implement the CtsServices::GenericService in C++ and Java.

Be careful of consuming CPU cycles

If your service will perform background processing, your implementation must have access to a thread-aware sleep mechanism. In Java, call the java.lang.Thread.sleep() method, or use a monitor object and call the Object.wait() method. In C, C++, ActiveX, or PowerBuilder, EAServer provides the JagSleep routine. The run method in your service must call one of these APIs periodically to suspend execution of the current thread. Otherwise, your service will dominate the server's CPU time and prevent other components from executing.

If coding service components in PowerBuilder, code your component's run method to call the JagSleep C routine; do not use the PowerBuilder timer event, which may suspend the EAServer process.

Services with a client interface
If your component runs as a service and also provides a client interface for remote invocations, beware that the run method may not have executed when the first client request arrives. run is called on a different thread after start returns; client invocations may arrive between the return from start and the invocation of run, and initialization performed in run may not have completed when the remote method executes on a different thread. To avoid problems, use one of these approaches:

- Do not code remote methods that rely on initialization performed in the run method. Initialization can be performed in the start method, which is guaranteed to complete before client invocations arrive.
- Use a synchronized boolean variable that is set when run has performed necessary initialization, and code remote methods to check this variable and wait for it to be set before executing code that relies on initialization performed in run.

Java example of GenericService methods

The example uses a static Boolean instance variable, *_run*, to indicate when the service should cease running. There is also a java.lang.Object that is used as a semaphore to allow synchronization among multiple threads. The start() method sets the *_run* variable to true; start() must also perform any other necessary initialization that are needed by your service, such as opening files, database connections, and so forth. run() executes a while loop as long as the *_run* variable is true. In each loop iteration, run() performs some of the work that the service is responsible for, such as refreshing a copy of a remote database table, then calls the Object.wait() method to relinquish the CPU. The stop() method sets the *_run* variable to false and calls the Object.notifyAll() method on the semaphore, causing the run() method to return. Before returning, run() cleans up resources that were allocated in the start() method.

```
public class MyService
{
public static boolean _run;
public static Object _lock = new Object();

public void start()
{
    _run = true;
    ... perform necessary initializations ...
}
```

```
public void run()
{
    while (_run)
    {
        try
        {
            ... do whatever this service does
                on each iteration, then go back
                to sleep for a while ...
            synchronized(_lock)
            {
                _lock.wait(100000);
            }
        }
        catch (InterruptedException ie)
        {
            _run = false;
        }
    }
    ... perform necessary cleanup and deallocations ...

}

public void stop()
{
    _run = false;
    // Wake up any instances that are waiting on the mutex
    synchronized (_lock)
    {
        _lock.notifyAll();
    }
}
}
```

C++ example of GenericService methods

The code fragment below shows how the GenericService methods can be implemented in a C++ component. This example uses a static Boolean instance variable, *_stop*, to indicate when the service should cease running. The start() method sets the *_stop* variable to false; start() must also perform any other necessary initialization that are needed by your service, such as opening files, database connections, and so forth. run() executes a while loop as long as the *_stop* variable is false. In each loop iteration, run() performs some of the work that the service is responsible for, such as refreshing a copy of a remote database table, then calls the JagSleep C routine to relinquish the CPU. The stop() method sets the *_stop* variable to true. stop() must also clean up any resources that were allocated in the start() method.

```
#include <jagpublic.c> // For JagSleep API

class MyService
{
private:
    static boolean _stop; // Declared static in case multiple
                          // instances are run.

public:
void start()
{
    _stop = false;
    ... perform necessary initializations ...
}

void stop()
{
    _stop = true;
}

void run()
{
    while (! _stop)
    {
        ... do whatever this service does
            on each iteration ...
        JagSleep(1000);
    }
    ... perform necessary cleanup and deallocations ...
}

};
```

Implement other required methods

Your component may implement additional interfaces besides CtsServices::GenericService. For example, in a component that manages application-specific log files, you need a method that other components can call to write to the application log. Follow the implementation rules for the component model that you are using. See the following chapters for more information:

- Chapter 11, “Creating CORBA Java Components”
- Chapter 14, “Creating CORBA C++ Components”
- Chapter 19, “Creating ActiveX Components”

Install the component as an EAServer service

In order to run as a service, your service component must be added to the host server’s list of services, as follows:

❖ **Installing services**

1 Start EAServer Manager if it is not already running.

2 Expand the Servers folder.

3 Expand the icon for the server.

4 Highlight the Installed Services folder under the server icon, then choose File | Install Services from the menu.

5 Components that implement the CtsComponents::GenericService interface are listed. Pick the component to install, then click OK.

6 The service will run the next time you refresh or restart the server.

❖ **Configuring a service to run in multiple threads**

By default, one thread runs per service. You can specify a larger number of threads as follows:

1 Display the server properties.

2 In the list of properties, select “com.sybase.jaguar.server.services”, then click Modify.

3 The value of this property is the list of services, using the form *Package/Component*, with entries separated by commas. To specify multiple threads for a service, enter the number of threads in brackets after the component name. For example:

```
YourPackage/YourService[10]
```

4 Click Ok to close the Modify Property window.

5 Click Ok to close the Server Properties window.

6 The change takes affect the next time you refresh or restart the server.

When multiple threads are requested

The host server calls the component's run method from the specified number of threads. If the Sharing option is enabled, all threads call run on the same component instance as start was called in. Otherwise, each thread will create a new instance of the component and call run on that instance. Each thread terminates when run returns.

This feature is useful when your service component performs a background task that lends itself to parallel processing. For example, if the run implementation extracts work requests from a queue and performs the requested operation, you can configure the server so multiple threads read requests from the queue and process them simultaneously. The component must be coded to ensure that access to the queue is thread-safe, for example, in Java, you might create synchronized methods to queue and dequeue.

The component must be stateless in order to run in multiple threads. Make sure the Automatic Demarcation/Deactivation is option is checked on the Transactions tab in the Component Properties window.

Note The start method and stop methods are only called on one instance of a service component. If Sharing is not set for the component, start must store any data required by the run method or other methods. For access by multiple instances, data must be stored in static fields or a persistent data store.

Determining service state

The jagtool getservicestate command returns the state of service components executing in the server. You must code your service component to implement the methods of the CtsServices::ExtendedService interface to allow users to query the component state with jagtool.

This interface extends CtsServices::GenericServices, and adds one method:

```
long getServiceState()
```

This method must return one of the constants listed in Table 33-1 to describe the state of the service. These constants are defined in module CtsServices.

Table 33-1: Service states

State	Description
UNKNOWN	The state is unknown.
STARTING	The service is starting. The start method has been called, but has not returned.
STARTED	The service is started, but not yet running. The start method has returned, but run has not been called.
RUNNING	The service is running, that is, executing the run method.
FINISHED	The service is finished processing. The run method has returned. This state applies only to services that do not run continuously until stopped.
STOPPING	The service is stopping. The stop method has been called, and is still running.
STOPPED	The service is stopped. The stop method has been called and has returned.

The following Java example shows service component code that determines and returns state:

```
import CtsServices.*;

...

public class MyService
{
    private static boolean _starting = false;
    private static boolean _running = false;
    private static boolean _stopping = false;
    private static boolean _stop = false;
    private static boolean _runHasBeenCalled = false;
    private static Object _lock = new Object();
    public void start()
    {
        _starting = true;
        // Perform initialization
        _starting = false;
    }
    public void stop()
    {
```

```
        _stopping = true;
        _running = false;
        _stop = true;
        synchronized (_lock)
        {
            _lock.notifyAll();
        }
        // Perform cleanup
        _stopping = false;
    }
    public void run()
    {
        _runHasBeenCalled = true;
        // Perform per-thread initialization here.
        _running = true;
        while (! _stop)
        {
            try
            {
                // do whatever this service does on
                // each iteration
                synchronized(_lock)
                {
                    _lock.wait(100000);
                }
            }
            catch (InterruptedException ie)
            {
                _stop = true;
            }
        }
        // Perform per-thread cleanup here.
        _running = false;
    }
    public int getServiceState()
    {
        if (_starting)
        {
            return SERVICE_STATE_STARTING.value;
        }
        else if (! _runHasBeenCalled)
        {
            return SERVICE_STATE_STARTED.value;
        }
        else if (_stopping)
        {
```

```
                return SERVICE_STATE_STOPPING.value;
            }
            else if (_stop)
            {
                return SERVICE_STATE_STOPPED.value;
            }
            else if (_running)
            {
                return SERVICE_STATE_RUNNING.value;
            }
            else
            {
                return SERVICE_STATE_FINISHED.value;
            }
        }
    }
```

Refreshing service components

To refresh the component implementation after it has been loaded, select the component icon, and choose File | Refresh.

Note Components that are installed as EAServer services are not refreshed when you refresh the package or server in which they are installed. To refresh a service component, you must select the component icon and choose File | Refresh.

For refresh, EAServer reloads component instances as follows:

1 The server calls the stop() method.

2 The server waits for the run() method to return in all instances that are running as services.

3 The server creates a new instance and calls the start() and run() methods, in that order. If the multiple instances are specified for the service, the server loads the additional instances that are required and calls run() on each instance.

After refresh, a new instance is guaranteed not to start before previous instances have ceased running. Consequently, a service component can not be refreshed unless the run() method returns. See “Implement GenericService interface methods” on page 610 for code examples that show how to coordinate the logic in the stop() and run() methods.

CHAPTER 34

Creating and Using EAServer Pseudocomponents

A pseudocomponent is instantiated and called without using the EAServer component dispatcher. Pseudocomponents can be instantiated by client programs or by components executing in EAServer.

Benefits of pseudocomponents

For cross-language development, pseudocomponents offer the benefit of a component-based architecture without incurring network overhead. For example, you can call methods in a C++ pseudocomponent from Java programs without the use of Java Native Interface (JNI) calls.

Since pseudocomponents are executed locally, in the same process, they do not incur the network overhead of client/server communication. When used in EAServer, pseudocomponents avoid the small thread- and context-management overhead incurred when the EAServer component dispatcher executes intercomponent calls.

However, pseudocomponents are not suitable for applications that require the transaction control, threading control, security constraints, instance lifecycle management, or other services provided by the EAServer component dispatcher.

Creating pseudocomponents

For the most part, pseudocomponents can be created and implemented like any EAServer CORBA/Java or C++ component. However, since they are not executed by the component dispatcher, there are additional restrictions on their implementation and use. This section explains the implementation restrictions and required property settings for pseudocomponents. For additional information on creating components, see “Defining components” on page 49.

Implementation restrictions

Pseudocomponents must be implemented in C++ or Java. If using Java, the component type must be Java/CORBA; you cannot instantiate an Enterprise JavaBean as a pseudocomponent.

Since pseudocomponents execute outside of the EAServer dispatcher, their execution is not governed by component properties defined in EAServer Manager. Thus, components that run as pseudocomponents are subject to these restrictions:

- They cannot participate in server-managed transactions.
- Lifecycle interface methods, such as those in CtsComponents::ObjectControl, are not called and instances are never pooled or reused. Each time a client program instantiates a proxy instance of a pseudocomponent, a new instance of the implementation class is constructed.
- They are not affected by component threading properties. A pseudocomponent’s methods run in the same process of the calling program, on the same thread from which they are called.
- Pseudocomponents executed in standalone programs cannot use server-side classes and methods such as connection management, client credential access, and so forth. These classes and methods are available only to components executed by the EAServer component dispatcher. A pseudocomponent that is executed by a server-dispatched component can use server-side features, but does so in the context of the server-dispatched component that instantiated the pseudocomponent.
- They are not affected by access control restraints. Any client with local access to a component’s implementation library or class can instantiate the component as a pseudocomponent.

Defining a pseudocomponent

A pseudocomponent must be defined in EAServer Manager in order to generate stubs and skeletons. Stubs and skeletons are required to execute the pseudocomponent. You can use any of the techniques described in "Defining components" on page 49 to define the component in EAServer Manager. You must configure the properties described in this section.

Using existing components
You can also instantiate existing C++ or CORBA/Java components as pseudocomponents, provided the implementation abides by the restrictions listed in this document.

Properties for a Java pseudocomponent

For a Java pseudocomponent, set the following properties and leave other properties at their default settings. Properties other than these have no affect on the behavior of the pseudocomponent:

- Component Type – choose Java - CORBA.
- Java Class – enter the dot-notation Java class name, for example `com.sybase.sample.PseudoJavaImpl`.

Java pseudocomponents must have a public constructor
If the implementation class declares a default (no arguments) constructor, the default constructor must be declared public.

Properties for a C++ pseudocomponent

The Component Properties dialog box appears. Set the following fields for a C++ pseudocomponent, leaving other fields at their default settings. Properties other than these have no affect on the behavior of the pseudocomponent:

- Component Type – choose C++.
- DLL Name – enter the base name for the shared library or DLL that will contain the component implementation. For example, the setting `mypseudo` indicates the Windows file *mypseudo.dll* on Windows.
- C++ Class – leave at the default setting (the name of the component, appended with "Impl", as in `PseudoCPPImpl`).

Direct-access pseudocomponent stubs and skeletons

You can generate special stubs and skeletons that improve the performance of pseudocomponent method calls issued from Java or C++. In a process called *marshalling*, regular CORBA stubs convert parameter and return values to the format required for IIOP network transport. Direct-access pseudocomponent stubs and skeletons improve performance by eliminating the marshalling step.

To generate direct-access pseudocomponent stubs and skeletons in EAServer Manager, select the normal EAServer Manager settings for Java/CORBA or C++ Stubs, except in the Advanced Options wizard page, select the option Generate Pseudo Component Access in C++ and Java Stubs and Skeletons.

Instantiating pseudocomponents

To instantiate a pseudocomponent, call the ORB.object_to_string method, passing a URL that specifies the information required to load the component. Java, C++, PowerBuilder, and ActiveX all use a variation of this method.

Pseudocomponent object URLs

The object URL for a pseudocomponent specifies the shared library file or Java class that contains the implementation, the EAServer package name, and the EAServer component name.

Identifying a C++ pseudocomponent

To identify a C++ pseudocomponent, format a URL as follows:

```
pseudo://cpp/library/package/component
```

Where:

- *library* is the base name of the shared library or DLL that contains the component implementation, without the platform-specific file extension. For example, `mypseudo` to indicate a file named *mypseudo.dll* on Windows or *mypseudo.so* on Solaris or Linux. The location of the library must be specified in the system's library search path.
- *package* is the EAServer Manager package name.

- *component* is the EAServer Manager component name.

Identifying a Java pseudocomponent

To identify a Java pseudocomponent, format a URL as follows:

```
pseudo://java/java-package/jaguar-package/component
```

Where:

- *java-package* is the is the dot-notation name of the Java package that contains the component skeleton. This package is the same as the package used by the implementation class. For example, if the implementation class is com.sybase.sample.MyJavaPseudo, specify `com.sybase.sample` as the Java package name. The code base under which the class is deployed must be specified in the CLASSPATH.
- *jaguar-package* is the EAServer Manager package name.
- *component* is the EAServer Manager component name.

Instantiating pseudocomponents from Java

Java applications or EAServer Java components can instantiate pseudocomponents implemented in Java or C++. Java applets cannot instantiate pseudocomponents. In order to instantiate a C++ pseudocomponent, the environment for a Java application must include all the settings required by the EAServer C++ client runtime, and the location of the library must be specified in the system's library search path. Java stub classes for the pseudocomponent must be available.

You can instantiate a pseudocomponent any time after initializing and instantiating an ORB instance. Call the ORB.string_to_object method, passing a URL formatted as described in "Pseudocomponent object URLs" on page 624. Narrow the returned object to an interface supported by the component. See Chapter 12, "Creating CORBA Java Clients" for more information on the ORB interface and narrowing objects to an interface.

Example: instantiating a C++ pseudocomponent

The following fragment instantiates a pseudocomponent proxy for a C++ component in the DLL *CppPseudo.dll* that is installed in the package Demo and has component name PseudoCpp. The returned object is narrowed to Arithmetic interface. On UNIX platforms, this syntax also works for a shared library with base name "CppPseudo", as in *CppPseudo.so*.

```
String url = "pseudo://cpp/CppPseudo/Demo/PseudoCPP";
```

```
org.omg.CORBA.Object obj = orb.string_to_object(url);
_comp = ArithmeticHelper.narrow(obj);
```

Example: instantiating a Java pseudocomponent

The following fragment instantiates a pseudocomponent proxy for a Java component. The implementation class and skeleton class are in the Java package Sample.PseudoComponents. The component is installed in the EAServer package Demo and has component name PseudoJava. The returned object is narrowed to Arithmetic interface.

```
String url =   "pseudo://java/Sample.PseudoComponents/Demo/PseudoJava";
org.omg.CORBA.Object obj = orb.string_to_object(url);
_comp = ArithmeticHelper.narrow(obj);
```

Instantiating pseudocomponents from C++

C++ standalone programs or EAServer components can instantiate pseudocomponents implemented in C++. Pseudocomponents implemented in Java can be instantiated only by C++ components that are executing in EAServer.

In order to instantiate a C++ pseudocomponent in a standalone program, the environment must include all the settings required by the EAServer C++ client runtime, and the location of the library must be specified in the system's library search path.

You can instantiate a pseudocomponent any time after initializing and instantiating an ORB instance. Call the ORB.string_to_object method, passing a URL formatted as described in "Pseudocomponent object URLs" on page 624. Narrow the returned object to an interface supported by the component. See Chapter 15, "Creating CORBA C++ Clients" for more information on the ORB interface and narrowing objects to an interface.

Example: instantiating a C++ pseudocomponent

The following fragment instantiates a pseudocomponent proxy for a C++ component in the DLL *CppPseudo.dll* that is installed in the package Demo and has component name PseudoCpp. The returned object is narrowed to PseudocomponentDemo::Arithmetic interface. On UNIX platforms, this syntax also works for a shared library with base name "CppPseudo", as in *CppPseudo.so*.

```
String url = "pseudo://cpp/CppPseudo/Demo/PseudoCPP";
CORBA::Object_var obj = orb->string_to_object(url);
PseudocomponentDemo::Arithmetic_var arith =
  PseudocomponentDemo::Arithmetic::_narrow(obj);
```

Example: instantiating a Java pseudocomponent

The following fragment instantiates a pseudocomponent proxy for a Java component. The implementation class and skeleton class are in the Java package Sample.PseudoComponents. The component is installed in the EAServer package Demo and has component name PseudoJava. The returned object is narrowed to PseudocomponentDemo::Arithmetic interface.

```
String url =   "pseudo://java/Sample.PseudoComponents/Demo/PseudoJava";
CORBA::Object_var obj = orb->string_to_object(url);
PseudocomponentDemo::Arithmetic_var arith =
  PseudocomponentDemo::Arithmetic::_narrow(obj);
```

Instantiating pseudocomponents from PowerBuilder

To instantiate pseudocomponents in PowerScript, use the String_To_Object method in the JaguarORB object, specifying the pseudocomponent URL as the string to resolve. For example, the following code can be called in a PowerBuilder component to retrieve a proxy for the CtsSecurity/SessionInfo built-in pseudocomponent:

```
// PowerBuilder objects
JaguarORB my_JaguarORB
CORBAObject my_corbaobj

// Proxy object for CtsSecurity::SessionInfo built in
// pseudocomponent
SessionInfo my_sessioninfo

long ll_return
my_JaguarORB = CREATE JaguarORB

// Initialize the ORB
ll_return = my_JaguarORB.init("")

// Convert a URL string to an object reference
ll_return = my_JaguarORB.String_To_Object &
    ("pseudo://cpp/libjdispatch/CtsSecurity/SessionInfo", &
    my_corbaobj)

// Narrow the object reference to the Manager interface
ll_return = my_corbaobj._narrow(my_sessioninfo, "CtsSecurity/SessionInfo")
```

For more information on using the JaguarORB object, see the *Application Techniques* manual in the PowerBuilder documentation. For information on the CtsSecurity/SessionInfo API, see the generated HTML documentation, available in the *html/ir* subdirectory of your EAServer installation.

Debugging C++ pseudocomponents

Once loaded in your debugger, a C++ pseudocomponent can be debugged like any other shared library or DLL. However, since the library is not loaded until a client program instantiates the pseudocomponent, setting breakpoints is tricky. The procedure below allows you to set breakpoints and step into your method code.

❖ **Debugging a C++ pseudocomponent**

1 Verify that the process is using the debug versions of the EAServer libraries. For pseudocomponents executing in EAServer, start the debug version of the server executable. For standalone programs, verify that the debug DLLs or libraries are ahead of the non-debug libraries in your system's library search path. (On UNIX platforms, the debug libraries are in the *lib/debug* directory of your client installation. On Windows, they are in the *dll/debug* directory.)

2 Attach the program that is instantiating the pseudocomponent with your debugger. This can be a standalone client executable, or EAServer process.

Alternatively, start the debugger to load the executable. For example, on UNIX, this command starts the dbx debugger and loads the debug server executable:

```
dbx $JAGUAR/devbin/jagsrv ServerName
```

As another example, on Windows this command starts the Microsoft Visual C++ debugger and loads the debug server executable:

```
msdev %JAGUAR%\devbin\jagsrv ServerName
```

In these examples, *ServerName* is the name of the server. If you are using the preconfigured server rather than one that you created yourself, use "Jaguar".

3 Set a breakpoint on the function jag_client_dbg_stop. This function executes every time the client runtime constructs a pseudocomponent instance. The jag_client_dbg_stop prototype is:

```
void jag_client_dbg_stop(char *compName)
```

The *compName* parameter specifies the name of the library or shared library that was just started. Several pseudocomponents may be loaded before yours. In the debugger, display the *compName* value when the jag_client_dbg_stop breakpoint is tripped, and monitor the value to determine when your component is loaded.

Note Make sure the jag_client_dbg_stop breakpoint is set before your client application instantiates any pseudocomponents.

4 When your pseudocomponent's DLL is loaded, you can specify the method names as breakpoints and step into the method's code when it is invoked.

CHAPTER 35

Creating JavaMail

EAServer supports version 1.1 of the JavaMail API. JavaMail allows you to send electronic mail from Java servlets, Java components, or standalone Java applications. The JavaMail API provides a standard Java interface to the most widely-used Internet mail protocols.

JavaMail requires JDK 1.2 or later
You must run EAServer with JDK 1.2 or later to use JavaMail in components, servlets, or JSPs. Java applications using JavaMail must be run with JDK (or JRE) 1.2 or later.

Introduction to JavaMail

JavaMail is a Java standard extension that provides a set of abstract classes that define the common objects and their interfaces for any general mail system. JavaMail providers implement the API to provide the concrete functionality needed to communicate using specific protocols such as the Simple Mail Transfer Protocol (SMTP) and the Internet Message Access Protocol (IMAP).

Using JavaMail in EAServer, you can send e-mail messages from Java components, servlets, or JSPs. For example, a Web-based bookstore could send e-mail to a customer acknowledging an order, or to a System Administrator warning that a database is full.

Note EAServer supports only the ability to build and send mail.

For information on how to design a JavaMail program, see the JavaMail Web site at http://java.sun.com/products/javamail. For information on many of the standards relating to Internet mail, see the Internet Mail Consortium Web site at http://www.imc.org.

Writing JavaMail for EAServer

You can implement JavaMail for EAServer as you would for any other server that follows the JavaMail specification. JavaMail for EAServer can be coded to the standard JavaMail API and uses classes in the javax.mail and javax.mail.internet packages.

Creating a JavaMail session

The javax.mail.Session object is responsible for managing a user's mail configuration settings and handling authentication for the individual transports used during the session.

To create platform-independent applications, a JavaMail program can use a resource factory reference to obtain a JavaMail session. A resource factory is an object that provides access to specific resources within a program's deployed environment using the specific naming conventions defined by JNDI. All resource factory references are organized by resource type in the application's component environment. For example, JavaMail resource factory references are found in *java:comp/env/mail*. For more information on using resource factory references, see:

- "Configuring resource references" on page 133, which describes resource references used in EJB components.
- "Resource references" on page 385, which describes resource references used in Web applications.

To obtain an initial JNDI naming context for your JavaMail session, create an instance of the javax.naming.InitialContext object. Then call the lookup method to invoke the javax.mail.Session factory reference to obtain a JavaMail session. This session will map to the local mail server as defined for the environment in which your JavaMail program is deployed. See "Deploying JavaMail-enabled applications" on page 635 for information on specifying your local resources.

Constructing a message

Message is an abstract class in the JavaMail API. Subclasses of Message implement the concrete functionality needed for specific messaging systems. The JavaMail reference implementation includes a MimeMessage class that implements the standard for basic Internet messages and the Multipurpose Internet Mail Extensions.

To construct a message, instantiate a MimeMessage object, set the required attributes (headers), and provide the appropriate header values and body content. At a minimum, you should specify the From, To, and Date headers.

Use the setFrom method to set the From header field using the value of InternetAddress. Use the setRecipients method to set the specified recipient type to a given address. Use the setSentDate method to set the date.

Sending a message

You use the Transport class to send a message. If you create a JavaMail session that uses the SMTP provider included with EAServer, you can simply use the Transport.send method to send your completed message to all the recipient addresses specified.

Sample EAServer JavaMail program

In this example, an e-mail message is sent to the user of a Web-based travel reservation system confirming the user's reservation.

```
public String mailIt
   (java.lang.String from,
   java.lang.String to,
   java.lang.String subject,
   java.lang.String textmessage)
{
   String status = "Your message was sent";
   try {

   //Obtain the initial JNDI context
   InitialContext ctx = new InitialContext();

   //Perform a JNDI lookup to obtain the resource
   //reference object
   Session session = (Session) ctx.lookup
```

```
            ("java:comp/env/mail/mymailserver");

         //Construct the message
         MimeMessage message = new MimeMessage(session);

         //Set the from address
         Address[] fromAddress =
            InternetAddress.parse(from);
         message.addFrom(fromAddress);

         //Set the to address
         Address[] toAddress = InternetAddress.parse(to);
         message.setRecipients(Message.RecipientType.TO,
            toAddress);

         //Set the subject and text
         message.setSubject(subject);
         message.setText(textmessage);

         //Send the message
         Transport.send(message);

      } catch(AddressException e) {
      status = "There was an error parsing theaddresses"+e;
      } catch(SendFailedException e) {
      status = "There was an error sending the message"+e;
      } catch (MessagingException e) {
      status = "There was an unexpected error"+e;
      } catch (NamingException e) {
      status = "The mail session could not be created.";
      }
      System.out.println("The status is:"+ status);
      return status;
      }
```

JavaMail providers

JavaMail is extensible which means that when new protocols are developed, providers for those protocols can be added to a system and used by preexisting JavaMail enabled applications. Applications can detect which providers are available to them via the Provider Registry.

The providers that come with the JavaMail reference implementation are listed in *javamail.default.providers*. If you add a package containing a new provider, it should include a *javamail.providers* file in its *META-INF* directory.

To list the available providers on your system:

```
import javax.mail.*;
class ListProviders
{
   public static void main(String[] args)
   {
java.util.Properties properties =
   System.getProperties();
Session session = Session.getInstance(properties,
   null);
Provider[] providers = session.getProviders();
for (int i = 0; i < providers.length; ++i)
   {
   System.out.println(providers[i]);
   }
}
```

Deploying JavaMail-enabled applications

If you use JavaMail in Web applications or EJB components, you can configure resource references to alias a JavaMail session to a JNDI name. The resource reference allows you to use JNDI to obtain mail sessions, as described in "Creating a JavaMail session" on page 632. The use of logical names allows your application to run in environments where the JNDI namespace does not match the names hard-coded in your application. When you deploy the application, you map the logical names to actual names that match the server's configuration. You must catalog the JNDI names used by your code in the application's deployment descriptor. Once your JavaMail-enabled Web application is deployed to a host server, you must configure the javax.mail.Session resource factory reference to the name of the local mail server for that server.

❖ **To define the local mail server for a JavaMail program:**

1 In EAServer Manager, open the Properties dialog for the Web application that includes the servlet, EJB, or application that contains the JavaMail program.

2 Select the Resource Refs tab.

3 Click Add to add a row to the table.

4 Select javax.mail.Session from the dropdown list in the Type column.

5 The resource reference name in the Name column should be the logical name that refers to the JavaMail resource object and is hard-coded in the JavaMail code. For example, Mail.

6 In the Deployment Settings field, type in the name of your local SMTP mail server for outgoing mail.

7 Provide a description of your JavaMail resource in the Description field.

8 Click OK.

CHAPTER 36

Configuring Java XML Parser Support

About JAXP

EAServer 4.0 includes support for JAXP (Java API for XML Parsing) 1.1. The package includes the industry-standard DOM and SAX APIs, Crimson SAX and DOM parsers, and the Xalan XSLT transformer from Apache.

- JAXP is an API that provides basic functionality for reading, manipulating, and generating XML documents and, depending on the needs of the application, gives developers the flexibility to swap between parsers without making code changes.
- SAX (Simple API for XML) is an event-driven parser that invokes one of several methods supplied by the caller when a ‘parsing event’ occurs. “Events” include recognizing an XML tag, finding an error, encountering a reference to an external entity, or processing a DTD specification. It is used by many servlets and network-oriented programs because it is the fastest and least memory-intrusive mechanism currently available for dealing with XML documents.
- DOM (Document Object Model) parser is a tree-like structure, where each node contains one of the elements from an XML structure. The tree is traversed to dynamically access and update the content, structure, and style of a document, which can then be incorporated back into the presented document page.
- XSLT is a language for transforming one XML document into another XML document through the use of a formatting vocabulary.

You can find programming examples for each of these APIs in Sun's *The Java/XML Tutorial*, available at http://java.sun.com/xml/tutorial_intro.html.

Configuring JAXP properties in EAServer Manager

JAXP provides a plugin model for XML parser classes. You can configure the parser implementation used by your application code. A JAXP 1.1 properties window will appear in each of the following EAServer Manager configuration modules:

- Server Properties
- Application Properties
- Web Application Properties
- Package Properties
- Component Properties
- Application Client Properties

The JAXP properties window allows you to choose which JAXP factory implementation to use for the SAXParserFactory, DocumentBuilderFactory, and TransformerFactory. For each of these factories, you can choose from the following options:

- **Platform Default** The default parsers are used and the entities custom class list is configured to load them. EAServer Manager adds the following JAR files to the entity's custom Java class list (displayed on the Java Classes tab in the Properties dialog for the entity):
  - *jaxp.jar*
  - *crimson.jar* (for SAXParserFactory or DocumentBuilderFactory)
  - *xalan.jar* (for TransformerFactory)
- **Not Configured** When this setting is in effect, the parsers configured in the server's CLASSPATH and BOOTCLASSPATH setting are used. The parser classes are the same as for the Platform Default setting, but the classes are loaded by the Java system class loader.

You must use the Not Configured option to prevent overriding JAXP settings in child entities. For example, to use the Custom option in a component, the server, application, and package that contain the component must use the Not Configured option. If all entities in the hierarchy use the Not Configured option, JAXP classes are loaded by the system class loader.

- **Custom** Allows you to specify a class name in the JAR file containing the specified class. The JAR file must be in the *WEB-INF/lib* directory for Web applications, or in the EAServer *java/classes* directory for all other entity types.

  EAServer Manager sets the entity's Factory property to the class name, and adds *jaxp.jar* and the JAR file you specified to the entity's custom class list.

Precedence of JAXP properties
The parser configuration at the highest level has precedence. For example, if you configure a parser at the server level, the server setting specifies the parser used in all components and Web applications running on that server. To prevent overriding the settings of child entities, specify the Not Configured setting in parent entities.

Exporting and importing application clients

When exporting an application client, the exporter examines the three Factory properties for that application client and creates the following entries in the exported JAR file:

- META-INF/services/javax.xml.parsers.SAXParserFactory

  The content of this file is the value of the entity's SAXFactory property if it is not empty.

- META-INF/services/javax.xml.parsers.DocumentBuilderFactory

  The content of this file is the value of the entity's DOMFactory property if it is not empty.

- META-INF/services/javax.xml.transform.TransformerFactory

  The content of this file is the value of the entity's XSLTFactory property if it is not empty.

When importing an application client, the importer looks at these same three entries. If these entries are not empty, the importer sets up the corresponding properties for the application client.

APPENDIXES

APPENDIX A

Executing Methods As Stored Procedures

The EAServer Methods As Stored Procedures (MASP) interface allows component methods to be executed as if they were database stored procedures.

The MASP interface allows you to invoke EAServer component methods from any front-end tool that can execute Sybase Adaptive Server Enterprise stored procedures. Server component developers can also use isql and the MASP interface to quickly test methods.

Creating invocation commands

Each method call requires a Transact-SQL® exec command that specifies the component name, the method name, and parameter values, as in:

```
exec MyPackage.MyComponent.MyMethod param1, param2,
...
```

You can also include the EAServer name in the invocation, as in:

```
exec MyServer.MyPackage.MyComponent.MyMethod
param1, param2, ...
```

The method call can return return values (including multiple result sets) and inout and out parameter values just like a stored procedure. MASP only supports primitive types, that is, the types listed in the pulldown menu when adding or modifying method parameters in EAServer Manager.

If the component method returns a value and not void, TabularResults::ResultSet, or TabularResults::ResultSets, the method's return value is returned as the first output parameter in the stored procedure results. In this case, you must define a dummy output parameter to receive the method's return value. Use this variable in place of the first stored procedure parameter in the request. For example, in a Client-Library client application, you would need to implement a ct_param() call for the return value.

In tools that allow exec commands to be batched, you can send several invocations in the same batch, as in:

```
exec Pkg1.Comp1.Meth1
exec Pkg1.Comp2.Meth2
exec Pkg1.Comp2.Meth3
```

Some tools may require the parts of the stored procedure name to be entered separately. In this case, the mappings are as follows:

Stored procedure name part	MASP equivalent
Remote server name	The server name
Database	Package name
Owner	Component name
Procedure name	Method name

Limitations

Components that save state between method calls using the continueWork and disallowCommit primitives will not work as expected. (See Chapter 2, "Understanding Transactions and Component Lifecycles" for more information on state primitives).

Each MASP method invocation creates a component instance, invokes the method, and then destroys the component instance. Since every MASP invocation creates a new instance to invoke a method on, you cannot set an instance's state with consecutive MASP invocations. For these reasons, there is no practical support for calling stateful components or EJB entity Beans from MASP.

MASP clients can call EJB stateless session Beans, as long as the Bean's home interface has a create method with no parameters.

MASP clients cannot call component methods named invoke.

Due to ODBC driver limitations, string or binary values sent to a client that connects through ODBC cannot be greater than 255 bytes in length. This applies to both inout parameters and to columns in a result set.

Using MASP from isql

Using the MASP interface, you can use isql to quickly test methods. For example, the following command invokes the getMajors method in the SVUEnrollment component in the SVU package:

```
1> exec SVU.SVUEnrollment.getMajors
2> go

 --- ----------------
 Eng English
 Phy Physics
 Ant Anthropology

           0
```

If you are using isql with a multi-byte character set, you should specify that character set using the isql -J option. For example, the following command sets the isql codeset to the sjis codeset.

```
isql -Usa -P -Sjaguar -Jsjis
```

Because EAServer does not support languages, the isql -z option is not valid nor can you set the client machine's LANG environment variable to a language other than US English.

Note This section applies to any client that sends a MASP request as a Transact-SQL command.

Using MASP from application builder tools

Using MASP, you can invoke EAServer component methods that can execute Sybase stored procedures.

Note Some application builder tools might not be able to use MASP if they issue metadata queries against EAServer.

PowerBuilder

You can connect to EAServer, create a stored procedure DataWindow, and get a list of available methods. You can pick one of these, create a DataWindow with it, and then when you execute the DataWindow, results from the method are displayed in the window.

Note that you must manually specify the format of the result sets and the expected parameters. As with all stored procedure DataWindows, these are read-only.

PowerDynamo

You can execute methods on an EAServer component as if they were stored procedures, and use the script's capabilities of dynamic table generation to display the results in an HTML table.

Other tools

You should be able to use MASP in other tool environments as well. Any tool that uses one of the following communication drivers and allows stored procedure execution should be able to use MASP:

- Sybase jConnect (any version)
- Sybase Adaptive Server Enterprise ODBC driver
- Sybase Open Client

Configuring the return status

The stored procedure return status of the MASP call reflects the status of the call. By default, the following status values are returned:

- A value of 1 indicates the method was invoked
- A value of 0 indicates that EAServer was not able to invoke the method. This error can occur for a variety of reasons, such as:
  - The syntax was incorrect
  - The specified method, component, or package that did not exist
  - The specified package was not installed on the server, or the user lacked permission to invoke the method

  When a MASP call fails, see the server's log file for more information on the cause.

You can configure your server to reverse the meanings of the status value by setting the server property `com.sybase.jaguar.server.masp.zero-success`. A value of `false`, the default, indicates the status values have the meanings described above. A value of `true` indicates that the meanings of 0 and 1 are reversed.

Set the property in EAServer Manager as follows:

1 Display the properties for the server by right-clicking on the icon and selecting Server Properties from the popup menu.

2 Click on the Advanced tab

3 If `com.sybase.jaguar.server.masp.zero-success` is displayed, highlight it and click Modify. Edit the displayed value and click Ok.

4 If `com.sybase.jaguar.server.masp.zero-success` is not displayed, click Add. Enter the property name and the value, then click Ok.

APPENDIX B

Migrating Open Server Applications to EAServer

The current Open Server is based on proprietary light-weight thread architecture and does not scale to symmetric multiprocessor (SMP) platforms. EAServer is based on native or kernel threads supporting SMP architecture. Migrating to EAServer requires minimal work on the server side, no changes to the client, and allows your Open Server applications to take advantage of many EAServer features.

Related documentation

Refer to the Sybase Open Server documentation for information on the Open Server API. This documentation is available on the Sybase Web site at http://www.sybase.com.

Migration overview

Migrating your Open Server applications allows you to take advantage of EAServer features such as:

- High scalability and high performance engine – EAServer is based on kernel threads and supports symmetric multiprocessors (SMP)
- Built in worker thread and thread pooling facilities supports large concurrent client loads
- Ability for applications to make blocking or synchronous calls that do not block the entire process
- Operating system based authentication services

- EAServer Manager, an easy-to-use graphical user interface for:
  - Installing and configuring event handlers
  - Managing the server
  - Monitoring the server
  - Accessing Open Client and ODBC connection cache facilities

To migrate an Open Server application to EAServer, the application must be built as a shared object or a DLL (on Windows) instead of a binary. The shared object contains all of the Open Server event handler code currently residing in the application code.

To migrate your Open Server applications to EAServer:

1 Modify your Open Server code to run in EAServer by:
   - Removing your existing code from the main function and placing it in an event handler and removing the srv_run and other routines. Refer to "Modifying main" on page 651 for more information.
   - Modifying your application so that it runs in a preemptive scheduling environment. Refer to "Coding changes and examples" on page 650 for coding examples and guidelines.

2 Once you have modified your code, create and build one or more DLLs or shared objects consisting of your event handlers. Refer to "DLLs, shared objects, and makefiles" on page 657 for more information.

3 Using EAServer Manager:
   - Install your event handlers. Refer to "Installing event handlers" on page 663.
   - Configure an EAServer listener to accept client requests. Refer to "Configuring an Open Server listener" on page 664.

Limitations

EAServer does not support DCE or Kerberos.

Coding changes and examples

The code changes required for your Open Server applications include:

- Moving the main code to an event handler and removing some routines.
- Modifying your code so that it is thread-safe.

Modifying main

You must move Open Server application code currently running in the main routine to one or more event handlers. Other routines, such as the srv_run routine are also removed. You must also remove routines that EAServer automatically initiates, and remove properties and handlers that are configured through EAServer Manager.

Traditional Open Server application

The following file is a traditional Open Server application that contains a main routine:

```
#include <ospublic.h>
#include <server.h>

/*
** File server.c containing a typical Open Server main()
function. For
** simplicity, there is no error handling here.
**
** This builds into an executable, with the supporting
code, such as
** event handlers, ending up as either static libraries
that become part
** of the executable, or as dynamic libraries loaded at
run time.
*/
main(int argc, char *argv[])
{
       /*
       ** Variables.
       */
       CS_INT conns;
       CS_INT threads;
       CS_CHAR *name;
       CS_CONTEXT *context;

       /*
       ** Process command line. Function get_params() is
in file appl.c.
       */
       get_params(argc, argv, &conns, &threads, &name);
       if (name == (CS_CHAR *) NULL)
       {
           printf("Usage: %s [-os_conns=<conns>] [-
os_threads=<threads>] "
                  "-os_name=<name>\n", argv[0]);
           exit(1);
```

```
        }
        /*
        ** Initialize server.
        */
        cs_ctx_alloc(CS_VERSION_100, &context);
        srv_version(context, CS_VERSION_100);
        if (conns > 0)
            srv_props(context, CS_SET,
SRV_S_NUMCONNECTIONS,
                    (CS_VOID *) &conns, sizeof(conns),
(CS_INT *) NULL);
        if (threads > 0)
            srv_props(context, CS_SET, SRV_S_NUMTHREADS,
                  (CS_VOID *) &threads, sizeof(threads),
(CS_INT *) NULL)
        srv_init((SRV_CONFIG *) NULL, name,
CS_NULLTERM);

        /*
        ** Register handlers. Files handler1.c and
handler2.c contain these
        ** functions.
        */
        srv_handle((SRV_SERVER *) NULL, SRV_START,
start_handler);
        srv_handle((SRV_SERVER *) NULL, SRV_ATTENTION,
attn_handler);
        srv_handle((SRV_SERVER *) NULL, SRV_BULK,
bulk_handler);
        srv_handle((SRV_SERVER *) NULL, SRV_CONNECT,
conn_handler);
        srv_handle((SRV_SERVER *) NULL, SRV_CURSOR,
cur_handler);
        srv_handle((SRV_SERVER *) NULL, SRV_DISCONNECT,
disc_handler);
        srv_handle((SRV_SERVER *) NULL, SRV_DYNAMIC,
dyn_handler);
        srv_handle((SRV_SERVER *) NULL, SRV_LANGUAGE,
lang_handler);
        srv_handle((SRV_SERVER *) NULL, SRV_RPC,
rpc_handler);
        srv_handle((SRV_SERVER *) NULL, SRV_OPTION,
opt_handler);

        /*
        ** Start server.
```

```
        */
        srv_run((SRV_SERVER *) NULL);
        exit(0);
}
```

Moving main() to an event handler

The main code has been placed in an event handler, other routines and properties have been removed:

```
#include <ospublic.h>
#include <server.h>

/*
** File server.c. The original main() function becomes
init_handler().
**
** Build this into a dynamic library, and register the
** init_handler() function in EAServer Manager.
EAServer
** will call the function at runtime.
**
** You may build the supporting code into the same
dynamic library, or into
** one or more different dynamic libraries. In either
case, you'll need to
** register each handler separately with EAServer, using
** EAServer Manager.
*/

CS_RETCODE CS_PUBLIC init_handler(CS_CONTEXT *context,
        int argc, char *argv[])
{
        /*
        ** Variables.
        **
        ** EAServer initializes context and passes it to
this function.
        */
        CS_INT conns;
        CS_INT threads;
        CS_CHAR *name;

        /*
        ** Process command line. Function get_params() is
in file appl.c.
        **
        ** Do not exit on error.
        */
        get_params(argc, argv, &conns, &threads, &name);
```

```
        /*
        ** Initialize server.
        **
        ** Get rid of cs_ctx_alloc(), srv_version(), etc.
Do not call srv_init().
        ** Certain properties previously set using
srv_props() are now set
        ** in EAServer Manager.
        */
        if (conns > 0)
            srv_props(context, CS_SET,
SRV_S_NUMCONNECTIONS,
                      (CS_VOID *) &conns, sizeof(conns),
(CS_INT *) NULL);
        if (threads > 0)
            srv_props(context, CS_SET, SRV_S_NUMTHREADS,
                  (CS_VOID *) &threads, sizeof(threads),
(CS_INT *) NULL);

        /*
        ** Register handlers. Files handler1.c and
handler2.c contain handler
        ** functions.
        **
        ** Register all the handlers using EAServer
Manager.
        */

        /*
        ** Start server.
        **
        ** EAServer calls srv_run(); Do not call it
yourself.
        */

        /*
        ** Return rather than exit.
        */
        return CS_SUCCEED;
}
```

Open Server properties

Do not attempt to set or reset the SRV_S_PREEMPT property.

Do not set the SRV_S_STACKSIZE property.

You must set the following properties using EAServer Manager:

- SRV_S_NUMCONNECTIONS
- SRV_S_NETBUFSIZE
- SRV_S_MSGPOOL
- SRV_S_NUMMSGQUEUES

Do not set these properties using srv_props() calls.

To set these properties from EAServer Manager:

1 Highlight the server whose properties you want to set.

2 Select File | Server properties.

3 Select the Resources tab and set the properties.

Refer to Chapter 3, "Creating and Configuring Servers," in the *EAServer System Administration Guide* for more information.

Making your code thread-safe

Open Server uses it's own implementation of threads with non-preemptive scheduling. EAServer uses native operating system threads with preemptive scheduling. Context switches were predictable in Open Server's non-preemptive environment. EAServer does not support non-preemptive scheduling. As a result, context switches are unpredictable and managed by the operating system's thread management facility. You must modify your Open Server application so that when a context switch does occur, your resources (variables, data, and so on) are maintained and not overwritten by another thread.

Protecting data

In a multi-threaded program, it is possible for multiple "threads" of control to be active concurrently. These threads can overwrite each other's data, resulting in unpredictable behavior such as race conditions.

Any data shared across threads is vulnerable to race conditions:

- Global variables – such as "errno" in C programs or user-defined global variables. Since any thread can access these global variables, one thread may inadvertently overwrite the value of a variable accessed by a different thread.

- Static variables – a C function containing a static variable can be executed concurrently by multiple threads, resulting in unpredictable changes to the variable's contents.

  Automatic variables are created on the stack. Since each thread typically has its own stack, these variables are generally immune to context switches.

Consider the following code segment running in a traditional Open Server application:

```
static int x = 0;
x += 100;

if (x > 1000)

x = 0;
```

A race condition cannot occur when this code runs because a context switch cannot occur with non-preemptive scheduling. However, when you move to EAServer, the result of the execution of this code is unpredictable.

You can protect the previous code with an Open Server mutex:

```
static int x = 0;
srv_lockmutex(mutex);
x += 100;

if (x > 1000)

x = 0;
srv_unlockmutex(mutex);
```

Identify sections of code that need explicit protection and protect them with any applicable synchronization mechanism such as Open Server mutexes, and test your software (for deadlocks, etc.).

Tools for protecting data

Solaris users may find tools in the SPARCworks/iMPact multithreaded development kit from SunSoft useful in analyzing your code to identify critical sections that need protection, and help test the software after protecting these sections. This kit contains tools such as LockLint, LoopTool, SPARCworks Debugger, and Thread Analyzer.

DLLs, shared objects, and makefiles

This section contains two sample makefiles. The first is an example of a traditional Open Server makefile for Solaris, which contains routines, such as main and srv_run, used to build an executable. The second makefile is used to build dynamic libraries for use with EAServer. The main() logic has been moved to an init handler.

In addition to the sample makefiles there are instructions for building shared objects for UNIX and DLLs for Windows.

Traditional Open Server makefile for Solaris

```
# A makefile that builds the Open Server executable,
server
# server.exe on Windows.

#

# The main() function is in server.c.

#

# The two files, handler1.c and handler2.c implement all
the handlers, and
# compile into a static library, handler.a (handler.lib
on Windows).

#

# All the remaining code is in files appl1.c and
appl2.c, and compiles into a
# static library, appl.a (appl.lib on Windows).

#

CFLAGS=       -I$(SYBASE)/include -I.

LIBS=         -lsrv -lblk -lct -ltcl -lcs -lcomn -lintl
-lm -lnsl -ldl

LDFLAGS=      -L$(SYBASE)/lib $(LIBS)


server:       server.o handler.a appl.a

              $(LINK.c) -o $@ server.o handler.a appl.a


handler.a:    handler1.o handler2.o

              $(AR) $(ARFLAGS) $@ handler1.o handler2.o
```

```
appl.a:       appl.o
              $(AR) $(ARFLAGS) $@ appl.o


.c.o:
              $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Makefile for EAServer

```
# Build a dynamic library, server.so (server.dll on
Windows), instead of an
# executable.
#There is no main() function anymore: the applicable
main() logic is in
# the init handler function. The file is still server.c.
#
# The two files, handler1.c and handler2.c, implement
all the handlers and
# compile into a dynamic library, handler.so
(handler.dll on Windows).
# Register each handler in EAServer Manager.
#
# All the remaining code is in files appl.c, and
compiles into a
# static library, appl.a (appl.lib on Windows). No
change here.
#No need to link with libraries.
CFLAGS=          -I$(JAGUAR)/include -I.

server.so:       server.o handler.so appl.a
                 $(LINK.c) -G -o $@ server.o handler.so
appl.a

handler.so:      handler1.o handler2.o
                 $(LINK.c) -G -o $@ handler1.o
handler2.o

appl.a:          appl.o
                 $(AR) $(ARFLAGS) $@ appl.o

.c.o:
                 $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Building shared objects on Solaris

When building shared objects on Solaris compile all the modules with the compile switches -KPIC -mt

The following link line should be used when creating a shared object on Solaris. Note that EAServer libraries have a "j" prefix in them, for example, *libjsrv_r.so*:

```
 ld -g -o <shared object name> <object files> -ljsrv_r
-ljct_r -ljcs_r\
-ljtcl_r -ljcomn_r -ljintl_r -ljtml_r -Bdynamic -lnsl -
ldl -lthread -lm
```

Building DLLs on Windows

When building DLLs on Windows use the compile flags:

```
CFLAGS = /W3 /MD /nologo /Z7 /Od /DWIN32 /Gz
```

You need to export all handler functions. Use a *.def* file for this purpose and specify this *.def* file /def link option. For example:

```
# Definition file dependencies
LIBRARY sample     INITINSTANCE
DESCRIPTION  'EAServer event Handler'
HEAPSIZE      22000
PROTMODE
CODE LOADONCALL EXECUTEREAD NONCONFORMING
DATA PRELOAD READWRITE MULTIPLE NONSHARED
EXPORTS
connect_handler
```

Note To run a server using an event-handler DLL, the directory containing the DLL must be specified in the PATH environment variable.

Modified APIs and new event handlers

This section discusses the modified Open Server APIs and two new event handlers.

Modified APIs

The modified Open Server APIs are:

- srv_sleep
- srv_props
- srv_deletemutex
- srv_droppproc

srv_sleep

srv_sleep is used to suspend the currently executing thread. srv_sleep now uses its final two parameters which had been reserved for future use. If you do not want to take advantage of the new functionality of srv_sleep no changes are required and your code will work as before.

The first reserved parameter is now an Open Server mutex, and the second is a time-out in milliseconds. Both of these parameters are optional, and should be set to (CS_VOID*)0 when not being used.

When a mutex is passed to srv_sleep, it is released before the thread suspends, but after it is marked suspended. The mutex is reacquired before srv_sleep returns. The behavior is modeled on condition variables (posix threads).

This pseduo-code fragment demonstrates how a multithreaded application may use the new sleep functionality to prevent race conditions where wakeups may be missed:

```
Sleep side:

srv_lockmutex(mutex_id)
status = NOT_YET_DONE;
while (status != DONE)
{
srv_sleep(..., mutex_id, ...);
}
srv_unlockmutex(mutex_id)
Wakeup side:

srv_lockmutex(mutex_id)
status = DONE;
srv_wakeup(...)
srv_unlockmutex(mutex_id)
```

If a srv_sleep call returns because of a time-out, the location pointed to by the infop parameter will be set to SRV_I_TIMEOUT. In this case, as in all others, the mutex is reacquired before the srv_sleep call returns.

The normal usage of srv_sleep in the native threads version should be within a loop checking the predicate. For example:

```
mutex_lock()
        while (work != DONE)
```

```
        {
                srv_sleep(...);
        }
    mutex_unlock()
```

In this usage, the time-out is not useful because it is reset each time.

srv_props

The property SRV_S_NATIVEMUTEX has been added to srv_props(). You can set SRV_S_NATIVEMUTEX to CS_TRUE or CS_FALSE. The default is CS_FALSE. If set to CS_TRUE, the srv_createmutex, srv_lockmutex, srv_unlockmutex, and srv_deletemutex APIs use native (operating system) mutexes. If set to CS_FALSE, these APIs use Open Server mutexes.

Mutex operations will be faster if you set the SRV_S_NATIVEMUTEX property to CS_TRUE because mutex operations will map almost directly to operating system mutex operations. Currently, Open Server allows mutexes to be referred to by name. This requires that mutexes and their names be stored in tables. When you set SRV_S_NATIVEMUTEX property to CS_FALSE, mutex operations require table lookups; these table lookups must be synchronized with other threads to ensure that the mutex table does not become corrupt.

Mutexes created when SRV_S_NATIVEMUTEX is set to CS_FALSE support recursive locking.

Mutexes created when SRV_S_NATIVEMUTEX is set to CS_TRUE do not support recursive locking. If a mutex is locked, any attempt to lock it a second time, even by the same thread that originally locked it, will block the thread.

srv_deletemutex

srv_deletemutex() has been modified so that only an unlocked mutex can be deleted.

srv_dropproc

srv_dropproc() has been modified so that asynchronous or involuntary thread terminations are not allowed.

Event handler prototypes

EAServer supports two additional event handlers, “Initialization” and “Error”. The initialization handler (or init handler) is used to perform any customization the application requires. Ideally, all of your main() code goes into the init handler. The prototype for all the existing open server event handler remains the same.

Initialization handler prototype

```
typedef CS_RETCODE  (CS_PUBLIC * SRV_INITHANDLE_FUNC)
      PROTOTYPE((
         CS_CONTEXT  *context,
         CS_INT      argc,
```

```
        CS_CHAR     **argv
        ));
```

context – is pointer to the CS_CONTEXT structure.

Initialization handlers must return CS_SUCCEED unless an error occurs that prevents the application from running successfully. Returning a value other than CS_SUCCEED aborts the server startup sequence.

You can use the initialization handler or the run handler to initialize your application's global resources, or you can install handlers for both events. The server log file is not open when the initialization handler is called. If you need to write messages to the log (using JagLog), use a start handler rather than the initialization handler.

Error handler prototype

Install error handlers through EAServer Manager and not srv_props() or srv_errhandle(). Refer to "Installing event handlers" on page 663 for more information.

```
typedef CS_RETCODE  (CS_PUBLIC * SRV_ERRORHANDLE_FUNC)
       PROTOTYPE((
            CS_CONTEXT  *context,
            CS_VOID     *argp,
            CS_INT      where,
            CS_SERVERMSG    *msg
            ));
```

Refer to "Additional event handler information" on page 664 for information about all event handlers.

EAServer configuration

After you have modified your Open Server application and have built your DLLs or shared objects, you need to register your event handlers and configure an Open Server port using EAServer Manager. This section discusses:

Installing event handlers

EAServer fully supports all Open Server event handlers. The only difference is that instead of creating a binary file linking the DLL or shared object, you create a DLL or shared object consisting of your event handlers and then specify the location of this file using EAServer Manager (instead of specifying them in the code).

Specifying event handlers

To specify an event handler from EAServer Manager:

1 Double-click the Servers folder.

2 Highlight the server for which you are specifying the event handler.

3 Select File | Server Properties.

4 Select the Handlers tab.

5 Enter the DLL or shared library name and the function name of the specific event handler being called, separated by a colon.

The following examples illustrate an entry for a connect event handler for Solaris and Windows:

- Solaris

  libsamp.so:debug_connect

  where libsamp.so is the shared library name and *debug_connect* is the function called whenever a connect event handler is called.

- Windows

  libsamp.dll:*debug_connect*

  where *libsamp.dll* is the DLL name and *debug_connect* is the function called whenever a connect event handler is called.

The following table summarizes the types of event handlers that you can install. For information on coding event handlers, refer to“Additional event handler information” on page 664and your Open Server documentation.

Table B-1: Individual server event handlers

Event handler	Called
Connect	Each time a client connects to EAServer
Disconnect	When the client disconnects from EAServer
Error	When an Open Server processing error occurs
Initialization	Before starting EAServer
Start	After initialization, but before accepting client requests
Stop	When a request to stop the server is made
Language	When a client sends a language request, such as a SQL statement
RPC	When a client issues a remote procedure call
Attention	When an attention had been received. An attention is an immediate event; EAServer services the attention as soon as it occurs, rather than adding it to the client's event queue
Cursor	When a client sends a cursor request
Dynamic	When a client sends a dynamic SQL request
Message	When the client sends a message
Option	When a client sends an option command
Bulk	When a client issues a bulk copy request

Configuring an Open Server listener

To support Open Server clients, you must install a listener in your server that supports Open Server clients. The listener must use protocol TDS and the "Enable Open Server Events" option must be enabled. If you are using the preconfigured Jaguar server, the Jaguar_OpenServer listener supports Open Server connections. Refer to Chapter 3, "Creating and Configuring Servers," in the *EAServer System Administration Guide* for more information about listeners.

Additional event handler information

This section contains additional information relevant to coding event handlers.

Calling convention for event handlers

All event handlers, error handlers, and any other function that is installed as a callback in the EAServer runtime must be coded according to the following rules:

- The function must use the C link-object naming convention. C++ programmers must declare EAServer callbacks in an extern C block. You will get link errors otherwise.
- Callback prototypes must include the CS_PUBLIC macro as shown by the examples in this chapter. On platforms such as Windows, the C compiler supports a broad collection of calling conventions for C functions. On these platforms, the CS_PUBLIC macro encapsulates the appropriate compiler keywords to ensure that the same calling convention is used by both EAServer and your callbacks. If the calling convention used by your callback and the calling convention used by EAServer do not match, the server will probably crash when the callback is called or soon after it returns.

Warning! Declare your callback functions with the CS_PUBLIC macro to avoid server crashes.

Initialization, run, start and exit events

An application's initialization handler and start handler are invoked when the server starts up. The exit handler is invoked when the server shuts down. Initialization and exit handlers are typically used to manage global resources used by the application. The sequence is as follows:

1 Server initialization – Initialization handler (if installed) is called.

2 Server start-up – Initialization handler has returned. The Start handler is called. The server is now ready to spawn new threads, but will not accept client connections until after the Start handler returns. The Start handler can spawn service (non-client) threads if necessary.

3 Normal operation – The server accepts client connections and associates each with a thread, spawning new threads when necessary. Each time a client connects, the server calls the application's connect handler. Each time a client disconnects, the server calls the application's disconnect handler.

4 Server shutdown – The server terminates all threads, then calls the exit handler.

Start handler template

The template for a start handler is:

```
#include <ospublic.h>

CS_RETCODE CS_PUBLIC start_handler (
                            CS_CONTEXT *ctx
                            )
```

where

context – is pointer to the CS_CONTEXT structure.

Start handlers must return CS_SUCCEED unless an error occurs that prevents the application from running successfully. Returning a value other than CS_SUCCEED aborts the server start-up sequence.

Your application does not require a start handler unless you want to use service threads or create global mutexes. In this case, the service threads must be created in the start handler.

Exit handler template

The template for an exit handler is:

```
#include <ospublic.h>

CS_RETCODE CS_PUBLIC exit_handler(
              CS_CONTEXT      *context
             )
{
    ... your code goes here ...
    return CS_SUCCEED;
}
```

where

context – is pointer to the CS_CONTEXT structure.

Exit handlers must return CS_SUCCEED.

Connect and disconnect handlers

Connect and disconnect handlers are invoked when client applications open and close connections to EAServer.

The connect and disconnect handler examples in this chapter call Server-Library routines. You can use these routines to perform user authentication.

You can find documentation for these routines in the *Open Server Server-Library/C Reference Manual*. You can view it on the web from the Technical Library page:

```
http://sybooks.sybase.com
```

Connect handler

The connect handler can be used to authenticate the connection's user name. Users must be validated based on user name/password, and you must supply code for password maintenance and checking.

Note srv_thread_props(SRV_T_USERDATA) is off-limits to EAServer programmers.

The following is an example of a connect handler that logs the user name, password, and locale name when a connection is opened:

```
CS_RETCODE CS_PUBLIC
debug_connect(srvproc)
SRV_PROC    *srvproc;
{
    CS_INT      spid;
    CS_INT      ulen;
    CS_INT      plen;
    CS_INT      llen;
    CS_CHAR     msg[CS_MAX_MSG];
    CS_CHAR     user[CS_MAX_NAME+1];
    CS_CHAR     password[CS_MAX_NAME+1];
    /* Initialization                              */
    spid = 0;
    /* Get the spid    */
    if (srv_thread_props(srvproc, CS_GET, SRV_T_SPID,
        (CS_VOID *)&spid,
        CS_SIZEOF(spid), NIL(CS_INT *)) != CS_SUCCEED)
    {
        return (CS_FAIL);
    }
```

```
    /*
    ** Get the username and password
    */
    if (srv_thread_props(srvproc, CS_GET, SRV_T_USER,
        (CS_VOID *)user,
        CS_MAX_NAME, &ulen) != CS_SUCCEED)
    {
        return (CS_FAIL);
    }
    if (srv_thread_props(srvproc, CS_GET, SRV_T_PWD,
           (CS_VOID *)password,
           CS_MAX_NAME, &plen) != CS_SUCCEED)
    {
        return (CS_FAIL);
    }
    /* Null terminate the username and password      *
/
    user[ulen] = (CS_CHAR)'\0';
    password[plen] = (CS_CHAR)'\0';
    /* Log the username and password values.    */
    sprintf(msg,"SPID %d) user '%s', password '%s'\n",
            spid, user, password);
    SRV_LOG(CS_TRUE, msg, CS_NULLTERM);
    return (CS_SUCCEED);
}
```

Disconnect handler

The following is an example of a disconnect handler:

```
CS_RETCODE CS_PUBLIC
fullpass_disconnect(srvproc)
SRV_PROC        *srvproc;
{
    CS_INT      spid;
    CS_CHAR     msg[CS_MAX_MSG];
    /* Initialization                           */
    spid = 0;
    /* Get the spid    */
    if (srv_thread_props(srvproc, CS_GET, SRV_T_SPID,
        (CS_VOID *)&spid,
        CS_SIZEOF(spid), NIL(CS_INT *)) != CS_SUCCEED)
    {
        return (CS_FAIL);
    }
    sprintf(msg,"SPID %d disconnected.\n", spid);
```

```
    SRV_LOG(CS_TRUE, msg, CS_NULLTERM);
    return CS_SUCCEED;
}
```

Build with the Visual C++ IDE

On Windows Platoforms, if you use the Visual C++ IDE or another command line compiler to build your DLL, make sure that you specify the correct options so that the compiler generates C functions using the standard C calling convention. After you build the DLL, copy it to the EAServer *dll* subdirectory.

A sample module definition (.def) file

EAServer Manager generates a *.def* file for your component. Visual C++ requires a module definition file that specifies which functions are exported from a DLL and some options that control how the DLL is loaded into memory. Module definition files end with the extension *.def*.

For most projects, you can use the generated file as-is. In some cases, you may want to edit settings other than those in the EXPORTS section. For example, your component may perform better with a smaller or larger HEAPSIZE setting.

Note Never edit the generated function names in the EXPORTS section of the *.def* file for a C component, otherwise, the EAServer dispatcher will not be able to call your methods.

Below is the example module definition file for the sample Enrollment component:

```
LIBRARY libEnrollment     INITINSTANCE
Description  'EnrollmentComponent - EAServer'
HEAPSIZE      22000
PROTMODE
CODE LOADONCALL EXECUTEREAD NONCONFORMING
DATA PRELOAD READWRITE MULTIPLE NONSHARED
EXPORTS
__skl_Enrollment_v_1_0_getMajorList
__skl_Enrollment_v_1_0_getCourses
__skl_Enrollment_v_1_0_getStudentRecord
__skl_Enrollment_v_1_0_putCourseRecord
```

```
__skl_Enrollment_v_1_0_destroy
__skl_Enrollment_v_1_0_createStudentRecord
__skl_Enrollment_v_1_0_create
__skl_Enrollment_v_1_0_getMajorEnrollmentRecord
__skl_Enrollment_v_1_0_removeCourseRecord
__skl_Enrollment_v_1_0_getCourseList
__skl_Enrollment_v_1_0_getAllEnrollmentRecord
```

For components, the *.def* file must use the mangled function names as shown in the example. For each method in your component, the mangled name is:

```
__skl_Comp_v_1_0_method
```

where

Comp is the component name.

method is the method name.

APPENDIX C

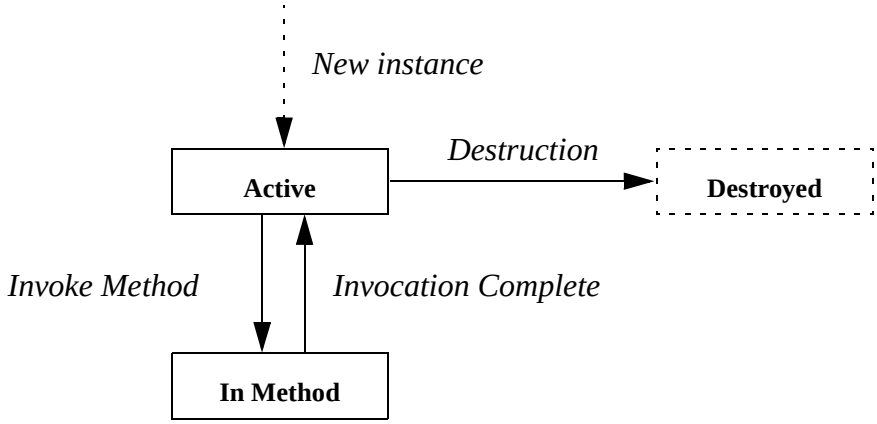
Creating C Components

C components provide a quasi-object model for the execution of a group of related C functions. Unlike a C++ object, separate instances of a C component lack a private data space. However, you can implement create and destroy methods to associate data with an instance of a C component.

C component lifecycle

This figure illustrates the states in the lifetime of a C component instance:

Figure C-1: C component lifecycle



The state transitions are:

- *New instance* – a new C component instance is created whenever a client application instantiates a stub for the component. At this point, the EAServer runtime calls the component's create routine (see "Customize the creation and destruction of components" on page 696).
- *In Method* – in response to a method invocation request from the client, the EAServer runtime calls the C routine that implements the method. The next state depends on which of the transaction state primitives is called.
  - If JagCompleteWork or JagRollbackWork is called, the instance is destroyed when the method invocation completes.
  - If JagContinueWork or JagDisallowCommit is called, the instance persists when the method invocation completes.

  If the component is transactional, the routine that is called also influences the outcome of the transaction that the component is participating in.

  Chapter 5, "C Routines Reference," in the *EAServer API Reference* contains reference pages for the C routines. Chapter 2, "Understanding Transactions and Component Lifecycles" describes EAServer's transaction model.

- *Destruction* – if the method called JagCompleteWork or JagRollbackWork, the instance is destroyed when the method completes. An instance is also destroyed when the client destroys its stub instance or if the client disconnects abruptly without explicitly destroying the stub.

Requirements

The following list describes the software requirements for developing C components and the hardware requirements for running C components. All software that is required to run C components in EAServer is supplied with the EAServer product.

- Development

  To create C components, you need a C or C++ development tool.

- Runtime

  For detailed system requirements, see the *EAServer Installation Guide* for your platform.

Procedure for creating C components

To create C components:

1 Define component interface and properties – using EAServer Manager, specify the component's name, DLL name, method prototypes, transactional semantics, and threading model.

2 Generate C component files – C component files are the C source files that are compiled into the C component. At this time, you generate UNIX and Windows makefiles as well as Visual C++ module definition files. You use UNIX and Windows makefiles to compile the C component files into C components.

3 Write C components – write the method logic in the method definition for the method implementation template files.

4 Compile C components – compile and link the method prototypes header file, the method implementation template files, and method skeletons file to create a dynamic link library (DLL) or UNIX shared library.

5 Install C Components – copy the DLL or shared library to the *cpplib* directory of the EAServer installation.

Define component interface and properties

The definition of a C component specifies the interfaces that the component implements as well as its other properties.

The component's transaction property determines how it participates in transactions. The threading property imposes constraints on concurrent execution of the component.

Define the component's interfaces

All component interfaces for EAServer components are defined in CORBA IDL modules that are stored in EAServer's IDL Repository. Chapter 5, "Defining Component Interfaces" describes how to define IDL interfaces.

Component developers typically use one of the following to define the interface or interfaces that their component implements:

- Use existing interfaces from EAServer's IDL Repository

  In some cases, client and server component developers may have agreed upon an existing interface or several interfaces that your component must implement. In this case, it is up to you, the component developer, to implement the agreed-upon interface. EAServer stores HTML documentation for all interfaces in the IDL repository in the *html/ir* subdirectory of your EAServer installation.

- Define a new IDL interface or interfaces

If you are defining the interface yourself, you can use EAServer Manager to create a new interface for the component. Chapter 5, "Defining Component Interfaces" describes how.

Note IDL interfaces for C components cannot have create and destroy methods. These conflict with the C create and destroy functions that are called when your component is instantiated and destroyed, respectively.

Transaction property

The component's transaction property determines how it participates in transactions. You can view and change this property using the Transactions tab of the component's property sheet.

When you mark a component as "Requires Transaction," commands that the component sends to third-tier database servers are automatically performed within a transaction. By default, components are not transactional.

Transactional component attribute describes the settings for this attribute. Chapter 2, "Understanding Transactions and Component Lifecycles" introduces the EAServer transaction processing model.

Instance properties

The Instances property imposes constraints on the concurrent execution of the component in different threads. You can view and change these properties using the Instances tab of the component's property sheet.

For a single-threaded component, multiple instances may exist simultaneously, but only one can be active at any one time. EAServer synchronizes instantiations, method invocations, and destructions of all instances. Use the single-threaded model if your component shares volatile global data or stateful resources between instances. For example, volatile global data might be a counter that is stored in a global variable. Sharing a stateful resource would occur if, for example, every component instance opened the same file and wrote text to it. Either example requires the single-threading model. To enable single-threading, do not select any of the options in the Instance Properties tab.

The following settings specify the constraints that are placed on concurrent execution of different instances of the component. The choices are:

- *Concurrency* – multiple invocations can be processed concurrently; that is, multiple instances can be simultaneously active on different threads. The component must be thread-safe. Use this setting if the component code uses no volatile global data and does not share stateful resources (such as a file) among instances. This threading model offers the highest performance.

  The EAServer shared properties feature provides a means to share data safely among instances of a multiple-threaded component. See "Share data between C or C++ components" on page 688 for more information.

- *Bind Thread* – instances are bound to the creating thread. The component uses thread-local storage. The Bind Thread check box determines whether a component instance is always invoked in the same thread or can be invoked on any thread. If Bind Thread is not selected, then EAServer can invoke the component's methods with any thread.

  Do not select this check box, unless your component uses thread-local storage. EAServer provides no APIs for thread-local storage, but you can issue thread system calls from the C component code. Do not use thread-local storage if you are implementing new components. Instead, use the JagGetInstanceData and JagSetInstanceData routines to associate data with a specific component. If you incorporate existing code that uses thread-local storage into a C component, select this check box.

- *Pooling* – instances are pooled after a commit or rollback.

- *Sharing* – a single shared instance services all client requests. Only one instance of the component can exist at any one time. Attempts to create new instances when one already exists will fail. In this model, only one instance of the component may exist at any one time. Attempts to create new instances when one already exists will fail. This option offers the worst performance. Select this check box only if the logic in your code requires that only one component instance exist at one time.

Generate C component files

To write a C component, you need these C component files. You compile these C component files (or C source files) into a DLL.

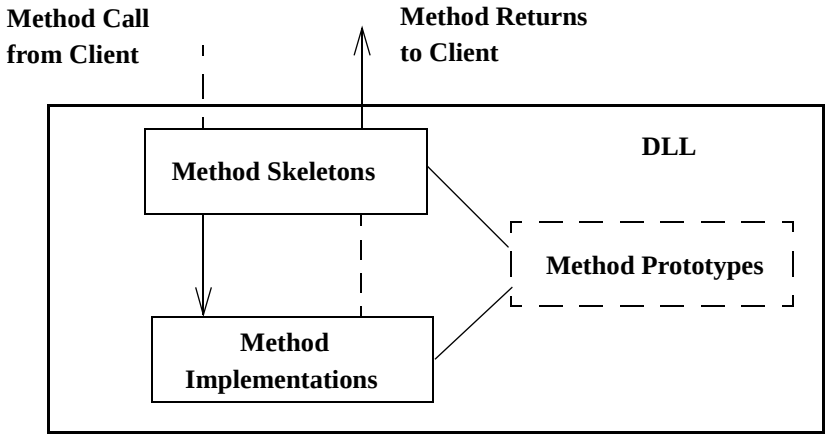
- Method skeletons file – this file contains method routines that read the parameters from the network and call the method. The method skeleton also sends the return status and output parameter data back to the client.

- Method prototypes file – this file contains the method declarations only. This file is an included file in the method skeletons file and the method implementation template files.
- Method implementation template files – these files contain the method and parameter declarations and an empty method definition. You enter any business logic into the empty method definition.

How method calls are made

The graphic below illustrates how EAServer calls the DLL's functions in response to a component methdod invocation:

Figure C-2: How a C component method is called



The sequence of events is:

1 The client invokes a method using the proxy or stub appropriate to the type of client. The stub or proxy sends the invocation information over the network to the server.

2 The method skeleton in the method skeletons file unmarshals the call and makes another call to the method implementation in the method implementation template file.

3 After the method executes, the method implementation returns the call to the method skeleton.

4 The method skeleton marshals the call and sends the call to the client.

Procedure for generating C component files

To generate C component files from a package or component, start EAServer Manager and complete these tasks:

1 Select the component or, if you want to generate files for all components in a package, select the package.

2 Select File | Generate Stub/Skeleton. The Generate Stubs and Skeletons wizard is displayed.

3 Select the Generate Skeletons check box. Unless you wish to generate stubs at the same time, deselect Generate Stubs. Enter values in the Skeletons Generation Options area as follows:

- C/C++ Code Base

  Enter the top-level directory path for the stub files. The path must be a valid UNIX or Windows path. It can include a drive and as many directories as you want.

  If you clear the field, the default is the directory specified by *$HOME* on UNIX and *%HOMEPATH%* for Windows.

  Sybase recommends that you specify the full path to the C code base directory. If you specify a relative path, it is created under the EAServer installation directory, relative to the *html/classes* subdirectory.

- Java Code Base

  If you are generating skeletons for a package that contains both Java and C components, specify the location where generated Java skeletons are to be created. Otherwise, you can leave this field alone.

  Warning! Do not use the component name as the method file name. The component name is already used for the method skeletons file.

4 Click Generate. EAServer Manager generates a method implementation file name, and create and destroy routine templates appended with *.new*.

5 Rename the generated method implementation template files, deleting *.new*.

File naming conventions

The component files are named according to this syntax:

file	file name
component skeleton	*component-name.c*
method prototypes	*component-name.h*
method implementation	*method-name.c.new*
create routine template	*create.c.new*
destroy routine template	*destroy.c.new*

where

component-name is the name of the component that you defined in EAServer Manager.

method-name can be either of the following:

- If you did not specify a name in the Method file name field, files are generated for each method that you defined in EAServer Manager.
- If you did specify a method file name, that name is used, and all methods are defined in this file. When specifying a file name, leave off the *.c* extension.

EAServer Manager creates the directory structure based on the code base that you specify and the component name, as follows:

```
code_base/jcts_skel/component_name
```

where:

code_base is the directory name that you specify for the Code Base field in EAServer Manager. If the specified value was not a full path, the directory will be located under the EAServer installation directory, relative to the *html/classes* subdirectory.

component_name is the component name as displayed in EAServer Manager.

Regenerate changed C component methods

When you add or delete methods or modify component method prototypes, you must regenerate the method skeletons and prototypes. You must manually add, delete, or modify the method in the implementation file. Before you regenerate the method skeletons and prototypes, move your modified implementation files to another directory or rename them so the new implementation template files do not overwrite your modified implementation files.

Write C components

After you generate method skeletons, prototypes, and implementation templates, write the code for each method in the method implementation file. You can include C or C++ functions in C components. EAServer provides C routines for common C component tasks (see Chapter 5, “C Routines Reference,” in the *EAServer API Reference*).

To adapt a C++ class for use as a C component, you must write C wrappers. For details, see “C components that are wrappers for C++ classes” on page 686.

EAServer Manager creates template files for each method when you define the method signatures (or method prototypes) and generate skeleton routines. You can modify the template files to implement the method bodies, or you can code your methods from scratch according to the rules laid out here.

Note Function overloading is not supported for C components.

You can also include EAServer routines to:

- Customize the creation and destruction of components
- Manage instance data
- Share data between components
- Connect to third-tier database servers
- Call other components
- Send result sets
- Set transaction state
- Write to a log file

Define implementation functions

Each method in the EAServer component definition is implemented by a C function with the same name as the method.

Implementation function return codes

Method implementation functions must return CS_RETCODE. Your implementation function can return the following values:

- CS_SUCCEED – to indicate successful execution.
- CS_FAIL – to indicate failure. The client stub raises an exception when the server method returns CS_FAIL. You can call the JagSendMsg routine prior to returning CS_FAIL to specify the exception text to be sent to the client.

In general, you should return CS_SUCCEED unless a fatal error occurs. Returning CS_FAIL prevents the client stub from receiving output parameter values. Use an inout or output parameter if you need to communicate method status information to the client application.

Calling conventions

EAServer calls methods using a specific C calling convention. Follow these rules to ensure compatibility with the EAServer method calling convention:

- Use extern C blocks – if using C++, make sure to declare your methods in an extern C block.
- Always use CS_PUBLIC – on platforms such as Windows, the C compiler supports a broad collection of calling conventions for C functions. On these platforms, the CS_PUBLIC macro encapsulates the appropriate compiler keywords to ensure that the same calling convention is used by Jaguar and your methods. In the function prototype, insert CS_PUBLIC between the return type and the function name.

Parameter datatypes

“Datatypes for C method implementation functions” on page 682 shows the datatypes displayed in EAServer Manager, the datatypes used by C components, and the argument modes. The left column contains the datatype name as it displays in EAServer Manager. The second and third columns contain the names of the corresponding C datatypes for input, inout, and output parameters.

If the EAServer Manager method definition returns a value other than *ResultSet* or *ResultSets*, an additional output parameter is added to the front of the implementation function's parameter list. This additional parameter receives the "logical return code" for the method invocation, as described in "Logical method return values" on page 684.

Datatypes for C method implementation functions

EAServer Manager	Mode	C Datatype
boolean	input inout, output, return	CS_BIT CS_BIT *
byte (a single byte)	input inout, output. return	CS_BINARY CS_BINARY *
char (a single character)	input inout, output. return	CS_CHAR CS_CHAR *
float	input inout, output. return	CS_REAL CS_REAL *
double	input inout, output. return	CS_FLOAT CS_FLOAT *
integer<16>	input inout, output. return	CS_SMALLINT CS_SMALLINT *
integer<32>	input inout, output. return	CS_INT CS_INT *
integer<64>	input inout, output. return	CS_LONG CS_LONG *
binary	input inout, output. return	CS_BINARY_HOLDER * CS_BINARY_HOLDER *
string	input inout, output. return	CS_LONGCHAR_HOLDER * CS_LONGCHAR_HOLDER *
string<255>	input inout, output. return	CS_CHAR * CS_CHAR *
timestamp	input inout, output. return	CS_DATETIME CS_DATETIME *

Argument modes

Argument modes specify how an argument is passed. Arguments can have one of these modes:

- Input - read-only; arguments are passed by value.
- Inout - read/write; arguments are passed by reference.
- Output – same as inout, except that input values are ignored.

- Return – the method returns a value of this datatype. See "Logical method return values" on page 684.

All parameters specified as input are passed by value except for those parameters declared as string, string<255>, or binary in EAServer Manager. Except for binary and string parameters, EAServer always preallocates sufficient space for inout, output, or return parameters. binary and string parameters are mapped to special datatypes, and you may need to reallocate space for the output value as described below.

string<255> parameters

string<255> parameters are passed as a CS_CHAR *. On input, EAServer null-terminates CS_CHAR parameter values using the length meta-information associated with the datatype. On output, updated CS_CHAR parameter values must be null-terminated.

Note string<255> parameter values cannot be longer than 255 bytes. Use string parameters if your application requires larger values.

---

string and binary parameters

string parameters are passed in a CS_STRING_HOLDER structure. binary parameters are passed in a CS_BINARY_HOLDER structure. These structures are defined in *jagpublic.h* as follows:

```
typedef struct _cs_longchar_holder
{
        CS_LONGCHAR *value;
        CS_INT length;
} CS_STRING_HOLDER;

typedef struct _cs_longbinary_holder
{
        CS_LONGBINARY *value;
        CS_INT length;
} CS_BINARY_HOLDER;

#define CS_LONGCHAR_HOLDER    CS_STRING_HOLDER
#define CS_LONGBINARY_HOLDER  CS_BINARY_HOLDER
```

To allow backward compatibility with code that was written for EAServer version 1.1, you can use CS_LONGCHAR_HOLDER in place of CS_STRING_HOLDER, and CS_LONGBINARY_HOLDER in place of CS_BINARY_HOLDER.

On input, the *value* field contains the input value and the *length* field specifies the input length. For output, you can set a new value and length in the structure as follows:

- If the output value is longer than the input length, you must reallocate memory for the value and reset the *value* and *length* fields. Use the JagFree and JagAlloc routines as shown in the example below.
- If the output value is not longer than the input length, you can copy the output value directly into the buffer addressed by the input *value* pointer and reset the *length* field.

The following example calls the JagFree and JagAlloc routines to reallocate a larger *value* buffer:

```
JagFree(myholder->value);
myholder->value = JagAlloc(new_length);
if (myholder->value == NULL)
{
    JagLog(JAG_TRUE, "Out of memory!\n");
    return CS_FAIL;
}
memcpy(myholder->value, new_value, new_length);
myholder->length = new_length;
```

NULLs

NULLs cannot be passed to or returned by method calls. Instead of using NULL for string parameters, pass zero-length values.

Logical method return values

If the EAServer Manager method definition returns a value other than ResultSet or ResultSets, the C function signature contains an additional parameter in the first position. This parameter functions as a logical return value for method invocations. When the C function returns, the output value of this parameter is forwarded to the client, and the client receives it as the return value for the stub method invocation. Datatype mappings for this added parameter are the same as for an output parameter.

If the EAServer Manager method definition returns ResultSet or ResultSets, you must use the C Result Set API calls to build the result set or sets to be sent to the client, as described in “Methods that return row results” on page 688.

Implementing the method behavior

In most cases, the implementation of C component methods require no special coding. Simply add code to the method body that contains the application logic to respond to the input parameter values and assign the correct return values to inout, output, and return parameters.

The exceptions to this rule are:

- Components that require instance specific data

  C component instances do not have private data. If your design requires the allocation of separate data for instances of the same component, see “Components that require instance specific data” on page 686.

- C components that are wrappers for C++ classes

  Since C component methods must be C functions, you must code a set of C wrapper functions that instantiate and interact with an instance of the C++ class. See “C components that are wrappers for C++ classes” on page 686 for more information.

- Methods that interact with remote database servers

  You can use a connection cache to improve performance when connecting to database servers. See “Methods that interact with remote database servers” on page 688 for more information.

- Methods that return row results

  an EAServer method can return row results to the client. Doing so requires the use of Server-Library calls. See “Methods that return row results” on page 688 for more information.

- Share data between C or C++ components

  Components within the same package can share the same data. See “Share data between C or C++ components” on page 688 for more information.

- Methods that set transactional state

  Methods in a transactional component should call one of the transaction primitive routines to set the transaction state before returning. See “Methods that set transactional state” on page 695 for more information.

Components that require instance specific data

C components do not contain private data. To allocate separate data for instances of the same component, use the JagSetInstanceData and JagGetInstanceData routines.

EAServer provides two functions for managing instance-specific data:

- JagSetInstanceData – Associates a reference to instance data with the current C component instance.
- JagGetInstanceData – Retrieves the address of component instance data.

Chapter 5, "C Routines Reference," in the *EAServer API Reference* contains reference pages for these routines.

C components that are wrappers for C++ classes

Since methods in a C component must be implemented as C functions, you must code C wrappers for C++ classes.

Note Beginning in version 2.0, EAServer provides direct support for running C++ classes as components, as described in Chapter 14, "Creating CORBA C++ Components" Sybase supports the technique described here, but recommends that you create a C++ component to run your C++ classes directly.

The procedure for creating a wrapper for a C++ class is as follows:

1 Code a C create function that instantiates the C++ object and stores the object reference as instance-specific data. For example, if the C++ object is *StockTrade*, create could be implemented as follows:

```
CS_RETCODE CS_PUBLIC create() {
  StockTrade *st_ref;
  /*
  ** Create an instance of the C++
  ** StockTrade object.
  */
  st_ref = new StockTrade();
  /*
  ** Associate it with the EAServer component
  ** instance.
  */
  if (JagSetInstanceData((CS_VOID *)st_ref)
```

```
        != CS_SUCCEED)
    {
        return CS_FAIL;
    }
    return CS_SUCCEED;
}
```

2 For each C++ method, code a C wrapper function that retrieves the C++ object reference and uses it to call the C++ method. For example, the following shows a C wrapper to call a StockTrade::buyStock C++ method:

```
CS_RETCODE CS_PUBLIC buyStock (
      CS_CHAR  *ticker,
      CS_INT   n_desired,
      CS_INT   n_bought)
{
  StockTrade *st_ref;
  if (JagGetInstanceData((CS_VOID *)&st_ref)
      != CS_SUCCEED)
  {
    return CS_FAIL;
  }
  st_ref::buyStock(ticker, n_desired,
                           n_bought);
  return;
}
```

3 Code a destroy function that retrieves the C++ object reference and destroys it. For example:

```
CS_RETCODE CS_PUBLIC destroy() {
  StockTrade *st_ref;
  if (JagGetInstanceData((CS_VOID *)&st_ref)
      != CS_SUCCEED)
  {
    return CS_FAIL;
  }
  delete st_ref;
  return CS_SUCCEED;
}
```

4 Define the EAServer C component to include the C wrapper functions as its methods.

Methods that interact with remote database servers

EAServer uses a connection cache to maintain a pool of connections to database servers. A component can connect to a database server using an existing connection in a connection cache without creating a new connection.

Note EAServer's transactional model works only with connections obtained from the EAServer Connection Manager. Connections that you open yourself will not be affected by EAServer transactions.

For more information about coding connection management routines into components, see Chapter 26, "Using Connection Management".

Methods that return row results

To return row results, call the result-set routines listed in Chapter 5, "C Routines Reference," in the *EAServer API Reference*. "Sending result sets from a C or C++ component" on page 475 explains the call sequence and contains examples.

Share data between C or C++ components

Components in the same package can share data—that is, variable values. For example, a counter that tracks how many objects have been created for a single component could be used as a shared variable. Shared variables are organized into collections. These variables are referred to as shared because components in the same package can read and update the same data. A collection can contain any number of shared variables. Shared variables can be identified by name or by index number. Shared variables are initialized as null and are not saved when the server is shut down.

Because it is important to maintain the integrity of the shared data in shared variables, a single read or update operation on a shared variable is atomic. *Atomic* means that an operation on data will complete before any other operations can access that data. Multiple reads and updates on any number of shared variables in a single collection can be synchronized by locking that collection.

Note You cannot use shared variables in components that are configured for automatic failover, because these components cannot use local shared resources. See "Component properties: Transactions" on page 58 for more information. If you need to share data, you can store shared data in a remote database.

To share data between components, you must include the *jagpublic.h* file in the C source file.

Procedure for sharing data

The general procedure for sharing data is:

1 Create or retrieve references to collections – the component must first create a collection before creating a shared variable in that collection. Creating a collection automatically retrieves a reference to the new collection. If the collection already exists, the component must retrieve a reference to the collection before creating a shared variable.

2 Lock collections – before creating a shared variable or reading and updating shared variables, lock the collection. Locking a collection ensures that the integrity of a shared variable will be maintained when the shared variable is read and updated by the same method multiple times. The component does not have to lock a collection if you are executing only one read or update on a shared variable.

3 Create or retrieve references to shared variables – the component must first create a shared variable before reading or updating the shared variable. Creating a shared variable automatically retrieves a reference to the new shared variable. If the shared variable already exists, the component must retrieve a reference to the shared variable before reading or updating a shared variable.

4 Read and update shared variables – after creating or retrieving a reference to a shared variable, the component can read and update the shared variable.

5 Unlock collections – after reading and updating all shared variables in a collection, unlock the collection. Unlocking a collection immediately after the component instance has completed all operations on a shared variable allows other component instances to access the shared variable right away.

6 Release shared variable and collection references – after reading and updating shared variables, release the reference to the shared variable. After all operations on shared variables in a collection have been completed, release the reference to the collection. Component objects use memory efficiently by releasing references immediately after the component object has completed operations on a shared variable or collection.

You can also list all collection names on the server by calling the JagGetCollectionList routine. For more information see “List all collections” on page 694.

Create shared variables and collections

The component must create the collection before it can create shared variables.

1 Create a collection and return a reference to the collection using the JagNewCollection routine. The component object must have a reference to a collection before calling any other routines on the collection.

2 Create a shared variable in a collection and return a reference to the shared variable using the JagNewSharedData or JagNewSharedDataByIndex routine. The component instance must have a reference to a shared variable to access the contents of that shared variable.

Create collections

To create a new collection, call the JagNewCollection routine. This routine:

- Creates a new collection with the specified lock level, returns a reference to that collection, and sets *pExists to JAG_FALSE, or
- Returns a reference to the existing collection with the specified name and sets *pExists to JAG_TRUE. The input lock level is ignored, and the collection’s current lock level is returned in *pLockLevel.

Lock level must be set to one of the following:

JAG_LOCKCOLLECTION – allows locks to be set on collections

JAG_LOCKDATA – does not allow locks to be set on collections

Create shared variables in collections

To create a new shared variable, call the JagNewSharedData or JagNewSharedDataByIndex routine. These routines create a shared variable value initialized to NULL. JagNewSharedData creates a new shared variable or returns a reference to an existing shared variable by name. JagNewSharedDataByIndex creates a new shared variable or returns a reference to an existing shared variable by index number. A shared variable created by index can only be retrieved or updated by index. Similarly, a shared variable created by name can only be retrieved or updated by name. Since a reference is returned, you do not need to follow these routines with the JagGetSharedData or JagGetSharedDataByIndex routine.

For both routines, *pExists is set to:

- JAG_TRUE if the shared variable does not exist
- JAG_FALSE if the shared variable exists

Lock and unlock collections

Locking a collection is strictly advisory. Use JagLockCollection and JagLockNoWaitCollection routines to lock collections. Even though a collection is locked, the JagGetSharedValue and JagSetSharedValue routines can still read and update the shared variables in the collection. To ensure that multiple read or update operations on any shared variable in a collection are atomic, lock the collection before executing read or update operations on the shared variables in the collection.

Call the JagGetLockLevel routine to determine a collection's isolation mode. If the collection's isolation mode is JAG_LOCKCOLLECTION, then the component object can lock the collection. Otherwise, the lock will be rejected.

If you call the JagLockCollection routine to lock a collection that is locked by another component, JagLockCollection waits until the collection is unlocked by the other component, then locks the collection. If the lock is successful, JAG_SUCCEED is returned. If the collection has already been locked by the calling object, this routine does not lock the collection again and JAG_SUCCEED is returned.

The JagLockNoWaitCollection routine does not wait until a locked collection is unlocked; the JagLockNoWaitCollection routine immediately returns execution to the calling routine. This routine returns JAG_SUCCEED and sets *pLocked to JAG_TRUE if the collection was not locked or if the collection is already locked by the same calling object. If the collection was locked by another component object, JAG_SUCCEED is still returned but *pLocked is set to JAG_FALSE.

The JagLockCollection and JagLockNoWaitCollection routines return JAG_FAIL if an error, such as the collection's isolation mode is JAG_LOCKDATA, occurs.

Call the JagUnlockCollection routine to release a lock on a collection. A locked collection is automatically released when the component object's method execution is completed. However, to make your application more efficient and prevent deadlocks, unlock a collection when the component object is finished updating or reading the shared variable in the collection so that other component objects can access the collection right away.

Read and update shared variables

Before reading or updating the shared variable, the component object must retrieve references to the collection and shared variable.

1 Call the JagGetCollection routine to retrieve a reference for a collection. If the component object has just created the collection, the component object doesn't need to call this routine.

2 Call the JagGetSharedData or JagGetSharedDataByIndex routine to retrieve a reference to a shared variable. If the component object has just created the shared variable, the component object doesn't need to call either of these routines.

3 Call the JagGetSharedValue routine to retrieve a shared variable value.

4 Call the JagSetSharedValue routine to assign a new value to the shared variable.

Retrieve references for collections

The JagGetCollection routine returns a reference to the specified collection. Once the component instance has retrieved a reference, the component object can lock and unlock the collection, create a new shared variable in the collection, or retrieve a reference to an existing shared variable.

If the collection exists, JAG_SUCCEED is returned and **ppCollection is set to the collection reference. If the collection does not exist, JAG_FAIL is returned and **ppCollection is set to NULL.

Retrieve references to shared variables

The JagGetSharedData and JagGetSharedDataByIndex routines return a reference to the specified shared variable. The component object must have already retrieved the collection reference before calling these routines. When calling the JagGetSharedData routine, you specify the shared variable by name. When calling the JagGetSharedDataByIndex routine, you specify the shared variable by index number.

For both routines, if the shared variable exists, JAG_SUCCEED is returned and **ppData is set to the shared variable reference. If the shared variable does not exist, JAG_FAIL is returned and **ppData is set to NULL for both routines.

Retrieve shared variable values

The JagGetSharedValue routine retrieves the value for a specified shared variable and places the value in a buffer. The component object must have retrieved the shared variable reference before executing this routine. The component object must create a buffer in which to copy a value. The buffer must be large enough to hold any value that can be stored in the shared variable. You must specify the buffer (and its size) in which the value is to be copied. The buffer must be large enough to contain the value. If the value is too large for the buffer, JAG_FAIL and the size of the value are returned.

If the value is successfully copied into the buffer, JAG_SUCCEED and the number of bytes copied to the buffer are returned. If **outlen* is 0, then there was no value to copy.

Update shared variables with new values

The JagSetSharedValue routine copies a value to a specified shared variable. The component object must have retrieved the shared variable reference before calling this routine. The component object must pass a pointer to the value you want the component object to copy to the shared variable. This routine copies the value to the shared variable. You must specify the size of the value. If the value is a null-terminated string, you must include the length of the null terminator in the length of the string.

EAServer maintains the values of shared data in its own memory space. When JagSetSharedValue() copies the data, it does not copy the pointer to the data. Similarly, JagGetSharedValue() copies the data into a buffer supplied by the caller, it does not place a pointer to the data in the user's buffer.

If the new value is copied to the shared value, JAG_SUCCEED is returned. If an error occurs, JAG_FAIL is returned.

Release shared variable and collection references

After a method finishes all operations on a collection, release the reference and all shared variable references. This helps to prevent memory leaks. Releasing collection and shared variable references does not release the shared variable values.

First, release shared variable references and then release the collection reference. To release shared variable references, call the JagFreeSharedDataHandle routine, passing the shared variable reference as input. To release collection references, call the JagFreeCollectionHandle routine on the collection reference.

If the shared variable or collection reference is released, JAG_SUCCEED is returned. If an error occurs, JAG_FAIL is returned.

List all collections

Call the JagGetCollectionList routine to retrieve a list of all the collection names on the server. The server returns a JagNameList structure. This routine can be called in conjunction with administering EAServer. Call the JagFreeCollectionList routine to free the memory allocated for the JagNameList structure.

The JagGetCollectionList routine returns a reference to a JagNameList structure that includes all the collection names defined in EAServer. The JagNameList structure is:

```
typedef struct _jagnamelist
{
    SQLINT            num_names;
    SQLPOINTER            *names;
} JagNameList;
```

where:

num_names is the number of array elements.

**names* is an array of *num_names* elements; each element points to a null-terminated collection name.

Methods that set transactional state

Methods in a transactional component should call one of the transaction state primitive routines listed in Chapter 5, "C Routines Reference," of the *EAServer API Reference*.

Even if your component is not transactional, you should call one of these methods to explicitly specify whether the instance should be deactivated.

For transactional components, choose the routine that reflects the state of the work that the component is contributing to the transaction, as follows:

- If the work is complete and without error, call JagCompleteWork.
- If the work is not necessarily finished, but not in error, call JagContinueWork.
- If the work is not finished and not ready for commit, call JagDisallowCommit.
- If the work cannot be completed, call JagRollbackWork (you should also log a description of the error and send an error to the client, as described in "Handle errors in your C component" on page 696).

For nontransactional components, call either JagCompleteWork or JagRollbackWork to deactivate and destroy the component instance. To keep the instance active, call JagContinueWork or JagDisallowCommit.

If a method does not explicitly set transaction state before returning, the default behavior is JagContinueWork.

Customize the creation and destruction of components

To customize what happens when a component instance is created or destroyed, write customized code into the create (*create.c.new*) and destroy (*destroy.c.new*) routine templates that are generated by EAServer Manager. create and destroy are typically used to manage instance-specific data that the component requires. For example, some methods might need to be executed in a certain sequence. You can customize the create and destroy routines to keep track of which methods have been executed. For details on managing instance-specific data, see “Components that require instance specific data” on page 686.

The create and destroy routines are optional. You can also implement create and destroy in another source file and ignore the generated templates. The create and destroy routines cannot have parameters and cannot return result sets.

create routine

The Jaguar server calls create when creating a new instance of the component. The signature for create is:

```
CS_RETCODE CS_PUBLIC create()
```

create must return CS_SUCCEED.

destroy routine

EAServer calls destroy when destroying an instance of the component. The signature for destroy is:

```
CS_RETCODE CS_PUBLIC destroy()
```

destroy must return CS_SUCCEED.

Handle errors in your C component

As a general rule, code C component methods to handle unrecoverable errors as follows:

1 Write detailed error descriptions to the server log file using JagLog.

2 Call JagSendMsg to send a descriptive message to the client.

3 If the component is transactional, call JagDisallowCommit or JagRollbackWork as appropriate.

4 Return CS_FAIL to indicate failed execution.

Chapter 5, "C Routines Reference," of the *EAServer API Reference* contains reference pages for these routines.

Compile C components

This section describes how to compile and link a dynamic link library (DLL) that contains EAServer methods. Your code must be built as a DLL in order to be installed into the EAServer runtime environment.

When you generate source files for your component, EAServer Manager creates an example makefile that builds the component library. You may have to edit this file to match your environment, as described in the following sections:

- "Build on UNIX" on page 697
- "Build on Windows" on page 698

Build on UNIX

For servers that run on UNIX, you must build shared library files that contain your C component methods. After building the shared library, copy it to the *cpplib* directory of your EAServer installation.

EAServer Manager generates a *make.unix* file when you generate the component skeleton. (See "Generate C component files" on page 676 for instructions on generating code with EAServer Manager.)

To build your component, run the following command:

```
make -f make.unix
```

This command builds a shared library named *libComp.so*, where *Comp* is the EAServer Manager name of the component.

Note If you edit the generated *make.unix* file, rename the edited version so that it is not overwritten if you regenerate the skeleton files.

Build on Windows

For servers that run on Windows, you must build dynamic link library (*.dll*) files that contain your C component methods.

Follow the instructions in the sections below to build a DLL. After building the shared library, copy it to the *cpplib* directory of your EAServer installation.

DLLs for C components

For Windows, you must build a DLL so that C functions use the standard C calling function.

EAServer Manager generates a *make.nt* file when you generate the component skeleton. (See “Generate C component files” on page 676 for instructions on generating code with EAServer Manager.)

The generated makefile assumes that the ODBC header files and libraries can be found in one of the following locations:

- The directory specified by the ODBCHOME environment variable, or
- If ODBCHOME is not set, *C:\msdev*, which is the default installation directory for Microsoft Visual C++.

The makefile links the following import libraries:

- *libjaguar.lib* – Contains import definitions for routines called by generated skeleton routines.
- *libjcm.lib* – Contains import definitions for Connection Manager routines.
- *libjdispatch.lib* – Contains import definitions for EAServer C routines.

If your code calls routines defined in other import libraries, you will need to rename and edit the generated makefile to use the additional import libraries. If you edit the generated *make.nt* file, rename the edited version so that it is not overwritten if you regenerate the skeleton files.

To use the generated makefile, run this command from a command window while in your component’s source directory:

```
nmake -f make.nt
```

After building the DLL, copy it to the *cpplib* directory of your EAServer installation.

Build with the Visual C++ IDE

If you use the Visual C++ IDE or another command line compiler to build your DLL, make sure that you specify the right options so that the compiler generates C functions using the standard C calling convention.

After building the DLL, copy it to the *cpplib* directory of your EAServer installation.

The module definition (.def) file

EAServer Manager generates a *.def* file for your component. Visual C++ requires a module definition file that specifies which functions are exported from a DLL and some options that control how the DLL is loaded into memory. Module definition files end with the extension *.def*.

For most projects, you can use the generated file as is. In some cases, you may want to edit settings other than those in the EXPORTS section. For example, your component may perform better with a smaller or larger HEAPSIZE setting.

Never edit the generated function names in the EXPORTS section of the *.def* file for a C component–these names are required to execute the DLL in EAServer. For each method in your component, the changed name is:

```
__skl_Comp_v_1_0_method
```

Debug C components

You can attach your debugger to the server executable and set breakpoints to step into your component code.

In order to debug components, you must be running the debug version of the server, and use a debugger running on the same host as the server. Chapter 3, "Creating and Configuring Servers," in the *EAServer System Administration Guide* describes how to start the debug server.

❖ **Debugging your C component**

1 Attach to the EAServer process with your debugger.

Alternatively, start the debugger with the EAServer executable. For example, on UNIX, enter:

```
dbx $JAGUAR/devbin/jagsrv ServerName
```

On Windows, enter:

```
msdev %JAGUAR%\devbin\jagsrv ServerName
```

ServerName is the name of the server. If you are using the preconfigured server rather than one that you created yourself, use "Jaguar."

2 Set a breakpoint on the function jag_dbg_stop. This function executes every time the server loads a component DLL. The jag_dbg_stop prototype is:

```
void jag_dbg_stop(char *compName)
```

The *compName* parameter specifies the name of the library or shared library that was just loaded. Several components may be loaded before yours. In the debugger, display the *compName* value when the jag_dbg_stop breakpoint is tripped, and monitor the value to determine when your component is loaded. Breakpoints on jag_dbg_stop are triggered before EAServer calls the component's create method.

Note Make sure the jag_dbg_stop breakpoint is set before your client application instantiates any stub objects.

3 When your component's DLL is loaded, you can specify the component's C function names as breakpoints and step into the method's code when it is invoked. Note that the actual C function names exported by the DLL will not match the method names that you see in your source code. The next section describes how to determine the C function names for your methods.

Determining actual function names for your methods

In many operating systems, all functions in a single executable must have unique names. For this reason, the generated skeleton code contains macros that rename each method with a longer name at compile time. The final name is guaranteed to be unique among installed components. You must use these longer names to set breakpoints when debugging.

To view the name mappings, look at the generated skeleton header file for your component. There will be a macro that renames each method. The final method is renamed according to this syntax:

```
package_component_method
```

where *package* is the package name, *component* is the component name and *method* is the method name. For example, the component named "sendrows" in a package named "jagdb," is renamed as follows:

```
#define    send_rows     jagdb_sendrows_send_rows
```

APPENDIX D

Using the Command Line IDL Compiler

You can use EAServer's command-line IDL compiler to generate stubs, skeletons, and ActiveX type library and registration files. You can also use EAServer Manager to perform these tasks. The command-line compiler is useful in environments where the generation of files must be automated.

Do you know about jagtool and jagant?
The jagtool utility, provided in EAServer 4.0 and later versions, also supports stub and skeleton generation from the command line, as well as common management tasks such as setting component properties. The jagant utility provides the same functionality, but can be run from Jakarta Ant build files. For more information, see Chapter 12, "Using jagtool and jagant," the *EAServer System Administration Guide*.

Requirements

The IDL compiler must be run from EAServer installation that contains the configuration repository for your components. The IDL compiler is a Java application and must be run by a version 1.2 or later Java virtual machine. These EAServer JAR files must be in the CLASSPATH:

- *java/lib/easclient.jar*
- *java/lib/easserver.jar*
- *java/lib/easj2ee.jar*

com.sybase.CORBA.idl.Compiler

Description

Generates EAServer proxies or component skeletons for interfaces and datatypes defined in CORBA IDL.

Syntax

```
com.sybase.CORBA.idl.Compiler prefix-opts idl-files format-opts suffix-opts
```

where:

- **prefix-opts** Is a list of one or more of these options:

Option	Explanation
`-v`	Run in verbose mode.
`-i folder`	Add folder for IDL include files. The default is the EAServer *Repository/idl* subdirectory.
`-r folder`	*Required.* Specify the location of the EAServer Repository subdirectory, for example, *d:\EAServer\Repository*.
`-lang language`	Specify a language for error messages. Possible values are: • `us_english` – The default. • `english` • `japanese` • `german` • `french` • `korean` • `chinese`

- **idl-files** Is a list of one or more IDL module files, separated by spaces. Nested IDL modules are organized in nested directories; you must specify the path to a nested module relative to the EAServer *Repository/idl* subdirectory. For example, the file name that defines module com::foo::bar is *com/foo/bar.idl*.
- **format-opts** is a list of one or more of the following options:

Option	Explanation
`-cr`	Use CR/LF in generated files.
`-f folder`	Specify base folder for code generation.
`-p folder`	Specify location of *jagproxy.dll* for generation of ActiveX type library and Registry files.
`-jp IM=JP`	Set Java package *JP* for IDL module *IM*, for example: `-jp com::foo::bar=com.foo.bar` The specified package overrides the default (see "Specifying Java package mappings for IDL modules" on page 87).

Option	Explanation
`-jv version`	Set Java version for code generation. The Java version affects only EJB stubs, specifically it configures the default for finder methods that return multiple keys and lack an IDL directive to specify the Java return type. See "Generating EJB stubs" on page 138 for more information.
`-nh`	Suppress generation of Java helper and holder classes.
`-np`	Suppress module prefix for PowerBuilder stubs.

- **target-opts** is a list of one or more of the following options:

Option	Explanation
`-cpp`	Generate C++ stubs following CORBA 2.3 C++.language bindings specification.
`-java`	Generate Java stubs following CORBA 2.3 Java.language bindings specification.
`-ejb`	Generate EJB stubs.
`-pb`	Generate code for CORBA 2.3 / PowerBuilder.
`-idl`	Generate pretty-printed IDL output.
`-reg load`	Generate REG file and optionally load into registry. `-reg load` generates loads the REG file. `-reg load` generates and does not load.
`-tlb save`	Generate TLB file and optionally save generated MIDL. `-tlb nosave` generates TLB and deletes the intermediate MIDL file. `-tlb save` generates TLB and keeps the MIDL file.
`-skeleton Pack/Comp`	Generate a skeleton for EAServer component *Pack*/*Comp*, where *Pack* is the EAServer package, and *Comp* is the component name as displayed in EAServer Manager.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X