Programmer's Guide (Java™)

iPlanet Application Server

Release 6.0

806-4791-01
April 2000

Contents

Preface

This preface describes the iAS documentation set and illustrates what you can expect to find in this *Programmer's Guide*.

This preface contains the following sections:

- Using the Documentation
- About This Guide
- What You Should Already Know
- How This Guide Is Organized
- Documentation Conventions
- Related Information

Using the Documentation

The following table lists the tasks and concepts that are described in the iPlanet Application Server (iAS) and iPlanet Application Builder (iAB) printed manuals and online read-me file. If you are trying to accomplish a specific task or learn more about a specific concept, refer to the appropriate manual.

Note that the printed manuals are also available as online files in PDF and HTML format.

Table 1 iAS documentation Roadmap

For information about	See the following	Shipped with
Late-breaking information about the software and the documentation	readme.htm	iAS 6.0, iAB 6.0

Table 1 iAS documentation Roadmap

For information about	See the following	Shipped with
Installing iPlanet Application Server and its various components (Web Connector plug-in, iPlanet Application Server Administrator), and configuring the sample applications	Installation Guide	iAS 6.0
Installing iPlanet Application Builder.	install.htm	iAB 6.0
Basic features of iAS, such as its software components, general capabilities, and system architecture.	Overview	iAS 6.0 Developer Edition (Solaris), iAS 6.0, iAB 6.0
Administering one or more application servers using the iPlanet Application Server Administrator tool to perform the following tasks: • Deploying applications with the Deployment Manager tool • Monitoring and logging server activity • Setting up users and groups • Administering database connectivity • Administering transactions • Load balancing servers • Managing distributed data synchronization	Administration and Deployment Guide	iAS 6.0
Migrating your applications to the new iPlanet Application Server 6.0 programming model from version 2.1 & 4.0, including a sample migration of an Online Bank application provided with iPlanet Application Server	Migration Guide	iiAS 6.0, iAB 6.0

Table 1 iAS documentation Roadmap

For information about	See the following	Shipped with
Creating iAS 6.0 applications within an integrated development environment by performing the following tasks: • Creating and managing projects • Using wizards • Creating data-access logic • Creating presentation logic and layout • Creating business logic • Compiling, testing, and debugging applications • Deploying and downloading applications • Working with source control • Using third-party tools	User's Guide	iAB 6.0
Creating iAS 6.0 applications that follow the new open Java standards model (Servlets, EJBs, JSPs, and JDBC), by performing the following tasks: • Creating the presentation and execution layers of an application • Placing discrete pieces of business logic and entities into Enterprise Java Bean (EJB) components • Using JDBC to communicate with databases • Using iterative testing, debugging, and application fine-tuning procedures to generate applications that execute correctly and quickly	Programmer's Guide (Java)	iAS 6.0, iAB 6.0

Table 1 iAS documentation Roadmap

For information about	See the following	Shipped with
Using the public classes and interfaces, and their methods in the Netscape Application Server class library to write Java applications	Server Foundation Class Reference (Java)	iAS 6.0, iAB 6.0
Creating iAS C++ applications using the iAS class library by performing the following tasks: • Designing applications • Writing AppLogics • Creating HTML templates • Creating queries • Running and debugging applications	Programmer's Guide (C++)	Order separately
Using the public classes and interfaces, and their methods in the Netscape Application Server class library to write C++ applications	Server Foundation Class Reference (C++)	Order separately

About This Guide

This guide describes how to create J2EE applications intended to run on the iPlanet Application Server.

This guide is intended for information technology developers in the corporate enterprise who want to extend client-server applications to a broader audience through the World Wide Web. In addition to describing programming concepts and tasks, this guide offers sample code, implementation tips, and reference material that includes a glossary.

What You Should Already Know

This guide assumes you are familiar with the following topics:

- the Java 2 Platform, Enterprise Edition (J2EE) specification
- the Internet and World Wide Web

- Hypertext Markup Language (HTML)
- Java programming
- JavaSoft APIs as defined in specifications for Enterprise JavaBeans, JavaServer Pages, and JDBC
- structured database query languages such as SQL
- relational database concepts
- software development processes, including debugging and source code control

How This Guide Is Organized

This guide is organized into fourteen chapters and two appendixes, loosely arranged into several parts.

The first part provides an overview for designing programs for the iPlanet Application Server environment. This part includes:

- Chapter 1, “Designing iAS Applications“

The next part describes the programming tasks associated with presentation logic and page design. This part includes the following topics:

- Chapter 2, “Controlling Applications with Servlets“
- Chapter 3, “Presenting Application Pages with JavaServer Pages“

The next part describes the programming tasks associated with business logic and data access. This part includes the following topics:

- Chapter 4, “Introducing Enterprise JavaBeans“
- Chapter 5, “Using Session EJBs to Manage Business Rules“
- Chapter 6, “Building Business Entity EJBs“
- Chapter 7, “Handling Transactions with EJBs“
- Chapter 8, “Using JDBC for Database Access“
- Chapter 9, “Rich Client“

The next part describes issues that affect all parts of an application. This part includes the following topics:

- Chapter 10, “Packaging for Deployment“

- Chapter 11, “Creating and Managing User Sessions“
- Chapter 12, “Writing Secure Applications“
- Chapter 13, “Taking Advantage of iAS Features“

The appendixes include the following reference material:

- Appendix A, “Using the Java Message Service“
- Appendix B, “Dynamic Reloading“
- Appendix C, “Sample Deployment Files“

Finally, a *Glossary* and Index are provided.

Documentation Conventions

File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except that slashes are used instead of backslashes to separate directories.

This guide uses URLs of the form:

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

In these URLs, *server* is the name of server on which you run your application; *domain* is your Internet domain name; *path* is the directory structure on the server; and *file* is an individual filename. Italic items in URLs are placeholders.

This guide uses the following font conventions:

- The `monospace` font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.
- *Italic* type is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.

Related Information

You can find a directory of URLs for the official specifications at *`installdir`*`/ias/docs/index.htm`.

Additionally, we recommend the following resources:

Programming with Servlets and JSPs

Java Servlet Programming, by Jason Hunter, O'Reilly Publishing

Java Threads, 2nd Edition, by Scott Oaks & Henry Wong, O'Reilly Publishing

The web site `http://www.servletcentral.com`

Programming with EJBs

Enterprise JavaBeans, by Richard Monson-Haefel, O'Reilly Publishing

The web site `http://ejbhome.iona.com`

Programming with JDBC

Database Programming with JDBC and Java, by George Reese, O'Reilly Publishing

JDBC Database Access With Java: A Tutorial and Annotated Reference (Java Series), by Graham Hamilton, Rick Cattell, Maydene Fisher

Chapter 1

Designing iAS Applications

This chapter summarizes the process of designing iAS applications and offers guidelines for effective design.

Designing applications for iAS involves four main steps, described in the following sections:

- Identifying Application Requirements
- Assembling the Development Team
- Designing the User Interface
- Following Guidelines for Effective Development

Identifying Application Requirements

In any development cycle, the first step—and arguably the most important step—is to gather the requirements of the application. This step may take weeks or months, and it involves your product management or product marketing team.

As a brief example, suppose you are developing an online banking application. After discussions with your customers, you might come up with the following requirements. This list is a just a sample, and does not include actual numbers.

- security
- specific features: account transfers, account reporting, online trades, special offers to qualified customers
- different types of end users; for example, individuals, corporations, or internal users (bank employees)
- internal reporting

Defining the Requirements

Along with identifying requirements, you may need to further define them. Understanding the requirements in detail helps establish the business rules and design guidelines. For example, you might look more closely at who the end users are.

Will users be anonymous, or will they be required to login? In many cases, an application provides introductory web pages that are freely browsable by anonymous users, while other pages would require logging in as a registered users. Registered users might be further classified as regular or premier customers, depending on specified criteria. What are these criteria? For example, a customer may earn premier status by purchasing at least $500 worth of merchandise per month or by maintaining a minimum of $5000 in an account.

Matching Requirements to the Application Model

Designing toward your requirements may affect one or more layers in the application model. Consider the end-user requirements, for example.

In the presentation layer, the application may need to present one set of pages for anonymous users, and another set for registered users. Also, when an anonymous user tries to access a feature reserved for registered users, you might design the application to present a page that explains why the attempt failed, and invite the user to become a member. By the same token, premier customers might have access to some pages that are denied to regular customers.

In the business logic layer, the application must authenticate login attempts against known users, as well as test that users meet the criteria for accessing particular application features.

In the data access layer, the application may need to restrict database access based on the category of end user.

Assembling the Development Team

The application model allows different individuals to focus on developing application elements at the same time. This section describes the skills needed from designers and developers for each application layer. The following categories are described:

- The Architect

- Team Roles for the Presentation Layer
- Team Roles for the Business Logic Layer
- Team Roles for the Data and Legacy Access Layer

The composition of the team depends on the size of your group and the scope of the application. For example, not all team roles may be needed, and team members could assume more than one area of responsibility.

The Architect

In component-based application development, one or more individuals must have an overall vision of the application, its control flow, and other interactions. Some organizations call this person the architect. Regardless of the name you use, this individual serves an important role by coordinating the efforts of the various design teams and by helping them think about “the big picture.”

Team Roles for the Presentation Layer

The presentation layer is where the user interface is dynamically generated. The following team members are needed:

- Java servlet developers
- JSP developers
- HTML designers
- graphic artists
- client-side JavaScript developers

Servlet developers create the presentation logic, whereas the other roles are for creating the presentation layout.

Java Servlet Developers

Servlets handle page-to-page navigation, session management, and simple input validation. Servlets also tie business logic elements together.

A servlet developer must understand issues related to HTTP requests, security, internationalization, and web statelessness (such as sessions, cookies, and timeouts). For iAS applications, servlets must be written in Java. Servlets are likely to call JSPs, EJBs, and JDBC RowSet objects. Therefore, a servlet developer works closely with the developers of those application elements.

JSP Developers

JSP developers work closely with servlet developers to define the application's presentation screens and storyboards (page navigation). Even on complex development projects, the same person may develop both JSPs and servlets. However, if you design the application so that most of the Java is in servlets rather than in JSPs, a JSP developer need not be proficient in Java.

JSPs are likely to call EJBs and JDBC RowSet objects.

HTML Designers

HTML designers optimize HTML pages. For example, designers might perform any of the following tasks:

- Ensure that HTML appears properly across different browser clients.
- Ensure that HTML loads efficiently across slow modem connections.
- Improve the appearance of pages initially designed by JSP developers.

Graphic Artists

Graphic artists create images that end up as GIFs or JPEGs. These images must be appropriate to the application and must be quick to download. Graphic artists work closely with HTML designers.

Client-Side JavaScript Developers

Client-side JavaScript can be used for several reasons. For example, it can handle simple input validation before passing data to the server, or it can make the user interface more exciting. Client-side JavaScript developers work closely with developers of servlets and JSPs.

Team Roles for the Business Logic Layer

The business logic layer encapsulates business rules and business entities. The following team members are needed:

- Session bean developers
- Entity bean developers

Session Bean Developers

Session beans encapsulate the logic for business processes and business rules. For example, a session bean might be developed to calculate taxes for a billing invoice. Applications usually handle complex business rules that can change frequently (for example, due to new business practices or new government regulations). As a result, an application typically uses more session beans than entity beans, and session beans may need continual revision.

A developer of session beans typically is a domain expert and understands complex, domain-specific logic as well as data validation rules. This developer works closely with developers of servlets and entity beans.

Session beans are likely to call a full range of JDBC interfaces, as well as other EJBs. Applications perform better when session beans are stateless. Here's why. Suppose tax is calculated in a stateful session bean. The application must then access a particular server where that bean's state information resides. If the server happens to be down, then application processing is delayed.

Entity Bean Developers

Entity beans represent persistent objects, such as a row in a database. The role of an entity bean developer is to design an object-oriented view of an organization's business data. Creating this object-oriented view often means mapping database tables into entity beans. For example, the developer might translate a Customer table, Invoice table, and Order table into corresponding customer, invoice, and order objects.

An entity bean developer works with developers of session beans and servlets to ensure that the application provides fast, scalable access to persistent business data.

Entity beans are likely to call a full range of JDBC interfaces. However, entity beans typically do not call other EJBs.

Team Roles for the Data and Legacy Access Layer

In the Data and Legacy Access layer, the main role is the developer of custom extensions. These developers are very likely to use C++, and they typically need to understand issues related to wrapping C++ in Java, such as Java Native Interfaces (JNI).

Extension developers use iPlanet Extension Builder. They are likely to integrate access to the following systems:

- CORBA applications
- mainframe systems
- third-party security systems

After the development team is assembled, the application's user interface can be designed.

Designing the User Interface

It is recommended that you design your application from front to back. That is, you should design the user interface before you design or develop EJBs. In this way, you ensure the most efficient application flow.

Begin by planning the navigation storyboards and UI screens. There are many third-party books and online guides that teach web site design. The following list summarizes the questions you may need to resolve:

- What is the page flow?
- What commands and buttons are available on each page?
- Will the pages use frames or not?
- Are there any corporate standards that determine headers and footers, logos, menu bars, or banner ads?
- At what point is login required?
- Are there any international issues? For example, are the icons or cartoon images meaningful to all users? Does translation pose a formatting problem?
- Are the commonly used features easy to find?

The specific tasks related to creating servlets and JSPs are covered in Part II of this guide.

Guidelines for Effective Development

This section lists some of the guidelines to consider when designing and developing a iAS application. This section is merely a summary. For more details, refer to later chapters in this guide.

The guidelines are grouped into the following goals:

- Easing Development
- Maintaining or Reusing Code
- Improving Performance
- Planning for Scalability

Easing Development

- Design the UI first; this assumes you know something about the datasources that the application must access.
- Use servlets for presentation logic; user JSPs for presentation layout.
- Develop servlets and JSPs that can conditionally generate different pages.

Maintaining or Reusing Code

- Choose base classes, helper classes, and helper methods in a way that encourages code reusability.
- Use relative paths and URLs, so that links remain valid if you move the code tree later.
- Minimize the Java in your JSPs; instead, put Java in servlets and helper classes. JSP designers can revise JSPs without being Java experts.
- Use property files or global classes to store hardcoded strings such as the names of datasources, tables, columns, JNDI objects, or other application properties.

- Use session beans, rather than servlets and JSPs, to store business rules that are domain-specific or likely to change often, such as input validation.
- Use entity beans for persistent objects; using entity beans allows management of multiple beans per user.
- For maximum flexibility, use Java interfaces rather than Java classes.
- Use extensions to access legacy data.

Improving Performance

- In most cases, deploy servlets and JSPs to iAS rather than to the iPlanet Web Server (iWS). iAS is best if an application is highly transactional, requires failover support to preserve session data, or accesses legacy data. iWS is useful if an application is mostly stateless, read-only, and non-transactional.
- Use entity beans and stateless session beans; design for colocation to avoid costly remote procedure calls.
- When an application is deployed, ensure that the necessary EJBs and JSPs are replicated and available to load into the same process as the calling servlet.
- When returning multiple rows of information, use JDBC RowSet objects when possible. When committing complex data to a database, use efficient database features, such as JDBC batch updates or direct SQL operations.
- Follow general programming guidelines for improving performance of Java applications.

Planning for Scalability

- Store scalar or serializable information in HttpSession objects that are configured to be distributed.
- Avoid using global variables.
- Design the application as if it will run in a multi-machine environment (also called a “server farm”)

Chapter 2

Controlling Applications with Servlets

This chapter describes how to create effective Java servlets to control interactions in iPlanet Application Server (iAS) applications.

Servlets, like applets, are reusable Java applications. Unlike applets, however, servlets run on an application server or web server rather than in a web browser.

In iAS, servlets make up the presentation logic of an application by acting as a central dispatcher for your application by processing form input, invoking business logic components by accessing Enterprise JavaBeans, and formatting page output using JSPs. Servlets control the application's flow from one user interaction to the next by generating content in response to a request from your user.

This chapter contains the following sections:

- Introducing Servlets
- Designing Servlets
- Creating Servlets
- Invoking Servlets

Introducing Servlets

The iAS servlets are based on the Java Servlet Specification v2.2. All specifications are accessible from *`installdir`*`/ias/docs/index.htm`, where *`installdir`* is the location in which you installed iAS.

There are three kinds of servlets: those that adhere strictly to the specification; those that take advantage both of the specification and additional iAS features; and those that adhere to the specification in non-iAS environments, but that take advantage of iAS features if they are available. This chapter describes standard servlets as well as the iAS features that augment the standards, and enables you to make the best choice for your intended deployment scenario.

The fundamental characteristics of servlets are as follows:

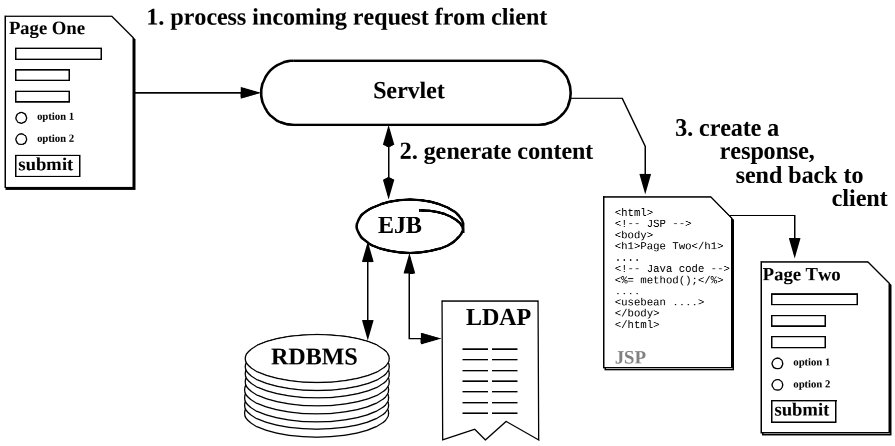
- Servlets are created and managed at run time by the servlet engine, a component of iAS that runs inside the Java server.
- The input data on which servlets operate is encapsulated in an object called the request object. A servlet's response to a query is encapsulated in an object called the response object.
- Servlets call EJBs to perform business logic functions. Servlets call JSPs to perform page layout functions.
- You can extend a servlet's functionality by using APIs provided with iAS. You are not required to use any of these APIs in order to create a servlet.
- Servlets control user sessions in order to provide some persistence of user information between interactions.
- Servlets can be a part of a particular application, or they can reside separately in order to be available to multiple applications. The latter type are said to be members of the generic ("Default") application.
- Servlets can be dynamically reloaded while the server is running.
- Servlets are addressable as URLs. The buttons on your application's pages usually point to servlets. Servlets can also call other servlets.

Several features are offered by iAS through iAS-specific APIs that enable your applications to take programmatic advantage of specific features of the iAS environment. For more information, see "Accessing Optional iAS Features" on page 51.

Servlets in iAS Applications

When a user of your application clicks a button on one of your application's pages, the information that the user entered on the page is sent to a servlet. The servlet processes the incoming data and orchestrates a response by generating content, often through the use of business logic components (Enterprise JavaBeans). Once the content is generated, the servlet creates a response page, usually by forwarding the newly generated content to a JavaServer Page (JSP). The response is delivered back to the client, which sets up the next user interaction.

The following illustration shows the life cycle of a single user interaction, the execution of a single servlet.



```
<html>
<!-- JSP -->
<body>
<h1>Page Two</h1>
....
<!-- Java code -->
<%= method();</%>
....
<usebean ....>
</body>
</html>
```

1. Process incoming request from client

When a user clicks a button to submit information to your application, form, page, and session data are passed as a parameter to a servlet in a request object. A response object is also passed which contains information about the client itself. The servlet can manipulate these objects as well as forward them to other application components.

If an instance of the servlet does not already exist, iAS instantiates it when it is called. Servlet configuration information is loaded into iAS when the servlet is instantiated.

2. Generate content

The servlet dispatches business logic tasks by invoking business objects (EJBs). The servlet may also invoke other servlets, JSPs, or other classes as needed to perform discrete tasks. The servlet gathers results in the request object.

3. **Create a response**

The servlet generates a response by either creating a browser page and sending it directly to the client, or by dispatching the task to a JSP. The servlet remains in memory, available to process another request.

For a more detailed description of a servlet's life cycle, see How Servlets Work"How Servlets Work" below.

How Servlets Work

This section describes the basic ingredients of all servlets. It shows how iAS runs servlets from the perspective of the server itself. Servlets exist in the iAS Java server process and are managed by an object called the servlet engine. The servlet engine is an internal object that handles all servlet metafunctions. These functions include instantiation, initialization, destruction, access from other components, and configuration management.

Types of Servlets

Servlets must implement the interface `javax.servlet.Servlet`. There are two main types of servlets:

- **Generic servlets** extend `javax.servlet.GenericServlet`. Generic servlets are protocol independent, meaning that they contain no inherent support for HTTP or any other transport protocol.
- **HTTP servlets** extend `javax.servlet.HttpServlet`. These servlets have built-in support for the HTTP protocol and are much more useful in an iAS environment.

For both types of servlets, you can implement the constructor method `init()` and/or the destructor method `destroy()` if you need to initialize or deallocate resources.

All servlets must implement a `service()` method. This method is responsible for handling requests made to the servlet. For generic servlets, you simply override the `service()` method to provide routines for handling requests. HTTP servlets provide a service method that automatically routes the request to another method in the servlet based on which HTTP transfer method is used, so for HTTP servlets you would override `doPost()` to process POST requests, `doGet()` to process GET requests, and so on.

Instantiating and Removing Servlets

Servlets are instantiated by the servlet engine. After the servlet is instantiated, the servlet engine runs its `init()` method to perform any necessary initializations. Override this method only if you need to perform an initial function for the life of the servlet, such as initializing a counter.

When a servlet is removed from service, the server engine calls the servlet's `destroy()` method so that the servlet can perform any final tasks and deallocate any resources it may have. Override this method to write log messages or clean up any lingering connections that would not be caught in garbage collection.

How Servlets Handle Requests

When a request is made, iAS hands the incoming data to the servlet engine, which processes the request, including form data, cookies, session information, and URL name-value pairs, into an object of type `HttpServletRequest` called the request object. Client metadata is encapsulated as an object of type `HttpServletResponse` and is called the response object. The servlet engine passes both as parameters to the servlet's **`service()`** method.

The default `service()` method in an HTTP servlet routes the request to another method based on the HTTP transfer method (POST, GET, etc.) For example, HTTP POST requests are routed to the `doPost()` method, HTTP GET requests are routed to the `doGet()` method, and so on. This enables the servlet to perform different processing on the request data depending on the transfer method. Since the routing takes place in `service()`, you generally do not override `service()` in an HTTP servlet. Instead, override `doGet()` and/or `doPost()`, etc., depending on the type of request you expect.

NOTE

The automatic routing in an HTTP servlet is based simply on a call to `request.getMethod()`, which provides the HTTP transfer method. Since request data is already preprocessed into a name-value list in iAS, you could simply override the `service()` method in an HTTP servlet without losing any functionality. However, this does make the servlet less portable, since it is now dependent on preprocessed request data.

You must override the `service()` method (for generic servlets) or the `doGet()` and/or `doPost()` methods (for HTTP servlets) to perform the tasks needed to answer the request. Very often, this means accessing EJBs to perform business transactions, collating the needed information (in the request object or in a JDBC ResultSet object), and then passing the newly generated content to a JSP for formatting and delivery back to the client.

How the Servlet Engine Allocates Resources

By default, the servlet engine creates a thread for each new request. This approach is less resource-intensive than instantiating a new copy of the servlet in memory for every request, but you must make sure to avoid threading issues, since each thread operates in the same memory space and variables can overwrite each other.

If a servlet is specifically written as single-threaded, the servlet engine creates a pool of ten (10) instances of the servlet to be used for incoming requests. If a request arrives when all instances are busy, it is queued until an instance becomes available. The number of instances in the pool is configurable in the Deployment Descriptor (i.e. the iAS-specific XML file.) Refer to Chapter 10, “Packaging for Deployment", for more details.

For tips on threading issues, see Handling Threading Issues.

Run-Time Servlet Modifications

Servlets and some other application components can be updated at run time without restarting the server. For more information, see Appendix B, “Dynamic Reloading ".

How Servlets Are Configured

Servlet configuration refers to the servlet’s metadata, information about the servlet that enables the server to create and use it in the framework of an application.

For more information about servlet configuration, see Chapter 10, “Packaging for Deployment ".

Important Servlet Files and Locations

Servlet files and other application files reside in a directory structure whose location is known to iAS as `AppPath`. This variable defines the top of a logical directory tree for the application, similarly to the document path in a web browser. By default, `AppPath` contains the value *`BasePath/`*`APPS`, where *`BasePath`* is the base iAS directory. (*`BasePath`* is also an iAS variable.)

`AppPath` and *`BasePath`* are variables held in the iAS registry, a repository for server and application metadata. See "The iAS Registry" in Chapter 10, "Packaging for Deployment "and the *Administration and Deployment Guide*.

The following table describes important files and locations for servlets:

Table 2-1 Important files and locations for servlets.

Location	Description
BasePath	Top of the iAS tree. All files in this directory are part of iAS. Defined by the iAS registry variable `BasePath`.
AppPath	Top of the application tree. Applications reside in subdirectories of this location. Defined by the iAS registry variable `AppPath`.
AppPath`/`*appName*`/*`	Top of the subtree for the application *`appName`*. Files in this directory are part of the specific application *`appName`*. This corresponds to the URL used to access the application's components; see "Invoking Servlets" on page 52.

Deploying Servlets

You normally deploy servlets with the rest of an application using the iAS Deployment Manager. You can also choose to deploy servlets manually for the purposes of testing or for updating servlets while the server is running.

For more information, see the iAS Deployment Tool *Administration and Deployment Guide*.

Designing Servlets

This section describes some of the basic design decisions you have to make when planning the servlets that help make up your application.

Web applications generally follow a request-response paradigm. A user normally interacts with a web application by following a directed sequence of completing and submitting forms. A servlet processes the data provided in each form, performs business logic functions, and sets up the next interaction.

How you design the application as a whole helps to determine how to design each servlet by defining the required input and output parameters for each interaction.

Choose a Component: Servlet or JSP

If the layout of a page is its main feature and there is little or no processing involved to generate the page, you may find it easier to use a JSP alone for the interaction.

For example, after the Online Bookstore sample application authenticates a user, it provides a boilerplate "portal" front page where the user can choose one of several tasks, including a book search, purchase selected items, etc. Since this portal conducts little or no processing itself, it could be implemented solely as a JSP.

Think of JSPs and servlets as opposite sides of the same coin. Each can perform all the tasks of the other, but each is designed to excel at one task at the expense of the other. The strength of servlets is in processing and adaptability, and since they are Java files you can take advantage of integrated development environments while you are writing them. However, performing HTML output from them involves many cumbersome `println` statements. Conversely, JSPs excel at layout tasks because they are simply HTML files and can be edited with HTML editors, though performing computational or processing tasks with them can be awkward. Choose the tool that is right for the job you undertake.

For more information on JSPs, see Chapter 3, "Presenting Application Pages with JavaServer Pages ".

Choose Servlet Type: HttpServlet or GenericServlet

Servlets that extend `HttpServlet` are much more useful in an HTTP environment, since that is what they were designed for. We recommend that all iAS servlets extend from `HttpServlet` rather than from `GenericServlet` in order to take advantage of this built-in HTTP support.

For more information, see "Types of Servlets" on page 34.

Create Standard or Non-Standard Servlets

One of the most important decisions to make with respect to the servlets in your application is whether to write them strictly according to the official specifications, which maximizes their portability, or whether to utilize the features provided by iAS as APIs. These APIs can greatly increase the usefulness of your servlets in an iAS framework.

You can also create more portable servlets that only take advantage of iAS features if the servlet is running in an iAS environment.

For more information on iAS-specific APIs, see "Accessing Optional iAS Features" on page 51.

Planning for Code Re-Use

Servlets by definition are discrete, reusable applications that run on a server. A servlet does not necessarily have to be tied to one individual application. You could create a library of servlets to be used across multiple applications.

However, there are disadvantages to using servlets that are not part of a specific application. In particular, servlets in the "default" application are configured separately from those that are part of the application.

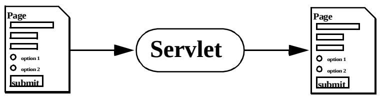
For more information, see "How Servlets Are Configured" on page 36.

How Many Servlets for Each Interaction?

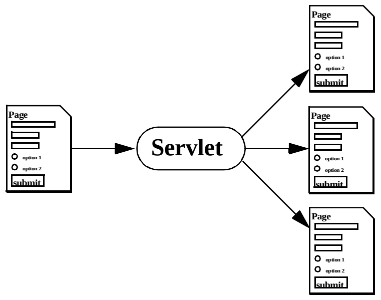
There are several possibilities for programming a given user interaction with a servlet. The method you choose for a given interaction determines how you write the servlet that handles the interaction.

One servlet handles one request

The most straightforward servlet expects one certain request and provides one certain response, as in the following diagram:

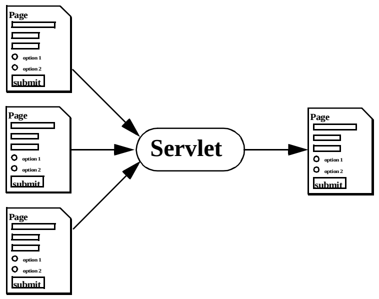


A servlet can tailor the output by setting conditions. For example, a login page could lead to a welcome page for users, an administrative page for site managers, and an error page for incorrect login, as in the following diagram:



One servlet handles multiple requests

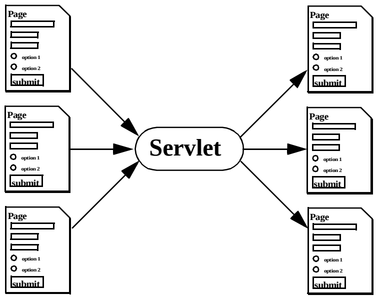
You can write a single servlet to handle many similar (or even dissimilar) requests by setting a conditional flag for each request, as in the following diagram:



For example, if a single servlet handles several steps, you could conditionalize the `service()` method based on a flag in the request object:

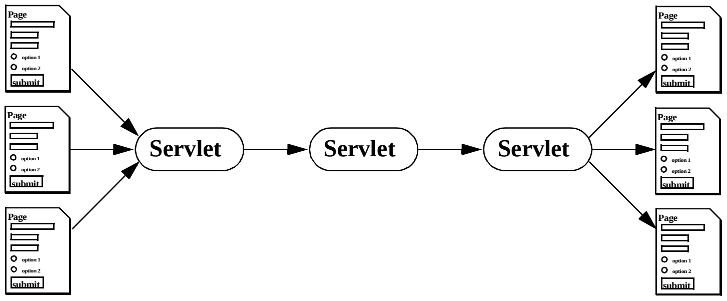
```
flag = request.getParameter("flag");
if ( flag.equals("step1")) {
    process_step1;}
else if ( flag.equals("step2")){
    process_step2;}
}
```

Additionally, the response the servlet provides can be conditional, based on the input it receives, as in the following diagram:



Several servlets handle one request

If reusable code is your goal, you can create several small servlets to handle individual functions in the course of a request, and each servlet can call other servlets for subprocessing using the `forward()` method, as in the following diagram:



You can call or include JSPs as well. By this method, you could create a library of discrete functional entities and use them wherever needed.

Creating Servlets

To create a servlet, you must perform the following tasks:

- Design the servlet into your application, or, if you want it to be accessed in a generic way, design it so that it accesses no application data.
- Create a class that extends either `GenericServlet` or `HttpServlet`, overriding the appropriate methods so that they handle requests.

- Use the iAS Administration Tool (iASAT) to create a Web-application Deployment Descriptor (DD) for the servlet.

For a description of design considerations, see “Designing Servlets” on page 37. For information about creating the files that make up a servlet, see the following sections:

- Writing the Servlet’s Class File
- Creating the Servlet’s Deployment Descriptor
- Accessing Optional iAS Features

Writing the Servlet’s Class File

This section describes how to write a servlet, and the decisions you have to make about your application and your servlet’s place in it.

Creating the Class Declaration

To create a servlet, write a public Java class that includes basic I/O support as well as the package `javax.servlet`. The class must extend either `GenericServlet` or `HttpServlet`. Since iAS servlets exist in an HTTP environment, the latter is recommended.

The following example header shows the declaration of an HTTP servlet called `myServlet`:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class myServlet extends HttpServlet {
    servlet methods...
}
```

Overriding Methods

Next, you must override one or more methods to provide instructions for the servlet to perform its designated task.

All of the processing done by a servlet on a request-by-request basis happens in the service methods, either `service()` for generic servlets or one of the `do`*`Operation`*`()` methods for HTTP servlets. This method accepts the incoming request, processes it according to the instructions you provide, and directs the output appropriately. You can create other methods in a servlet as well.

Business logic may involve accessing a database to perform a transaction, or passing the request to an EJB.

Overriding init

You can override the class initializer `init()` if you need to initialize or allocate resources for the life of the servlet instance, such as a counter. The `init()` method runs after the servlet is instantiated but before it accepts any requests. For more information, see the servlet API specification.

Note that all `init()` methods must call `super.init(ServletConfig)` in order to set their scope correctly. This makes the servlet's configuration object available to the other methods in the servlet.

The following example `init()` method initializes a counter by creating a public integer variable called `thisMany`:

```
public class myServlet extends HttpServlet {
    int thisMany;

    public void init (ServletConfig config) throws ServletException
{
        super.init(config);
        thisMany = 0;
    }
```

Now other methods in the servlet can access this variable.

Overriding destroy

You can override the class destructor `destroy()` to write log messages or to release resources that would not be released through garbage collection. The `destroy()` method runs just before the servlet itself is deallocated from memory. For more information, see the servlet API specification.

For example, the `destroy()` method could write a log message like this, based on the example for `Overriding init` above:

```
out.println("myServlet was accessed " + thisMany " times.\n");
```

Overriding service, doGet, and doPost

When a request is made, iAS hands the incoming data to the servlet engine, which processes the request, including form data, cookies, session information, and URL name-value pairs, into an object of type `HttpServletRequest` called the request object. Client metadata is encapsulated as an object of type `HttpServletResponse` and is called the response object. The servlet engine passes both objects as parameters to the servlet's `service()` method.

The default service() method in an HTTP servlet routes the request to another method based on the HTTP transfer method (POST, GET, etc.) For example, HTTP POST requests are routed to the doPost() method, HTTP GET requests are routed to the doGet() method, and so on. This enables the servlet to perform different processing on the request data depending on the transfer method. Since the routing takes place in service(), you generally do not override service() in an HTTP servlet. Instead, override doGet() and/or doPost(), etc., depending on the type of request you expect.

The automatic routing in an HTTP servlet is based simply on a call to request.getMethod(), which provides the HTTP transfer method. In iAS, request data is already preprocessed into a name-value list by the time the servlet sees the data, so you could simply override the service() method in an HTTP servlet without losing any functionality. However, this does make the servlet less portable, since it is now dependent on preprocessed request data.

You must override the service() method (for generic servlets) or the doGet() and/or doPost() methods (for HTTP servlets) to perform the tasks needed to answer the request. Very often, this means accessing EJBs to perform business transactions, collating the needed information (in the request object or in a JDBC ResultSet object), and then passing the newly generated content to a JSP for formatting and delivery back to the client.

Most operations that involve forms use either a GET or a POST operation, so for most servlets you override either doGet() or doPost(). Note that you can implement both methods to provide for both types of input, or simply pass the request object to a central processing method, as in the following example:

```
public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
            throws ServletException, IOException {
    doPost(request, response);
}
```

All of the actual request-by-request traffic in an HTTP servlet is handled in the appropriate do*Operation*() method, including session management, user authentication, dispatching EJBs and JSPs, and accessing iAS features.

If you have a servlet that you intend to also call using a RequestDispatcher method include() or forward() (see “Calling a Servlet Programmatically” on page 54), be aware that the request information is no longer sent as HTTP POST, GET, etc. RequestDispatcher methods always call service(). In other words, if a servlet overrides doPost(), it may not process anything if another servlet calls it, if the calling servlet happens to have received its data via HTTP GET. For this reason, be sure to implement routines for all possible types of input, as explained above.

NOTE

Arbitrary binary data, like uploaded files or images, can be problematic, since the web connector translates incoming data into name-value pairs by default. You can program the web connector to properly handle this kind of data and package it correctly in the request object. Accessing Parameters and Storing Data

Incoming data is encapsulated in a request object. For HTTP servlets, the request object is of type `HttpServletRequest`. For generic servlets, the request object is of type `ServletRequest`. The request object contains all the parameters in a request, and you can also set your own values in the request. The latter are called *attributes*.

You can access all the parameters in an incoming request by using the `getParameter()` method. For example:

```
String username = request.getParameter("username");
```

You can also set and retrieve values in a request object using `setAttribute()` and `getAttribute()`, respectively. For example:

```
request.setAttribute("favoriteDwarf", "Dwalin");
```

This reveals one way to transfer data to a JSP, since JSPs have access to the request object as an implicit bean. For more information, see “Using Java Beans” in Chapter 3, “Presenting Application Pages with JavaServer Pages".

Handling Sessions and Security

From the perspective of a web server or application server, a web application is a series of unrelated server hits. There is no automatic recognition that a user has visited the site before, even if their last interaction was mere seconds before. A session provides a context between multiple user interactions by “remembering” the application state. Clients identify themselves during each interaction by way of a cookie, or, in the case of a cookie-less browser, by placing the session identifier in the URL.

A session object can store objects, such as tabular data, information about the application’s current state, and information about the current user. Objects bound to a session are available to other components that use the same session.

For more information, see Chapter 11, “Creating and Managing User Sessions ".

Upon a successful login, you can direct a servlet to establish the user’s identity in a standard object called a session object that holds information about the current session, including the user’s login name and whatever other information you want to retain. Application components can then query the session object to obtain authentication for the user.

To provide a secure user session to your application, see Chapter 12, “Writing Secure Applications".

Accessing Business Logic Components

In the iAS programming model, you implement business logic, including database or directory transactions and complex calculations, in EJBs. A pointer to the `request` object can be passed as a parameter to an EJB, which then performs the specified task.

You can store the results from database transactions in JDBC ResultSet objects and pass pointers to these objects on to other components for formatting and delivery to the client. You can also store the results in the request object using the method `request.setAttribute()`, or in the session using the method `session.putValue()`. Objects stored in the request object are only valid for the length of the request, or in other words for this particular servlet thread. Objects stored in the session persist for the duration of the session, which can span many user interactions.

JDBC ResultSets are not serializable, and thus can not be distributed among multiple servers in a cluster. For this reason, do not store ResultSets in distributed sessions. For more information, see Chapter 11, “Creating and Managing User Sessions ".

This example shows a servlet accessing an EJB called ShoppingCart by creating a handle to the cart by casting the user’s session ID as a cart after importing the cart’s remote interface. The cart is stored in the user’s session.

```
import cart.ShoppingCart;

   // Get the user’s session and shopping cart
   HttpSession session = request.getSession(true);
   ShoppingCart cart =
(ShoppingCart)session.getValue(session.getId());

   // If the user has no cart, create a new one
   if (cart == null) {
        cart = new ShoppingCart();
        session.putValue(session.getId(), cart);
   }
```

You can access EJBs from servlets by using the Java Naming Directory Interface (JNDI) to establish a handle, or proxy, to the EJB. You can then refer to the EJB as a regular object; any overhead is managed by the bean’s container.

This example shows the use of JNDI to look up a proxy for the shopping cart:

```
String jndiNm = "java:comp/env/ejb/ShoppingCart";
javax.naming.Context initCtx;
Object home;
   try
    {
      initCtx = new javax.naming.InitialContext(env);
    }
    catch (Exception ex)
    {
       return null;
    }
    try
    {
       java.util.Properties props = null;
       home = initCtx.lookup(jndiNm);
    }
    catch(javax.naming.NameNotFoundException e)
    {
       return null;
    }
    catch(javax.naming.NamingException e)
    {
       return null;
    }
    try
    {
       IShoppingCart cart = ((IShoppingCartHome) home).create();

}
catch (...) {...}
```

For more information on EJBs, see Chapter 4, “Introducing Enterprise JavaBeans".

Handling Threading Issues

By default, servlets are not thread-safe. The methods in a single instance of each servlet can be executed numerous times (up to the limit of available memory) simultaneously, with each execution occurring in a different thread though only one copy of the servlet exists in the servlet engine.

This arrangement is efficient in how it uses system resources, but it can be dangerous because of how memory is managed in Java. Because parameters (objects and variables) are passed by reference, different threads can overwrite the same memory space as a side effect.

To make a servlet (or a block within a servlet) thread-safe, do one of the following:

- Synchronize access to all instance variables, as in the following example:

`public synchronized void` *`method`*`()` `(whole method) or synchronized(this) {...} (block only). Because synchronizing slows response time considerably, synchronize only blocks, or write your blocks so that they do not need to be synchronized.`

For example, this servlet has a thread-safe block in `doGet()` and a thread-safe method called `mySafeMethod()`:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class myServlet extends HttpServlet {

public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
            throws ServletException, IOException {
    //pre-processing
    synchronized (this) {
        //code in this block is thread-safe
    }
    //other processing;
    }

public synchronized int mySafeMethod (HttpServletRequest request) {
    //everything that happens in this method is thread-safe
    }
}
```

- Create a single-threaded servlet by implementing `SingleThreadModel`. In this case, when you register a single-threaded servlet with iAS, the servlet engine creates a pool of 10 servlet instances (i.e., 10 copies of the same servlet in memory) that can be used for incoming requests. The number of servlet instances in this pool can be changed by setting the `number-of-singles element` in the iAS specific web application deployment descriptor (DD) to a different number. The iAS Deployment Tool is used to modify this number in the iAS specific web application DD. Refer to Chapter 10, “Packaging for Deployment" in this document and the iASDT *Administration and Deployment Guide* for more iAS web application DD details. A single-threaded servlet can be slower under load because new requests must wait for a free instance in order to proceed, but this is less of a problem with distributed, load-balanced applications since the load automatically shifts to a less busy `kjs` process.

For example, this servlet is completely single-threaded:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class myServlet extends HttpServlet
    implements SingleThreadModel {
    servlet methods...
}
```

Delivering Results to the Client

The final activity of a user interaction is to provide a response page to the client. The response page can be delivered to the client in two ways:

- Creating a Response Page in a Servlet
- Creating a Response Page in a JSP

Creating a Response Page in a Servlet

You can generate the output page within a servlet by writing to the output stream. The recommended way to do this depends on the type of output.

You must always specify the output MIME type using `setContentType()` before any other output commences, as in this example:

```
response.setContentType("text/html");
```

For textual output, such as plain HTML, create a `PrintWriter` object and then write to it using `println`. For example:

```
PrintWriter output = response.getWriter();
output.println(“Hello, World\n”);
```

For binary output, you can write to the output stream directly by creating a `ServletOutputStream` object and then writing to it using `print()`. For example:

```
ServletOutputStream output = response.getOutputStream();
output.print(binary_data);
```

Note that your servlet can not call a JSP if you create a `PrintWriter` or `ServletOutputStream` object.

NOTE

If you are using iAS with iPlanet Web Server (iWS), do not set the date header in the output stream using `setDateHeader()`. Doing so results in a duplicate date field in the HTTP header of the response page that the server returns to the client. This is because NES automatically provides this header field. Conversely, Microsoft Internet Information Server (IIS) does *not* add a date header, so you must provide one in your code.

Creating a Response Page in a JSP

Servlets can invoke JSPs in two ways:

- The `include()` method in the `RequestDispatcher` interface calls a JSP and waits for it to return before continuing to process the interaction. The `include()` method can be called multiple times within a given servlet.

This example shows a JSP invocation using `include()`:

```
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("JSP_URI");
dispatcher.include(request, response);
... //processing continues
```

- The `forward()` method in the `RequestDispatcher` interface hands control of the interaction to a JSP. The servlet is no longer involved with output for the current interaction after invoking `forward()`, thus only one call to the `forward()` method can be made in a particular servlet. Note that you can not use the `forward()` method if you have already defined a `PrintWriter` or `ServletOutputStream` object.

This example shows a JSP invocation using `forward()`:

```
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("JSP_URI");
dispatcher.forward(request, response);
```

NOTE

You identify which JSP to call by specifying a URI (Universal Resource Identifier). The path is a `String` describing a path within the scope of the `ServletContext` and must begin with a backslash (”/”). There is also a `getRequestDispatcher()` method in the Request Object that takes a `String` argument indicating a complete path. See the Java Servlet Specification, v2.2 section 8 for more details about this method.

For more information about JSPs, see Chapter 3, “Presenting Application Pages with JavaServer Pages".

Creating the Servlet’s Deployment Descriptor

Servlet deployment descriptors (DDs) are created by the iAS Deployment Tool (iASDT). These descriptors are packaged within the Web Application ARchive (.war) files that contain metadata, information about the servlet that identifies it and establishes its role in the application.

Deployment Descriptor Elements

The deployment descriptor for a servlet contains standard J2EE specified elements as well as iAS specific elements. The servlet DDs conveys the elements and configuration information of a web application between Developers, Assemblers, and Deployers. For a description of these elements refer to Chapter 10, “Packaging for Deployment".

Dynamically Reloading Servlets

You can reload servlets into iAS without restarting the server. Simply overwrite the servlet by redeploying. iAS “notices” the new component and reloads it within 10 seconds. For more information, see Appendix B, “Dynamic Reloading".

To load a new servlet configuration file into iAS, perform the following steps to make the changes active:

1. Stop the server.
2. Deploy the new servlet XML war file.
3. Restart the server.

Accessing Optional iAS Features

The iAS provides many additional features to augment your servlets for use in an iAS environment. These features are not a part of the official specifications, though some are based on emerging Sun standards and will conform to those standards in the future.

For more details on iAS features, see Chapter 13, “Taking Advantage of iAS Features ".

The iAS also provides support for more robust sessions, based on a model from a previous version of iAS. This model uses the same API as the session model described in the servlet 2.2 specification, which is also supported. For more details on distributable sessions, see Chapter 11, “Creating and Managing User Sessions".

Invoking Servlets

You invoke a servlet either by directly addressing it from an application page with a URL, or by calling it programmatically from another servlet that is already running.

Calling a Servlet With a URL

Most of the time, you call servlets using URLs embedded as links in your application's pages. This section describes how to invoke servlets using standard URLs.

Invoking Specific Application Servlets

The URL request path that leads to a servlet servicing a request is composed of several sections. Each of these sections has an important purpose in locating the appropriate servlet. The request object will expose the following elements obtained from the request URI path:

- Context Path
- Servlet Path
- PathInfo

For more information on these elements see *section 5.4 Request Path Elements of the Java Servlet Specification, v2.2.*

Address servlets that are part of a specific application as follows:

http://*server*:*port*/NASApp/*appName*/*servletName*?*name*=*value*

Each section of the URL is described in the table below:

Table 2-2 URL fields for Servlets within a Specific Application

URL element	Description
server:*port*	Address and optional port number for the web server handling the request.
NASApp	Indicates to the web server that this URL is for a iAS application, so the request is routed to the iAS executive server.
appName	The application's name.

Table 2-2 URL fields for Servlets within a Specific Application

URL element	**Description**
`servletName`	The servlet's name, as configured in the XML file.

The application name *appName* must correspond to a directory under *AppPath* (see "Important Servlet Files and Locations" on page 37). This directory contains files extracted out during registration, like .jsps, and static content files like .html, .jpg and so on. For example:

```
http://www.my-company.com/NASApp/OnlineBookings/directedLogin
```

Invoking Generic Application Servlets

Address servlets that part of the generic application as follows:

`http://`*`server`*`:`*`port`*`/servlet/`*`servletName`*`?`*`name`*`=`*`value`*

Each section of the URL is described in the table below:

A

Table 2-3 URL Fields for Servlets within a Generic Application

URL element	**Description**
server:*port*	Address and optional port number for the web server handling the request.
`servlet`	Indicates to the web server that this URL is for a generic servlet object.
servletName	The servlet's name, as specified in the servlet-name element in the web app XML file.
?*name*=*value*...	Optional name-value parameters to the servlet.

For example:

```
http://www.leMort.com/servlet/calcMortgage?rate=8.0&per=360&bal=180
000
```

NOTE

All servlets deployed to use the "/servlet" path, should be deployed as part of the "default" application.

Calling a Servlet Programmatically

You can call a servlet programmatically from another servlet in two ways, as described below.

Note that you identify which servlet to call by specifying a URI, or Universal Resource Identifier. This is normally a path relative to the current application. For instance, if your servlet is part of an application called `Office`, the URL to a servlet called `ShowSupplies` is shown below. The bold section is the portion of the URI to use in the call:

`http://`*`server`*`:`*`port`*`/NASApp/Normal/ShowSupplies?name=value`

- Include another servlet's output using the `include()` method from the `RequestDispatcher` interface. `include()` calls a servlet by its URI and waits for it to return before continuing to process the interaction. `include()` can be called multiple times within a given servlet.

For example:

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/ShowSupplies");
dispatcher.include(request, response);
```

- Hand control of the interaction to another servlet using the `forward()` method in the `RequestDispatcher` interface. takes the servlet's URI as a parameter.

Note that forwarding a request means that the original servlet is no longer involved with output for the current interaction after invoking `forward()`, thus only one `forward()` call can be made in a particular servlet.

This example shows a servlet invocation using `forward()`:

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/ShowSupplies");
dispatcher.forward(request, response);
```

NOTE

Both mechanisms of invoking servlets, either programatically, or using the include()/forward() mechanism, or from the URL, can use the URL patterns for the servlet specified in the DD XML, as well as the <servlet-name> entry. For example,

if an XML entry for a servlet in the web.xml file is

```
<servlet-name>Foo</servlet-name>
<servlet-mapping>
 <servlet-name>Foo</servlet-name>
<url-pattern>Bar</url-pattern>
</servlet-mapping>
```

then you can access the servlet either as

http://s:p/NASApp/AppName/Foo

   or

http://s:p/NASApp/AppName/Bar

where

s: server name
p: port number
AppName: name of the application

Chapter 3

Presenting Application Pages with JavaServer Pages

This chapter describes how to use JavaServer Pages (JSPs) as page templates in iPlanet Application Server (iAS) web applications.

This chapter contains the following sections:

- Introducing JavaServer Pages
- How JSPs Work
- Designing JSPs
- Creating JSPs
- Programming Advanced JSPs
- Deploying JSPs
- Invoking JSPs
- JSP 1.1 Tag Summary
- Value-added Features

Introducing JavaServer Pages

JavaServer Pages (JSPs) are browser pages in HTML or XML. They can optionally contain Java code, which enables them to perform complex processing, conditionalize output, and communicate with other objects in your application. JSPs in iAS 6.0 are based on the JSP 1.1 specification. The specifications are accessible from `installdir/ias/docs/index.htm`, where *`installdir`* is the location where you installed iAS.

In an iAS application, you use JSPs as the individual pages that make up your application. You can call a JSP from a servlet to handle the output from a user interaction, or, since JSPs have the same access to the application environment as any other application components, you can use a JSP as a destination from an interaction.

How JSPs Work

A JSP is made up of JSP elements and template data. *Template data* refers to anything not in the JSP specification, including text and HTML tags. For example, the minimal JSP, which requires no processing by the JSP engine, is a static HTML page.

The iAS compiles JSPs into HTTP servlets the first time they are called. This makes them available to the application environment as standard objects, and it also enables them to be called from a client using a URL.

JSPs run inside a Java process on the server. This process, called a JSP engine, is responsible for interpreting JSP-specific tags and performing the actions they specify in order to generate dynamic content. This content, along with any template data surrounding it, is assembled into a page for output and returned to the caller.

The response object contains a reference to the calling client, and this is where a JSP presents the page that it creates. If you call a JSP from a servlet using the `forward()` method from the `RequestDispatcher` interface, `forward()` provides the response object as a parameter to the JSP. If a JSP is invoked directly from a client, the response object is provided by the server that is managing the relationship with the caller.

In either case, the page is automatically returned to the client through the reference provided in the response object. You do not have to write any code to return a page to a client.

You can create JSPs that are not a part of any particular application. These JSPs are considered to be part of a generic application. JSPs can also run in the iPlanet Web Server (iWS) and other web servers, but these JSPs have no access to any application data, which makes their use limited.

JSPs and some other application components can be updated at run time without restarting the server, making it easy to change the look and feel of your application without stopping service. For more information, see Appendix B, “Dynamic Reloading" in this Programmer’s Guide.

Designing JSPs

This section describes some of the decisions you must consider when you write JSPs. Since JSPs are compiled into servlets, the design considerations for servlets are also relevantto JSPs. See Chapter 2, “Controlling Applications with Servlets".

The information on a page can loosely be categorized into page layout elements, which consists of tags and information pertaining to the structure of the page, as well as page content elements, which consists of the actual information your page presents to the user.

You design page layout as you would design any browser page, interleaving content elements where needed. For example, one page element might be a welcome message (“Welcome to our application!”) at the top of the page. You could personalize this message with a call to the user’s name after authentication (“Welcome to our application, Mr. Einstein!”).

Since page layout is a more or less straightforward task, the design decisions you must make are related more to the way the JSP interacts with the rest of the application and how it is optimized.

The following sections describe specific design issues:

- Choose a Component: Servlet or JSP
- Designing for Ease of Maintenance: How Many JSPs?
- Designing for Portability: Generic JSPs

Choose a Component: Servlet or JSP

If the layout of a page is its main feature and there is little or no processing involved to generate the page, you may find it easier to use a JSP alone for the interaction.

Think of JSPs and servlets as opposite sides of the same coin. Each can perform all the tasks of the other, but each is designed to excel at one task at the expense of the other. The strength of servlets is in processing and adaptability, and since they are Java files you can take advantage of integrated development environments while you are writing them. However, performing HTML output from them involves many cumbersome `println` statements that must be coded by hand. Conversely, JSPs excel at layout tasks because they are simply HTML files and can be created with HTML editors, though performing computational or processing tasks with them can be awkward. Choose the tool that is right for the job you undertake.

For example, the following simple component is presented as both a JSP and a servlet for comparison. This component, which performs no complex content generation activities, works best as a JSP:

JSP:

```
<html><head><title>Feedback</title></head><body>
<h1>The name you typed is: <% request.getParameter("name"); %>.</h1>
</body></html>
```

Servlet:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class myServlet extends HttpServlet {
    public void service (HttpServletRequest req,
                         HttpServletResponse res)
                throws ServletException, IOException
    {
    response.setContentType("text/html");
    PrintWriter output = response.getWriter();
    output.println("<html><head><title>Foo</title></head>"
                 + "<body>\n"
                 + "<h1>The name you typed is:"
                 + req.getParameter("name") + ".</h1>"
                 + "</body></html>";
    }
}
```

For more information on servlets, see Chapter 2, “Controlling Applications with Servlets".

Designing for Ease of Maintenance: How Many JSPs?

Each of your JSPs can call or include any number of other JSPs. For example, you could have a generic corporate banner, a standard navigation bar, and left-side column table of contents, each of which resides in a separate JSP that you include for each page you build. This type of page can also be constructed with a JSP functioning as an frameset, dynamically determining the pages to load into each of its sub-frames.

A JSP can be included when the JSP is compiled into a servlet or when a request arrives.

Designing for Portability: Generic JSPs

You can write JSPs that are completely portable between different applications and different servers. They have a disadvantage in that they have no knowledge of application data in particular, but this is only a problem if they require that kind of data.

One possible use for generic JSPs is as portable page elements, like navigation bars or corporate headers and footers, that are meant to be included in other JSPs. You can create a library of reusable generic page elements to use throughout your application, or even among several applications.

For example, the minimal generic JSP is a static HTML page with no JSP-specific tags. A slightly less minimal JSP might contain some Java code that operates on generic data, such as printing the date and time, or makes a change to the page's structure based on a standard value in the request object.

Creating JSPs

You create JSPs in much the same way that you create static HTML files, including using a WYSIWYG page editor if that is your preference. JSP elements, including directive elements, scripting elements, and action elements, as described in the following sections.

The following sections describe how to use JSP-specific tags in HTML files in order to create JSPs. If you use an HTML editor to create pages and edit their layout, you can create a JSP by inserting these tags into the raw source code at the appropriate points.

This section contains the following subsections:

- General Syntax
- Java Directives
- Java Scripting Elements
- Java Actions
- Implicit Objects

General Syntax

JSPs that adhere to the JSP 1.1 specification follow XML syntax for the most part, which is consistent with HTML. In other words, tags are demarcated with < and >, constructs have a start tag (`<tag>`) and end tag (`</tag>`), and tags are case-sensitive, such that `<tag>` is different from `<Tag>` or `<TAG>`.

In general, you insert JSP tags inline in the file where they are required, as if they were standard HTML tags. For example, if the request contains a parameter `name` that contains the user's name, you could write a welcome sentence like this:

```
<p>Hello, <%= request.getParameter("name"); %>.</p>
```

JSP Tags

JSP tags use the form `<jsp:`*`tag`*`>`, a form taken from XML. Some tags (particularly scripting tags) have a shortcut for use in HTML files, generally starting with `<%` and ending with `%>`.

Note that these shortcuts are not valid for XML files.

Empty elements, tag constructs that have nothing between the start and end tags, can be shortened to one tag ending with `/>`, as in this example:

```
<!-- include tag with no body: -->
<jsp:include page="/corporate/banner.jsp"></jsp:include>

<!-- include tag can also be written like this: -->
<jsp:include page="/corporate/banner.jsp" />
```

White space is not usually significant, although you should make sure to put a space character between the opening tag and any attributes. For example:
`<%= myExpression %> is valid, <%=myExpression %>` is not.

Escape Characters

In attributes, where you may find difficulty with nested single and double quotes, use the following escape characters:

- `'` is quoted as `\'`
- `"` is quoted as `\"`
- `%>` is quoted as `%\>`
- `<%` is quoted as `<\%`

Comments

There are two types of comments in JSPs:

- comments to the JSP page itself, documenting what the page is doing
- comments that appear in the generated document sent to the client

JSP Comments

A JSP comment is contained within `<%--` and `--%>`, and can contain anything except the text `--%>`. The following example, therefore is incorrect:

```
<%-- anything but a closing --%> ... --%>
```

An alternative way to place a comment in a JSP is to use a Java comment. For example:

```
<% /** this is a comment ... **/ %>
```

Generating Comments in Output to Client

In order to generate comments that appear in the response output stream to the requesting client, use the HTML and XML comment syntax as in the following example:

```
<!-- comments ... -->
```

These comments are treated as uninterpreted template text by the JSP engine. If the generated comment is to have dynamic data, this can be obtained through an expression syntax, as in the following example:

```
<!-- comments <%= expression %> more comments ... -->
```

Java Directives

```
<%@ directive { attr="value" }* %>
```

Use directives to set preferences within the JSP. Each directive has a number of attributes that affect the behavior or state of the JSP.

Valid directives are:

- `<%@ page%>`
- `<%@ include%>`
- `<%@ taglib... %>`

<%@ page%>

Syntax

```
<%@ page language="java"
         extends="className"
         import="className{,+}"
         session="true|false"
         buffer="none|sizeInKB"
         autoFlush="true|false"
         isThreadSafe="true|false"
         info="text"
         errorPage="jspUrl"
         isErrorPage="true|false"
         contentType="mimeType{;charset=charset}"
%>
```

Description

Sets page-level preferences for the JSP.

Attributes

Valid attributes include the following:

Table 3-1 Java `page` Directive

Attribute	Valid Values	Description
`language`	`java`	Default: `java`. Scripting language for this JSP. Currently, iAS only supports `java`.
`extends`	valid Java class name	Defines a specific superclass for this JSP. This restricts the JSP engine in many ways, and should be avoided if possible.
`import`	comma-separated list of valid Java class names	Types and classes that are available to other methods in this JSP. This is identical to the `import` statement in a Java class.

Table 3-1 Java `page` Directive

Attribute	Valid Values	Description
`session`	`true` or `false`	Default: `true`. Indicates that the page must participate in an HTTP session. If `language=java` and `session=true`, this option creates an implicit variable called `session` which points to or creates a session of type `javax.servlet.http.HttpSession`
`buffer`	`none`, or buffer size in kilobytes	Defines an output buffer. If this is set to `none`, all output is written directly to the output stream (a `PrintWriter` object). If a size is provided, then either the buffer is flushed or an exception is raised when it is filled with output. The behavior is determined by the autoFlush attribute.
`autoFlush`	`true` or `false`	Determines behavior when output buffer is full. If true, output is flushed to the output stream when buffer is full. If false, an exception is raised when the buffer is full.
`isThreadSafe`	`true` or `false`	Default: `false`. Indicates the level of thread safety implemented in the page. The value determines behavior inside the JSP engine: if `true`, multiple requests can be made to the JSP instance simultaneously, otherwise multiple requests are handled serially. For the most part, you should ensure that your JSP is thread-safe regardless of this setting, as this setting has no effect on shared objects such as sessions or contexts.
`info`	text	A string incorporated into the translated page that can subsequently be obtained from the page's implementation of the `Servlet.getServletInfo()` method.

Table 3-1 Java page Directive

Attribute	Valid Values	Description
errorPage	valid URL for a JSP error page	Error page for this JSP. This error page must be a JSP. Any Throwable object that is thrown but not caught by the original page are forwarded to the error page. The error page has an implicit variable called exception that contains a reference to the un-caught exception. Note that if autoFlush=true, then if the contents of the initial JspWriter has been flushed to the ServletResponse output stream (i.e., if part of the page has already been sent to the client) then any subsequent attempt to invoke an error page may fail.
isErrorPage	true or false	Default: false. Indicates whether the current JSP page is the possible target of another JSP page's errorPage. If true, then the implicit variable exception is defined and its value is a reference to the offending Throwable from the source JSP page in error.
contentType	content type, optionally with charset	Default: text/html;charset=ISO-8859-1 Defines the MIME type and character encoding for the response. Values are either of the form *TYPE* or *TYPE;*charset=*CHARSET*

Examples

```
<%@ page errorpage="errorpg.htm" %>
<%@ page import="java.io.*,javax.naming.*" %>
```

<%@ include%>

Syntax

```
<%@ include file="file" %>
```

Description

The include directive enables you to include other JSPs (or static pages) when the JSP is compiled into a servlet. The resource is treated as if it were a part of the JSP.

There is another method for including other resources: the `<jsp:include>` action, which includes the resource at request-time. For more information on file inclusion, see "Including Other Resources" on page 81.

Attributes

The include directive has only one attribute, as described below:

Table 3-2 Java `include` Directive

Attribute	Valid Values	Description
`file`	Valid URL (absolute) or URI (relative path)	The file to be included.

The file attribute can either be relative to the current JSP, or absolute to the application's context root. For relative file attributes, the file name should not begin with a backslash ('/'), but for absolute file attributes, the file name should begin with a backslash ('/').

Example

If Foo.jsp is a JSP file in the application MyApp (typically located in $installdir/ias/APPS/MyApp), and Foo.jsp contains the following tag:

```
<%@include file="/bar/baz.jsp"%>
```

then the system tries to include the file baz.jsp from `$installdir/ias/APPS/MyApp/bar/baz.jsp`.

If baz.jsp contains the following tag:

```
<%@include file="boo.jsp"%>
```

then the system will also include the file **$installdir/ias/APPS/MyApps/bar/boo.jsp**.

<%@ taglib... %>

Syntax

```
<%@ taglib uri="uriToTagLibrary" prefix="prefixString" %>
```

The `taglib` directive enables you to create your own custom tags. For more information on creating your own tags, see section "Value-added Features" on page 91.

Attributes

The taglib directive specifies two attributes, as described below:

Table 3-3 Java `taglib` Directive

Attribute	Valid Values	Description
`uri`	Valid URI (relative path)	The URI (Universal Resource Identifier) is either an absolute (from the application's context root) or a relative reference to a .tld XML file, that describes the tag library. The URI can also be an alias that is unaliased by the <taglib> entry in the web application descriptor of the JSP. (see JSP v1.1 specification section 5.2 Tag Library for more details)
`prefix`	String	A prefix for the custom tag.

Example

Consider the following JSP file, Foo.jsp, in the application MyApp, and a corresponding XML deployment file with a web-app section as follows:

```
<taglib>
   <taglib-uri> http://www.mytaglib.com/spTags </taglib-uri>
   <taglib-location> /foo/bar/baz.tld</taglib-location>
<taglib>
```

```
The JSP file contains the following:
<%@ taglib uri="http://www.mytaglib.com/spTags" prefix="mytags" %>
<mytags:specialTag attribute="value"> ... </mytag:specialTag>
```

The JSP engine looks inside the web app descriptor to find a matching taglib-location for "`http://www.mytaglib.com/spTags`". The engine locates `/foo/bar/baz.tld`, and therefore looks for an XML file `$installdir/ias/APPS/MyApp/foo/bar/baz.tld`. This is the Tag Library Descriptor file that describes the tags used in the file.

The URI, or taglib-location (if the URI is aliased), can also be relative. In this case the `.tld` file is searched for relative to the current directory. *See JSP v1.1 specification, section 5.2 Tag Library for more details.*

Java Scripting Elements

Scripting elements are made up of the following types of tags:

- Java Declarations <%! ... %>
- Java Expressions <%= ... %>
- Java Scriptlets <%...%>

There are several implicit objects available to your scripts, including the request and response objects. For more information about implicit objects, see “Implicit Objects” on page 79.

Java Declarations <%! ... %>

Syntax

`<%! declaration %>`

Description

Use declarations to define variables that are valid throughout the JSP. You can declare anything legal in Java, including methods, as long as the declaration is complete. Nothing appears in the output stream as a result of a declaration.

Example

```
<%! int i=0; %>
<%! String scriptname="myScript"; %>
<%! private void myMethod () { ... } %>
```

Java Expressions <%= ... %>

Syntax

`<%= expression %>`

Description

Use expressions as variables to be evaluated. The value of the expression is substituted where the expression occurs. The result appears on the output stream.

Example

```
<p>My favorite color is <%= userBean.favColor %>.</p>
```

Java Scriptlets <%...%>

Syntax

`<% `*`script`*` %>`

Description

Use scriptlets to define blocks of code to be executed. Any legal code can appear here,

Example

```
<% int balance = request.getAttribute("balance");
   if (balance < LIMIT) {
      println (UNDERLIMIT_ALERT);
   }
   String balString = formatAsMoney(balance);
%>
Your current balance is <%= balance %>.
```

Java Actions

Actions, as the name implies, perform some activity, such as creating/loading a Java bean or setting or retrieving bean properties, including other JSPs, or specifying required plugins.

Some actions allow request-time expressions as parameters, enabling you to set values for these attributes dynamically for the request.For examples, see the attribute in question. The attributes that allow expressions as parameters are the `value` and `name` attributes of `<jsp:setProperty>` and the `page` attribute of `<jsp:include>` and `<jsp:forward>`.

Standard actions are described as follows:

- `<jsp:useBean>` creates or accesses Java beans
- `<jsp:setProperty>` sets bean properties
- `<jsp:getProperty>` retrieves bean properties
- `<jsp:include>` includes other JSPs or HTML pages at request-time
- `<jsp:forward>` forwards execution control to another JSP
- `<jsp:plugin>` dynamically loads browser plugins for special data types

<jsp:useBean>

Syntax

```
<jsp:useBean id="name" scope="scope"
             class="className" |
             class="className" type="typeName" |
             beanName="beanName" type="typeName" |
             type="typeName">
// optional body
</jsp:useBean>
```

Description

The `<jsp:useBean>` action tries to find a Java bean with the given name (`id`) and scope (`scope`). If one exists, it is made available, otherwise it is created using the provided name, scope, and type/class information. A variable called *`name`*, specified with the attribute `id="`*`name`*`"`, is made available to the JSP so that you can access the object if the action succeeds.

`<jsp:useBean>` can be an empty tag, as in `<jsp:useBean ... />`, or it can contain other actions and close with the end tag `</jsp:useBean>`. Other actions that normally appear here are `<jsp:setProperty>` actions that set properties in the (possibly newly-created) bean. Template text, other scripts or declarations, etc. are treated normally. Note that the body of a `<jsp:useBean>` tag is executed only once, when the bean is created.

The `<jsp:useBean>` action must specify a unique `id="`*`name`*`"` attribute. If the action succeeds in creating or accessing an object, this name makes the object available to scripting tags further down in the JSP.

Attributes

Valid attributes are as follows:

Table 3-4 Attributes for <jsp:useBean>

Attribute	Description
`id`	Unique identifying name for the object.

Table 3-4 Attributes for <jsp:useBean>

Attribute	Description
`scope`	Lifecycle for this object. One of the following: `page`: Object is valid for this page only, even if the request encompasses more than one page. The object is not forwarded to other pages. `request`: Object is bound to the request object (retrieve with `getAttribute(name)` where *`name`* is the object's `id`), and is thus available for the life of this request. `session`: Object is bound to the session object (retrieve with `getValue(`*`name`*`)` where *`name`* is the object's `id`) and is thus available wherever the session is available, for the life of the session. A session must be active for this JSP in order to use this scope. `application`: Object is bound to the `ServletContext` (retrieve with `getAttribute(`*`name`*`)` where *`name`* is the object's `id`) and is thus available for the existence of the application, unless it is specifically destroyed.
`class`	Valid classname for this bean, used to instantiate the bean if it does not exist. If `type` is specified, `class` must be assignable to `type`. `beanName` and class can not both be specified for the same bean.
`beanName`	Valid name of a bean, of the form *`a.b.c`* (classname) or *`a/b/c`* (resource name). `beanName` and `class` can not both be specified for the same bean. The `beanName` attribute can be an expression, evaluated at request-time.
`type`	Defined variable type for this bean. This attribute enables the type of the variable to be distinct from that of the implementation class specified. The type is required to be either the class itself, a superclass of the class, or an interface implemented by the class specified. If unspecified, the value is the same as the value of the `class` attribute.

Examples

This example shows the creation of a bean, or access to a bean that already exists, called `currentUser` of type `com.netscape.myApp.User`:

```
<jsp:useBean id="currentUser" class="com.netscape.myApp.User" />
```

In this example, the object should have been present in the session. If so, it is given the local name wombat with `WombatType`. A `ClassCastException` may be raised if the object is of the wrong class, and an `InstantiationException` may be raised if the object is not defined.

```
<jsp:useBean id="currentUser"
             type="com.netscape.myApp.User"
             scope="session" />
```

Also see the **Examples** section for `<jsp:setProperty>`, below.

<jsp:setProperty>

Syntax

```
<jsp:setProperty name="beanName"
                 property="propertyName"
                 param="requestParameter" | value="value"
</jsp:setProperty>
```

Description

The `<jsp:setProperty>` action sets the value of properties in a bean. It is used in the body of a `<jsp:useBean>` tag to set bean properties. The property's value may be determined with an expression. It may also come directly from the request object.

Attributes

Valid attributes are as follows:

Table 3-5 Attributes for `<jsp:setProperty>`

Attribute	Description
`name`	Name of the bean in which you want to set a property. The name must be defined previously in the file with `<jsp:useBean>`.
`property`	The name of the bean property whose value you want to set. The property must be a valid property in the bean. If `propertyName="*"` then the tag iterates over the parameters in the request object, matching parameter names and value type(s) to property names and setter method type(s) in the bean, setting each matched property to the value of the matching parameter. If a parameter has an empty value, the corresponding property is not modified. Note that any previous value for that parameter persists.

Table 3-5 Attributes for `<jsp:setProperty>`

Attribute	Description
`param`	The name of the request object parameter whose value you want to give to a bean property. If you omit `param`, the request parameter name is assumed to be the same as the bean property name If the `param` is not set in the request object, or if it has an empty value, the `<jsp:setProperty>` action has no effect. A `<jsp:setProperty> a`action may not have both `param` and `value` attributes.
`value`	The value to assign to the given property. This attribute can accept an expression as a value; the expression is evaluated at request-time. A `<jsp:setProperty>` action may not have both `param` and `value` attributes.

Examples

In this example, the `name` and `permissions` properties are set:

```
<jsp:useBean id="currentUser" class="com.netscape.myApp.User" >
    <jsp:setProperty name="currentUser"
                     property="name"
                     param="name">
    <jsp:setProperty name="currentUser"
                     property="permissions"
                     param="permissions">
</jsp:useBean>
```

`These examples both set a property name to the value of a corresponding request parameter also called name:`

```
<jsp:setProperty name="myBean" property="name" param="name" />
```

```
<jsp:setProperty name="myBean" property="name"
                 value="<%= request.getParameter(\"name\" %>)" />
```

<jsp:getProperty>

Syntax

```
<jsp:getProperty name="beanName"
                 property="propertyName">
```

Description

`A <jsp:getProperty> action places the value of a bean property, converted to a String, into the output stream.`

Attributes

Valid attributes are as follows:

Table 3-6 Attributes for `<jsp:getProperty>`

Attribute	Description
`name`	Name of the bean from which you want to retrieve a property. The name must be defined previously in the file with `<jsp:useBean>`.
`property`	The name of the bean property whose value you want to retrieve. `property` must be a valid property in the bean.

Examples

```
<jsp:getProperty name="currentUser" property="name" />
```

<jsp:include>

Syntax

<code><jsp:include page="*uri*" flush=”true”/></code>

Description

A `<jsp:include>` action includes the specified page in the current page at request-time, preserving the current page's context. Using this method, the included page is written to the output stream.

There is another method for including other resources: the `<%@ include%>` directive, which includes the resource at compile-time. For more information on file inclusion, see "Including Other Resources" on page 81.

Attributes

Valid attributes are as follows:

Table 3-7 Attributes for `<jsp:include>`

Attribute	Description
`page`	This is either an absolute or relative reference to the page being included. For absolute references, this field begins with a backslash (“/”), and is rooted at the application’s context root. For relative references, this field is relative to the JSP file performing the include, and may contain an expression that is evaluated at request time.
`flush`	Determines whether the included page is to be flushed to the output stream.

Examples

```
<jsp:include page="/templates/copyright.html" flush="true" />
```

<jsp:forward>

Syntax

```
<jsp:forward page="URL" />
```

Description

The `<jsp:forward>` action allows the runtime dispatch of the current request to a static resource, a JSP page, or a Java servlet in the same context as the current page, terminating the execution of the current page. This action is identical to the `forward()` method of the `RequestDispatcher` interface.

Attributes

Valid attributes are as follows:

Table 3-8 Attributes for `<jsp:forward>`

Attribute	Description
`page`	Valid URL pointing to the page to be included. This attribute may contain an expression which is evaluated at request-time, as long as this expression must evaluate to a valid URL.

NOTE

If page output is unbuffered (as with `<% page buffer="none" %>`) and data has already been written to the output stream, this tag results in a runtime error.

Examples

```
<jsp:forward page="/foo/handleAlternativeInput.jsp" />
```

The following element might be used to forward to a static page based on some dynamic condition.

```
<% String whereTo = "/templates/"+someValue; %>
<jsp:forward page="<%= whereTo %>" />
```

<jsp:plugin>

Syntax

```
<jsp:plugin type="bean|applet"
            code="objectCode"
            codebase="objectCodebase"
            { align="alignment" }
            { archive="archiveList" }
            { height="height" }
            { hspace="hspace" }
            { jreversion="jreversion" }
            { name="componentName" }
            { vspace="vspace" }
            { width="width" }
            { nspluginurl="URL" }
            { iepluginurl="URL" } >
            { <jsp:params
                  <jsp:param name=" paramName" value="paramValue" />
              </jsp:params> }
            { <jsp:fallback> fallbackText </jsp:fallback> }
</jsp:plugin>
```

Description

The `<jsp:plugin>` action enables a JSP author to generate HTML that contains the appropriate client browser dependent constructs (`object` or `embed`) to instruct the browser to download (if required) an appropriate Java plug-in and execute an Applet or JavaBeans component. The attributes of the `<jsp:plugin>` tag provide configuration data for the presentation of the element.

The `<jsp:plugin>` tag is replaced by either an `<object>` or `<embed>` tag, as appropriate for the requesting user agent, and emitted into the output stream of the response.

There are two related actions that are only valid within a `<jsp:plugin>` action:

- `<jsp:params>` sets off a block containing parameters to the Applet or JavaBeans component. Individual parameters are set with
- <jsp:param name="name" value="value">
- The section ends with `</jsp:params>`. The names and values are component-dependent.
- `<jsp:fallback>` indicates the content to be used by the client browser if the plugin cannot be started (either because `object` or `embed` is not supported by the client browser, or due to some other problem). The body of this tag is presented to the browser in the event of a failure of the surrounding `<jsp:plugin>`. For example:

```
<jsp:plugin ...>
   <jsp:fallback><b>Plug-in could not be started!</b></jsp:fallback>
</jsp:plugin>
```

If the plugin can start but the Applet or JavaBeans component cannot be found or started, a plugin-specific message is usually presented to the user, often as a pop-up window reporting a `ClassNotFoundException`.

Attributes

The `<jsp:plugin>` tag takes most of its attributes from the HTML `<applet>` and `<object>` tags (`<applet>` is defined in HTML 3.2, `<object>` in HTML 4.0). Refer to the official HTML specifications where these tags are introduced:

- HTML 3.2: `http://www.w3.org/TR/REC-html32.html`
- HTML 4.0: `http://www.w3.org/TR/REC-html40/`

Valid attributes for the `<jsp:plugin>` tag are as follows.

Table 3-9 Attributes for <jsp:plugin>

Attribute	Description
`type`	Identifies the type of the component, `bean` or `applet`.
`code`	As defined by the HTML spec.
`codebase`	As defined by the HTML spec.
`align`	As defined by the HTML spec.
`archive`	As defined by the HTML spec.
`height`	As defined by the HTML spec.

Table 3-9 Attributes for <jsp:plugin>

Attribute	Description
`hspace`	As defined by the HTML spec.
`jreversion`	Identifies the specification version number of the JRE the component requires in order to operate. Default: `1.1`
`name`	As defined by the HTML spec.
`vspace`	As defined by the HTML spec.
`title`	As defined by the HTML spec.
`width`	As defined by the HTML spec.
`nspluginurl`	URL where JRE plugin can be downloaded for Netscape Navigator, default is implementation defined.
`iepluginurl`	URL where JRE plugin can be downloaded for Microsoft Internet Explorer, default is implementation defined.

Examples

```
<jsp:plugin type="applet"
            code="Tetris.class"
            codebase="/html" >
    <jsp:params>
        <jsp:param name="mode" value="extraHard"/>
    </jsp:params>

    <jsp:fallback>
        <p> unable to load Plugin </p>
    </jsp:fallback>
</jsp:plugin>
```

Implicit Objects

The JSP 1.1 specification defines some objects that are available implicitly for every JSP. They can be referred to from anywhere in your JSP, without previously defining them (for example, with `<jsp:useBean>`).

The following objects are available implicitly for every JSP:

Table 3-10 Implicitly Available Objects for Every JSP

Object	Description	Scope	Java type
`request`	The request that triggered this JSP's execution.	request	protocol-dependent subtype of `javax.servlet.ServletRequest`, e.g. `javax.servlet.HttpServletRequest`
`response`	The response to the request (i.e., the returned page and its path back to the caller).	page	protocol-dependent subtype of `javax.servlet.ServletResponse`, e.g. `javax.servlet.HttpServletResponse`
`pageContext`	The page context for this JSP.	page	`javax.servlet.jsp.PageContext`
`session`	The session object (if any) created for or associated with the caller.	session	`javax.servlet.http.HttpSession`
`application`	The servlet context for this JSP, from the servlet's configuration object via `getServletConfig().getContext()`.	applicatio n	`javax.servlet.ServletContext`
`out`	An object that writes to the output stream.	page	`javax.servlet.jsp.JspWriter`
`config`	The servlet configuration object (`ServletConfig`) for this JSP.	page	`javax.servlet.ServletConfig`
`page`	The instance of this page's class that is processing the current request. page is a synonym for the keyword this.	page	`java.lang.Object`

Table 3-10 Implicitly Available Objects for Every JSP

Object	Description	Scope	Java type
`exception`	For error pages only, the uncaught `Throwable` exception that caused the error page to be invoked.	page	`java.lang.Throwable`

For example, you can refer to the request object as follows:

One of the request parameters was `<%= request.getParameter("param"); %>.`

Programming Advanced JSPs

This section provides some instructions for using advanced programing techniques. This section includes the following subsections:

- Including Other Resources
- Using Java Beans
- Accessing Business Objects

Including Other Resources

An important feature of JSPs is their ability to dynamically include other page-generating resources, or their results, at run-time. You can include the contents of a static HTML page, or you can process a separate JSP and include its results in your output page.

For example, corporate headers and footers can be included on each page appropriately by creating page stubs that contain just the included elements. Note that it is possible to include entire pages on a conditional basis, which provides much more flexibility than simply inserting flat navigation bars or corporate headers.

There are two ways to include a resource in your JSP:

- the `<%@ include%>` directive:

```
<%@ include file="filename" %>
```

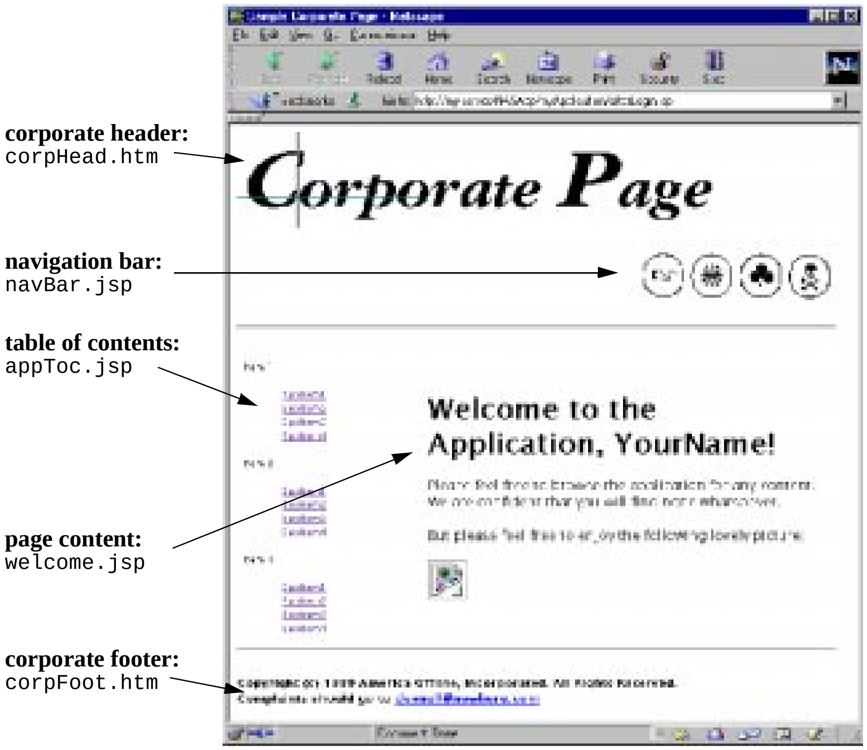
- the `<jsp:include>` action:

```
<jsp:include page="uri" flush="true" />
```

If you include a resource using the `<%@ include%>` directive, the resource is included when the JSP is compiled into a servlet, and is thus treated as if it were a part of the original JSP. If the included resource is also a JSP, its contents are processed along with the parent JSP. For more information, see “Java Directives” on page 63.

If you include a resource using the `<jsp:include>` action, the included resource is included when the JSP is requested by a caller. For more information, see “Java Actions” on page 70.

The following example shows how each portion of a page can be provided by a separate resource, accessed from a single JSP. The source code for this example page shows both methods for including resources: static resources are included using the `<jsp:include>` action, dynamic resources are included using the `<%@ include%>` directive.


corporate header: corpHead.htm
navigation bar: navBar.jsp
table of contents: appToc.jsp
page content: welcome.jsp
corporate footer: corpFoot.htm

Corporate Page

Welcome to the Application, YourName!

afterLogin.jsp:

```
<html><head><title>Sample Corporate Page</title></head><body>

<p align="left"><jsp:include page="corpHead.htm" flush="true" /></p>
<%@ include file="navBar.jsp" %>
<hr size="3">

<table border=0><tr>
<td width="25%"><%@ include file="appToc.jsp" %></td>
<td width="75%"><%@ include file="appToc.jsp" %></td>
</tr></table>

<hr>
<p align="left"><jsp:include page="corpFoot.htm" flush="true" /></p>
</body></html>
```

Using Java Beans

JSPs support several tags for the purpose of instantiating and accessing Java beans. You use beans to perform computations in order to obtain a set of results, which are stored as bean properties. JSPs provide automatic support for creating beans and for examining their properties.

Beans themselves are separate classes created according to the Java Bean specification. It is beyond the scope of this manual to describe how to create standard Java beans. For more information about Java beans, visit the official web site `http://java.sun.com/beans`. There are many tutorials that describe how to create beans, some of which are accessible from the official site.

It is common in beans to have "getter" and "setter" methods to retrieve and set bean properties. Getter methods are named `getXxx()`, where `Xxx` is a property called `xxx` (the first letter is capitalized for the method name). If you have a corresponding setter called `setXxx()`, the setter must take a parameter of the same type as the return value from the getter.

NOTE

Note that this support is for standard Java beans, not EJBs. To access EJBs from your JSP, see "Accessing Business Objects" on page 84.

NOTE

In NAS 4.0, which supported the JSP 0.92 specification, the request and response objects were accessed via “implicit beans”. This support has changed in the JSP 1.1 specification; several objects, including the request and response objects, are available implicitly, with varying degrees of scope. See “Implicit Objects” on page 79.

Accessing Business Objects

Because JSPs are compiled into servlets at run-time, they have full access to the rest of the processes running in the server, including EJBs. You can access beans or servlets in the same way you would access them from a servlet, as long as the Java code you write is embedded inside an escape tag.

The method described here for contacting EJBs is identical to the method used from servlets. For more information, see “Accessing Business Logic Components” in Chapter 2, “Controlling Applications with Servlets" in the *Programmer’s Guide*.

This example shows a JSP accessing an EJB called ShoppingCart by creating a handle to the cart by casting the user’s session ID as a cart after importing the cart’s remote interface:

```
<%@ import cart.ShoppingCart %>;
...
<% // Get the user’s session and shopping cart
   ShoppingCart cart =
(ShoppingCart)session.getValue(session.getId());

   // If the user has no cart, create a new one
   if (cart == null) {
        cart = new ShoppingCart();
        session.putValue(session.getId(), cart);
   } %>
...
<%= cart.getDataAsHTML() %>
```

```
This example shows the use of JNDI to look up a proxy, or handle, for
the cart:
```

```
<% String jndiNm = "java:/comp/ejb/ShoppingCart";
   javax.naming.Context initCtx;
   Object home;
       try {
             initCtx = new javax.naming.InitialContext(env);
       } catch (Exception ex) {
             return null;
       }
       try {
             java.util.Properties props = null;
             home = initCtx.lookup(jndiNm);
       }
       catch(javax.naming.NameNotFoundException e)
       {
             return null;
       }
       catch(javax.naming.NamingException e)
       {
             return null;
       }
       try {
              IShoppingCart cart = ((IShoppingCartHome)
home).create();
             ...
      } catch (...) {...}
%>
...
<%= cart.getDataAsHTML() %>
```

NOTE

You must usually provide a method in the EJB to convert raw data to a format acceptable on the page, such as `getDataAsHTML()` in the examples above.

For more information on contacting EJBs, see “Accessing Business Logic Components” in Chapter 2, “Controlling Applications with Servlets".

Deploying JSPs

There are two ways iAS deploys JSPs; they are deployed as either unregistered JSPs or as registered JSPs.

Unregistered JSPs

Unregistered JSPs are deployed by simply being copied into the corresponding directory (the name of which is the same as the application name), in the AppPath. These JSPs can then be invoked using the URL access as follows:

- http://s:p/AppPrefix/AppName/JSPFileName

Refer to the “Invoking JSPs” on page 87 for more details.

Registered JSPs

The iAS allows for JSPs to be registered with GUIDs, using XMLs, a la Servlets. This allows for JSPs to use iAS-value-added features like load balancing. This is performed by registering XML files with the <jsp-file> entry as detailed in the Servlet 2.2 specification.

The following XML file is an example of a deployment descriptor for a registered JSP:

```
 MyApp.xml:
    <?xml version="1.0" ?>
    <!DOCTYPE web-app>
    <web-app>
      <display-name> An Example Registered JSP File </display-name>
       <servlet>
            <servlet-name>JSPExample</servlet-name>
             <jsp-file>JSPExample.jsp</jsp-file>
       </servlet>
        <servlet-mapping>
            <servlet-name>JSPExample</servlet-name>
            <url-pattern>/jspexample</url-pattern>
        </servlet-mapping>
   </web-app>

ias-MyApp.xml:
     <?xml version="1.0" ?>
     <ias-web-app>
        <servlet>
            <servlet-name>JSPExample</servlet-name>
            <guid>{aaaabbbb-A456-161A-8be4-0800203942f2}</guid>
         </servlet>
    </ias-web-app>
```

In the above example, the JSP is registered with the GUID specified in the ias-MyApp.xml file. Although this example indicates that the servlet name is JSPExample, it does not mean that use of the .jsp extension is required. Its completely possible to have the servlet name as JSPExample.jsp instead. This JSP can then be accessed from a URL by using one of the following:

- http://s:p/AppPrefix/MyApp/JSPExample
- http://s:p/AppPrefix/MyApp/JSPExample.jsp (use if the servlet-name entry in the XML file is JSPExample.jsp)

Invoking JSPs

You can invoke a JSP programmatically from a servlet, or you can address it directly from a client using a URL. You can also include JSPs as well as invoke them; see "Including Other Resources" on page 81.

Calling a JSP With a URL

You can call JSPs using URLs embedded as links in your application's pages. This section describes how to invoke servlets using standard URLs.

Invoking JSPs in a Specific Application

JSPs that are part of a specific application are addressed as follows:

`http://`*`server`*`:`*`port`*`/AppPrefix/`*`appName`*`/`*`jspName`*`?`*`name`*`=`*`value`*

Each section of the URL is described in the table below:

URL element	Description
server:*port*	Address and optional port number for the web server handling the request.
AppPrefix	Indicates to the web server that this URL is for a iAS application, so the request is routed to the iAS executive server. This is configured using the registry entry SSPL_APP_PREFIX.
appName	The application's name, which must be identical to the name of the application's subdirectory under `AppPath`.

URL element	Description
`jspName`	The JSP's filename, including the `.jsp` extension.
`?name=value...`	Optional name-value parameters to the JSP. These are accessible from the `request` object.

For example:

http://www.my-company.com/AppPrefix/OnlineBookings/directedLogin.jsp

NOTE

The use of a generic application for a JSP has the same requirements and restrictions as a generic application for a servlet. There needs to be an application called "Default" that must have an XML file registered. Any URL request that accesses a servlet or a JSP with the /servlet/ entry is forwarded to the generic application "Default". This requirement is detailed in section "Invoking Generic Application Servlets," of Chapter 2, "Controlling Applications with Servlets".

Invoking JSPs in the Generic Application

JSPs that are not part of a specific application are addressed as follows:

http://*server*:*port*/servlet/*jspName*?*name*=*value*

Each section of the URL is described in the table below:

URL element	Description
`server:port`	Address and optional port number for the web server handling the request.
`servlet`	Indicates to the web server that this URL is for a generic servlet object.
`jspName`	The JSP's name, including the `.jsp` extension.
`?name=value...`	Optional name-value parameters to the JSP. These are accessible from the `request` object.

For example:

http://www.leMort.com/servlet/calcMort.jsp?rate=8.0&per=360&bal=180000

Invoking a JSP From a Servlet

A servlet can invoke a JSP in one of two ways:

- The `include()` method in the `RequestDispatcher` interface calls a JSP and waits for it to return before continuing to process the interaction.
- The `forward()` method in the `RequestDispatcher` interface hands control of the interaction to a JSP.

For more information on these methods, see “Delivering Results to the Client” in Chapter 2, “Controlling Applications with Servlets" in the Programmer’s Guide.

An example of this is as follows:

```
public class ForwardToJSP extends HttpServlet
{
   public void service(HttpServletRequest req,
                       HttpServletResponse res)
   throws ServletException, IOException
  {
     RequestDispatcher rd = req.getRequestDispatcher("/test.jsp");
     rd.forward(req, res);
  }
}
```

JSP 1.1 Tag Summary

Directives

```
<%@ page|include|taglib { attr="value" }* %>

    attr: page language="java"
               extends="className"
               import="className{,+}"
               session="true|false"
               buffer="none|sizeInKB"
               autoFlush="true|false"
               isThreadSafe="true|false"
               info="text"
               errorPage="jspUrl"
               isErrorPage="true|false"
               contentType="mimeType{;charset=charset}"

          include file="filename"

         taglib uri="uriToTagLibrary"
                prefix="prefixString"
```

See “Java Directives” on page 63.

Expressions

<%= expression %>

See “Java Scripting Elements” on page 69.

Scriptlets

<% scriptlet %>

See “Java Scripting Elements” on page 69.

Comments

`<%-- comment --%>`	*JSP comment, not passed to client*
`<!-- comment -->`	*standard HTML comment, passed to client*
`<% /** comment **/ %>`	*Java comment, encapsulated in scriptlet, passed to client*

See “Comments” on page 63.

Bean-Related Actions

```
<jsp:useBean id="name" scope="scope"
             class="className" |
             class="className" type="typeName" |
             beanName="beanName" type="typeName" |
             type="typeName">
// optional body
</jsp:useBean>

<jsp:setProperty name="beanName"
                 property="propertyName"
                 param="requestParameter" | value="value"
</jsp:setProperty>

<jsp:getProperty name="beanName"
                 property="propertyName">
```

See “Java Actions” on page 70.

Other Actions

```
<jsp:include page="relativeUrl"
             flush="true" />


<jsp:forward page="URL" />


<jsp:plugin type="bean|applet"
            code="objectCode"
            codebase="objectCodebase"
            { align="alignment" }
            { archive="archiveList" }
            { height="height" }
            { hspace="hspace" }
            { jreversion="jreversion" }
            { name="componentName" }
            { vspace="vspace" }
            { width="width" }
            { nspluginurl="URL" }
            { iepluginurl="URL" } >
            { <jsp:params
                 <jsp:param name=" paramName" value="paramValue" />
              </jsp:params> }
            { <jsp:fallback> fallbackText </jsp:fallback> }
</jsp:plugin>
```

See “Java Actions” on page 70.

Value-added Features

PagePath

A static (HTML) or dynamic (JSP) page is looked up by first looking inside the directory identified by $AppPath (assigned to the registry key \\SOFTWARE\Netscape\Application Server\4.0\AppPath), and, then, if not found, inside any of the directories identified by $PagePath (registry key: \\SOFTWARE\Netscape\Application Server\6.0\PagePath). This is an optional key that is not present by default, and will need to be added, if used. Whereas $AppPath is a single directory, $PagePath is a list of directories, separated by the path-list character of the current system (':' on Unix, and ';' on NT).

A dynamic page is compiled into the directory $AppPath/compiled_jsp, where both its class and intermediate source file are stored. Hence, a dynamic page should have a unique path relative to $AppPath and $PagePath to avoid confusion. Also, note, that the rules for relative paths specified in JSP 1.1 apply to both AppPath and PagePath, i.e. a uri to a page is relative to the current application context.

Hence, suppose A.jsp is stored as:

$AppPath/myApp/dir/A.jsp, and includes B.jsp as <jsp:include page="B.jsp"/>, then the JSP engine will first look for B.jsp in $AppPath/myApp/dir, and then in $PagePath[0]/myApp/dir, $PagePath[1]/myApp/dir, etc.

Custom Tag Extensions

The JSP v1.1 specification details a protocol to support user-defined custom tags. (See JSP v1.1 specification Chapter 5 Tag Extensions.) Although the specification does not mandate any tags, as a value-add to iAS users, iAS ships with certain custom tags that follow the JSP 1.1-defined protocol for tag extensions.

Some of these tags are meant to provide support for LDAP and database querying, and others provide support for using conditionals inside JSPs, for which there's no primitive support as part of the specification.

In order to support JSP page-caching, iAS ships with a Cache tag library. Details on how this is used are also provided in this chapter's section "JSP Page Caching" on page 109.

The other tags included within iAS are for internal use only, specifically to support converting GX tags in NAS 4.0 applications. These tags are used in generating JSP 1.1 pages from JSP 0.92 pages (which supported GX tags), and should not be used anywhere externally.

The following tag libraries are introduced by iAS:

- Query
- LDAP
- Conditional
- Attribute

Database Query TagLib

The query taglib supports declarative declarations of RowSets in a JSP page, along with loops for looping through result-sets and a display tag fordisplaying column values.

List of Query tags:

1. <rdbm:useQuery id="export_name"
   scope="[page | request | session | application]" command="select * from..."
   queryFile="foo.gxq" queryName="firstQuery"
   execute="[true | false]" dataSourceName="jdbc/..."
   url="odbc:...">...</rdbm:useQuery>

2. <rdbm:param query="query-declaration-export-name"
   name="name-of-parameter" value="value" bindOnLoad="[true | false]"
   type="[String | Int | Double | Float | BigDecimal | Date | Boolean | Time | Timesta
   mp"
   format="java-format-string for dates">value</rdbm:param>

3. <rdbm:loop id="export_name" scope="[page | request | session | application]"
   query="query-declaration-export-name"
   start="[request-parameter-name | request-attribute-name | last | constant]"
   max="integer-maximum-number-of-rows"
   execute='{true | false]">...</rdbm:loop>

4. <rdbm:field query="query-declaration-export-name" name="field name"
   format="format for doubles" urlEncode="{false/true}">default

   value</rdbm:field>

5. <rdbm:close resource="query-declaration-export-name"/>

6. <rdbm:execute query="query-declaration-export-name"/>

7. <rdbm:goRecord query="query-declaration-export-name"
   execute="{false/true}"
   start="[request-parameter-name | request-attribute-name | last | constant]">defa
   ult
   start</rdbm:goRecord>

useQuery tag

The *useQuery* tag declares a use of a result-set. The *useQuery* tag allows the tool to understand what query is being made, and hence, what fields are available for use. If the result set is already in the scope that the useQuery wants to save to, then the body of this tag is skipped, and, although the RowSet is currently created, nothing is done with the created RowSet.

If the result-set does not already exist, the created RowSet is exported using the id attribute of the *useQuery* tag at the specified *scope*, which defaults to request. Either the *command* specified is used, or a query located in the *queryFile* is loaded. The name of the loaded query is either the name located in the *queryName* attribute, or, if none is specified, the value of the tag's *id* attribute. Once the rowset is initialized, it may be executed if the *execute* tag is specified. It is important to execute a query if you want to use the field tag outside of a loop.

If the query is loaded from a file rather than specified in the *command* attribute, the file is loaded and cached by the `QueryLoader` class. The two attributes *queryFile* and *queryName* work in conjunction. The *queryFile* locates the query file. If the value of the attribute is a relative file specification, then the file is looked up in the `RDBMS.path.query` path. This path should be specified as a System property as a series of semi-colon delimited directories. If this variable is not set, then the NAS-specific `GX.path.query` property is used instead. The file is **not** located relative to the jsp itself. The query file should look something like:

```
query name1 [using (ODBC, cdx, netscape)] is
select *
from foo, bar
where :whereClause  /* :whereClause is an example of a bindOnLoad, named
parameter */

query name2 is
select * from foo, bar where foo.x = bar.y and foo.name = :name /* :name is an
example of a named parameter */
```

A blank line (without spaces, tabs, etc!) separates queries which are named using the *query ... is* construct.

Param tag

The *Param* tag sets a parameter on a RowSet. The name of the parameter can either be an index or the actual name of the parameter as saved in the Dictionary. Note that *bindOnLoad* parameters **MUST** exist in the body of the *useQuery* tag before any non-bindOnLoad parameters. The value of the parameter is either the value stored in the value attribute, or the contents of the param tag's body. Because JSP1.1 tags don't generally nest (<%= ... %> being the notable exception), the only way to bind a parameter to a value from another query would be to place a field tag within the body of a parameter tag and have the parameter tag use its body as the value, which is why you can place the value in the boy of the tag.

Param tags may exist within the *useQuery* tag, in which case they set the parameters directly against their parent query, or elsewhere, probably before a loop tag re-executes the RowSet, in which case the parameter is set on the RowSet that the *useQuery* tag exported.

Loop tag

The *loop* tag loops through the contents of a ResultSet. The *query* attribute is used to locate the ResultSet or an enclosing *useQuery* tag. The *start* attribute is used to indicate the starting position of the loop. Start may either refer to a parameter, or to an attribute, which is looked up using PageContext.findAttribute() or a constant Integer value. That value then indicates which record number to start on, or *"last"*, which will cause the RowSet to be scrolled to the end, and then back *max* rows. The max attribute is used to indicate the maximum number of records to display.

If `execute` is specified, then the RowSet is executed before looping begins.

Field tag

The *field* tag displays a particular column in the ResultSet. The *query* attribute locates the enclosing *useQuery* tag or a previous *useQuery* tag's exported ResultSet. The *name* attribute identifies the column name to display. The *format* attribute allows one to format Strings, numbers, or dates into the appropriate type. The *Attribute* urlEncode can be used to encode the strings. If the column is null, the body of the *field* tag is output.

Close tag

The close tag releases resources back to the system. The *resource* attribute locates the exported query resource (ResultSet) and calls close() on it.

Execute tag

The *execute* tag executes the identified query.

goRecord tag

The goRecord tag optionally executes the specified query and moves the resultset to the record indicated by the start attribute. Start may either refer to a parameter, or to an attribute or a constant, if the start attribute is last, then the result-set is moved to the last record.

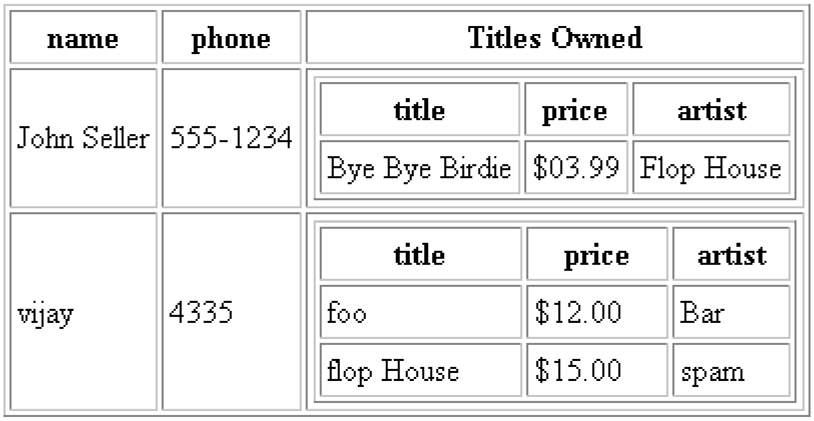
Example

The following tags produce the displayed output.

```
<HTML>
<BODY>
<%@ taglib prefix="rdbm" uri="rdbmstags6_0.tld" %>
<h2>Now let us see</h2>
<rdbm:useQuery id="a" queryFile="dbms/queries.gxq"
   dataSourceName="jdbc/cdx">
```

```
</rdbm:useQuery>
<rdbm:useQuery id="b" queryFile="dbms/queries.gxq"
    dataSourceName="jdbc/cdx">
</rdbm:useQuery>

<table border=1 cellPadding=3>
   <tr><th>name</th><th>phone</th><th>Titles Owned</th></tr>
   <rdbm:loop id="loop1" query="a" max="5" execute="true">
      <tr>
        <td><rdbm:field query="a" name="name"/></td>
        <td><rdbm:field query="a" name="phone"/></td>
        <td>
           <rdbm:param query="b" id="owner" type="Int">
           <rdbm:field query="a" name="id"/>
            </rdbm:param>
            <table border=1 cellPadding=3 width="100%">
              <tr><th>title</th><th>price</th><th>artist</th></tr>
              <rdbm:loop id="loop2" query="b" max="5"execute="true">
                <tr>
                  <td><rdbm:field query="b" name="title"/></td>
                <td><rdbm:field query="b" format="$#,###.00"
                      name="price"/>
                </td>
                  <td><rdbm:field query="b" name="artist"/></td>
                </tr>
              </rdbm:loop>
            </table>
        </td>
     </tr>
   </rdbm:loop>
   </table>
   </td>
   </tr>
</rdbm:loop>
</table>
<rdbm:close resource="a"/>
<rdbm:close resource="b"/>
</BODY>
</HTML>
```

Now let us see



name	phone	Titles Owned		
John Seller	555-1234	**title**	**price**	**artist**
		Bye Bye Birdie	$03.99	Flop House
vijay	4335	**title**	**price**	**artist**
		foo	$12.00	Bar
		flop House	$15.00	spam

LDAP Tag Library

One unfortunate aspect currently is that LDAP connections are likely to be request-specific -- that is, the current user may be the only user that is authenticated to read the LDAP attributes of that user's data. Because of this, an additional LDAPAuthenticate/Authorize type of tag is required so that the mapping between "current user" and "connection to use to perform LDAP searches" can be coded. When the LDAP server is remote and a general authorize-capable login is not available, the LDAPAuthenticate tag should be used instead.

List of LDAP Tags:

1. <ldap:[authenticate|connection] query="name of ldap exported query" url="ldap://..." password="..."> </ldap:[authenticate|connection]>
2. <ldap:authorize query="name of ldap exported query" dn="distinguished name for the user to authorize against"> </ldap:authorize>
3. <ldap:param name="parameter **name** in authenticate userDN or query url" query="name of ldap exported query" value="...">default value</ldap:param>
4. <ldap:password query="name of ldap exported query" value="...">default value</ldap:password>

5. <ldap:useQuery **id**="exported LDAPTagSearch" scope="[page|request|session|application]" url="ldap://... queryFile="filename for ldap query" queryName="name of the query in the ldap query file" connection="classname of an LDAPPoolManager" authorize="distinguished name for the user to authorize against">...</ldap:useQuery>
6. <ldap:loop[Entry] **id**="name of attribute to export loop'd value" scope="[page|request|session|application]" query="name of ldap exported query" start="request variable name" max="number" pre="number of records before jump" jump="value of sort to jump to" useVL="true/false"> </ldap:loop[Entry]>
7. <ldap:loopValue **id**="name of **attribute** to export loop'd value" scope="[page|request|session|application]" query="name of ldap exported query" entry="name of ldap exported entry from loopEntry" attribute="name of attribute to loop through" start="..." max="..."> </ldap:loopValue>
8. <ldap:field query="name of query to use" entry="name of ldap exported entry from loopEntry" url="ldap://..." **attribute**="name of attribute to display"> </ldap:field>
9. <ldap:sort query="name of ldap exported query" **order**="..."/>
10. <ldap:close **resource**="name of ldap exported query"/>

Authenticate tag (also called 'connection')

The *authenticate* tag works in the context of an `LDAPTagSearch`. The LDAPTagSearch is either retrieved from the PageContext using findAttribute and the name of the *query* attribute, or by finding a parent *useQuery* tag and gettings its LDAPTagSearch. The *url* and *password* attributes are used to authenticate the LDAPConnection which the LDAPTagSearch holds onto. If the *url* attribute has parameters (that is, if the attribute has :foo values in it after the standard `ldap://server:portNumber/` section of the ldap URL), then the body of the authenticate tag needs to contain *Param* tags for each of the parameters. If the *password* attribute is unspecified, then the body of the authenticate tag should contain a *Password* tag as well. At the end of the tag, the tag attempts to authenticate the LDAPTagSearch.

Authorize tag

The *authorize* tag works in the context of an `LDAPTagSearch`. The LDAPTagSearch is either retrieved from the PageContext using findAttribute and the name of the *query* attribute, or by finding a parent *useQuery* tag and gettings its LDAPTagSearch. The *dn* attribute is used to authorize the LDAPConnection which the LDAPTagSearch holds onto. If the *dn* attribute has parameters (that is, if the attribute has :foo values in it), then the body of the authorize tag needs to contain *Param* tags for each of the parameters. At the end of the tag, the tag attempts to authorize the LDAPTagSearch.

Param tag

The *param* tag sets parameters in LDAP URLs. LDAP Urls are specified in the *url* attribute of the *authenticate* tag, the *dn* attribute of the *authorize tag* and the *url* attribute of the *field* and *useQuery* tags.

At anyrate, params in the url are any java-legal-identifer with a prepended ':', much as the query parameters in a gxq file. For example, user in **ldap://nsdirectory.mcom.com:389/uid=:user,ou=People,dc=netscape,dc=com**. All parameters must be resolved by the end of the *field*, *authenticate*, *authorize*, or *useQuery* tags. Note 389 is not a tag, first because it's before the DN section of the LDAP Url, and second, because it isn't a java-level-identifier.

The body of the Param tag becomes the value of the parameter as named by the *name* attribute, assuming no value is specified in the Param tag itself.

Password tag

The *password* tag sets the password for the *authenticate* tag. Like the *param* tag, the body of the *password* tag becomes the value of the password, assuming that no value is specified as an attribute of the *password* tag. The *password* tag is legal only inside the *authenticate* tag.

useQuery tag

The *useQuery* tag describes the URL used to search the LDAP repository. At the end of its body, an `LDAPTagSearch` is placed into the context at the level indicated by *scope* using the name indicated by *id*. The *url* property contains the url of a *query* that, typically, a loop tag will loop through, or a *field* tag will display. This is because the *loop* tag cannot specify parameter mappings except in the body **--** which is rather too late for loop to determine if there are even any results! The *field* tag can already specify an url, and thus doesn't need to reference a query, though it can.

The url can also be loaded from a query file. The two attributes *queryFile* and *queryName* work in conjunction. The *queryFile* locates the query file. If the value of the attribute is a relative file specification, then the file is looked up in the `LDAP.path.query path`. This path should be specified as a System property as a series of semi-colon delimited directories. If this variable is not set, then the NAS-specific `GX.path.query` property is used instead. The file is **not** located relative to the jsp itself. The query file should look something like:

query name1 is
ldap://directory:389/dc=com?blah

query name2 is
ldap://directory:389/dc=org?blah

A blank line (without spaces, tabs, etc!) separates queries which are named using the *query ... is* construct.

Loop tag (also called the 'loopEnrty' tag)

The *loopEntry* tag loops through a series of LDAPEntries resulting from a search that returns multiple entries. The *query* attribute points to an exported LDAPTagSearch (see above Query tag). The *start* and *end* tags work as specifed in the Query's Loop tag. If the *useVL* attribute is true, then an {id}_contentCount value is exported which corresponds to the contentCount of the VirtualListResponse. On each pass through the loop, the currenty LDAPEntry is exported at the *scope* specified using the *id* specified. The *pre* and *jump* attributes correspond to the *beforeCount* and *jumpTo* parameters in the VirtualListControl constructor. If the loop is using a VirtualListControl, that is, if the *useVL* attribute is set, then a VirtualListControl is used to position the window of returned entries. The actual public draft url for VirtualLists is here.

LoopValue tag

The *loopValue* tag loops through a multi-value attribute of an LDAPEntry or the first LDAPEntry in an LDAPSearchResults. The *query* attribute points to an exported LDAPTagSearch (see above Query tag). If this is not specified, then the *entry* attribute points to an exported entry as specified by a containing *Loop* tag. One or the other must be specified. It is an error to specify both. The *attribute* tag names the multi-value attribute. The *start* and *end* tags work as specifed in the Query's Loop tag. On each pass through the loop, the current LDAPAttribute-value is exported at the *scope* specified using the *id* specified.

Field Tag

The *field* tag prints out the value of a single-value attribute as specified in the *query* or *url* or *entry* attributes and the *attribute* attribute. If no value exists, the body of the *field* tag is passed through. The body of the *field* tag will only be evaluated if the *url* has parameters (and hence, there would be parameter bindings in the body that need to be evaluated and set), or if the mapped value is null. If the attribute name is DN, then the distinguished name for the *entry* is returned as the value of the field.

Sort tag

The *sort* tag works in conjunction with the *useQuery* tag, setting a sort order for the enclosing query. The *query* attribute identifies the enclosing query tag (or an exported LDAPTagSearch, if the *sort* tag occurs outside of the *useQuery* tag's body). The *order* attribute specifies the sort-order, as described by the keyDescription Parameter to the LDAPSortKey constructor. The useQuery tag supports multiple Sorts. Sorts are prioritized in the order specified.

Close tag

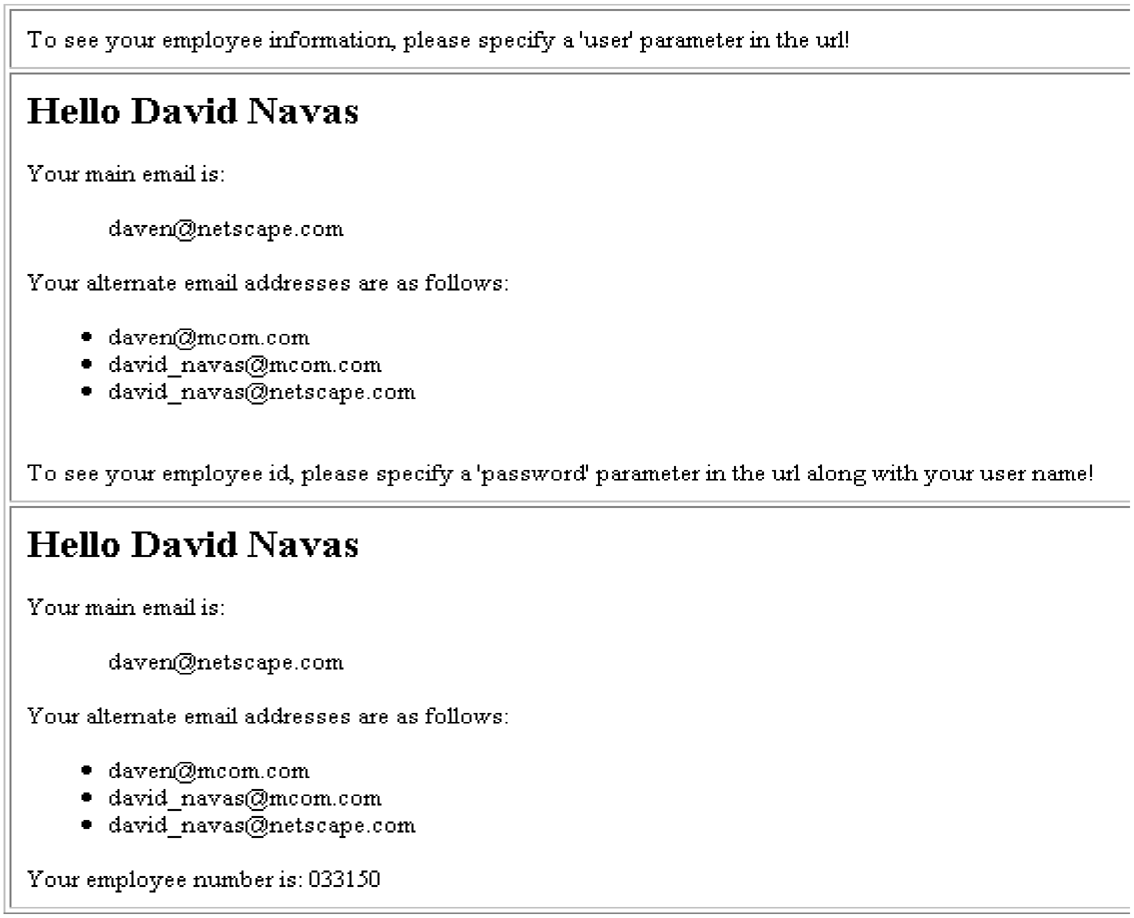
The *close* tag releases resources back to the system. The *resource* attribute locates the exported query resource (LDAPTagSearch) and calls close() on it. This calls abandon on any executing SearchResults, and releases the Connection back to the connection-pool (or disconnect()s the connection, if the connection doesn't come from a pool, which can happen when using the authenticate tag).

Example

The following example uses both the ldap tags and the switch tags, which are described below. It is assumed that the switch tags are mostly self-describing.

```
<HTML>

<BODY>

<%@ taglib prefix="cond" uri="condtags6_0.tld" %>

<%@ taglib prefix="ldap" uri="ldaptags6_0.tld" %>

<%@ taglib prefix="attr" uri="attribtags6_0.tld" %>


<cond:parameter name="user">
   <cond:exists>
      <ldap:query id="c" url="ldap://localhost:389/uid=:user,
        ou=People,dc=netscape,dc=com?cn,mailalternateaddress,mail">
        <cond:parameter name="password">
          <cond:exists>
             <ldap:authenticate query="c"
                   url="ldap://localhost:389/dc=
                   com??sub?(uid=:user)">
                <ldap:param name="user">
                   <attr:getParameter name="user" />
                </ldap:param>
                <ldap:password>
                   <attr:getParameter name="password" />
                </ldap:password>
            </ldap:authenticate>
         </cond:exists>
      </cond:parameter>
     <ldap:param name="user"><attr:getParameter name="user" />
     </ldap:param>
    </ldap:query>
    <h2>Hello
    <ldap:field query="c" attribute="cn">

No Contact Name for <attr:getParameter name="user" /> in LDAP!

   </ldap:field></h2>
   <p>

 Your main email is:
   <blockquote>
   <ldap:field query="c" attribute="mail"/>
   </blockquote>

Your alternate email addresses are as follows:

<ul>
<ldap:loopValue id="foo" scope="request" query="c"
attribute="mailalternateaddress">
<li><attr:get name="foo" scope="request"/>
</ldap:loopValue>
</ul>
<cond:ldap name="c">
<cond:authenticated>
<p>

Your employee number is:
<ldap:field attribute="employeenumber" query="c">
They removed the employee numbers from ldap -- doesn’t that suck?!

</ldap:field>
</cond:authenticated>
<cond:else>
<cond:parameter name="password">
<cond:exists>Your specified password is incorrect.  Please
retry!</cond:exists>
<cond:else>To see your employee id, please specify a ’password’
parameter in the url along with your user name!<p></cond:else>

</cond:parameter>
</cond:else>
</cond:ldap>
<p>
<ldap:close resource="c"/>
</cond:exists>
<cond:else>
```

```
To see your employee information, please specify a ’user’ parameter
in the url!
<p>
</cond:else>
</cond:parameter>
</body></html>
Would produce one of the following (at least, it would if they hadn’t
removed employee number from nsdirectory recently!):
```


To see your employee information, please specify a 'user' parameter in the url!

Hello David Navas

Your main email is:

daven@netscape.com

Your alternate email addresses are as follows:

- daven@mcom.com
- david_navas@mcom.com
- david_navas@netscape.com

To see your employee id, please specify a 'password' parameter in the url along with your user name!

Hello David Navas

Your main email is:

daven@netscape.com

Your alternate email addresses are as follows:

- daven@mcom.com
- david_navas@mcom.com
- david_navas@netscape.com

Your employee number is: 033150

Conditional Tag Library

The conditional family of tags supports switch/case kinds of tags, allowing us to special case when a RowSet is at the end, or when a user should be given management-only types of information, or when a user has requested high-bandwidth content, etc. The root tags are as follows:

1. <cond:switch type="[value|role|rowset|ldap|attribute|parameter]" value="constant value, role name, rowset name, etc."> ... </cond:switch>
2. <cond:case operation="[=|<|>|<=|>=|!=|<>|><|=>|=<|~=|equals|equalsIgnoreCase |else|exists|notEmpty|executeNotEmpty|isLast|connected|authenticated|{method-name}]"
   value="..."></cond:case>
3. <cond:value value="blah">default value</cond:value>
   The value tag can exist in either the switch or case tags
4. <cond:dynamicValue value="blah"> ... <cond:value/> ... <cond:*case*/> ... </cond:dynamicValue>

However, for ease of use and better readability, the following equivalents exist:

1. <cond:role> ... </cond:role>
2. <cond:rowset name="rowset name"> ... </cond:rowset>
3. <cond:ldap name="ldap connection name"> ... </cond:ldap>
4. <cond:attribute name="attribute name"> ... </cond:attribute>
5. <cond:parameter name="parameter name | $REMOTE_USER$"> ... </cond:parameter>
6. <cond:else> ... </cond:else>
7. <cond:equals value="..."> ... </cond:equals>
8. <cond:equalsIgnoreCase value="..."> ... </cond:equalsIgnoreCase>
9. <cond:exists> ... </cond:exists>
10. <cond:notEmpty> ... </cond:notEmpty>
11. <cond:executeNotEmpty> ... </cond:executeNotEmpty>
12. <cond:isLast> ... </cond:isLast>
13. <cond:Connected> ... </cond:connected>
14. <cond:authenticated> ... </cond:authenticated>

Some of these may be more expressive than we really require. For example:

```
<cond:parameter name="foo"> ... </cond:parameter>
```

is really the same as:

```
<cond:switch><cond:value><%= request.getParameter("foo")
%></cond:value> ... </cond:switch>
```

Additionally, one might assume that this:

```
<cond:rowset value="rowset
name"><cond:exists>...</cond:exists></cond:rowset>
```

would be the same as:

```
<cond:rowset value="rowset name"><cond:case
operation="=">...</cond:case></cond:rowset>
```

It is currently unclear whether the increased expressiveness is worth the tradeoff in possible user-confusion.

Switch tag

The switch tag defaults to be a straight value comparison, much as a "real" switch tag is. However, it is more likely to be used in one of its other incarnations. In particular, the *rowset* type of switch replaces some of the callbacks that `DBRowSet` contained.

The switch tag keeps track of whether a particular case statement has fulfilled the switch statement. It exports nothing, except for its body to the content. page.

Case tag

The case tag contains an operation and (possibly) a second operand, which is used to determine if the case statement fulfills the Switch tag. Note that if a case and switch combination is used in which a value is required, and no value is specified, a value is obtained from an enclosing *cond:dynamicValue* tag. This allows the case tag to implement only the Tag interface, making the JSP build more efficient. The case tag's body will not be evaluated unless the case statement fulfills an as-yet unfulfilled switch statement.

If no operation is specified, the operation is assumed to be "else" -- that is, fulfill the switch regardless. We may want to change that, although whether we default to "=" or *equals* poses a very interesting question. If no operation is specified and the switch is of type *role*, the operation is assumed to be *equals*.

Note that certain case operations make sense in only certain types of switches. The *isLast* and *notEmpty* tags are useful for both ldap (query or entry) and rowset type of switches. The *executeNotEmpty* operation only make sense for *rowset* types of switches. The *connected* and *authenticated* operations only make sense for *ldap* types of switches. Similarly, the "=, <, >" etc. types of operations only make sense when comparing numerical values. The value of the switch and the value of the case are converted to Doubles (if necessary), and their values compared. The *equals* and *equalsIgnoreCase* operations make sense only against Strings, though equals is called against the switch's value's equals method -- which might be implemented by an Object to compares itself against a String (the value of the case, always!). *notEmpty* also makes sense for Strings as a check that a parameter has a specified, non-zero length setting.

Value tag

The value tag's body is evaluated and passed up the first parent of the value tag that implements `IValueContainingTag` which both `switch` and `dynamicValue` do (naturally). You can also specify the value in a value attribute of the value tag. But then, if you could do that, you'd have put the value in the switch or case directly, wouldn't you?

dynamicValue tag

The *dynamicValue* tag's body should have, at least, two elements in it. One is a *value* tag, which builds the dynamic value of interest. The second is one of the other case tags whose value attribute is extracted from the enclosing dynamicValue instance. The *dynamicValue* tag does have a *value* tag, but only an assembly programmer would truly love it:

```
<cond:attribute name="foo">
  <cond:dynamicValue value="10">
        <cond:case operation="<">less than ten</cond:case>
        <cond:case operation="=">equal to ten</cond:case>
        <cond:case operation=">">greater than ten</cond:case>
   </cond:dynamicValue>
</cond:attribute>
```

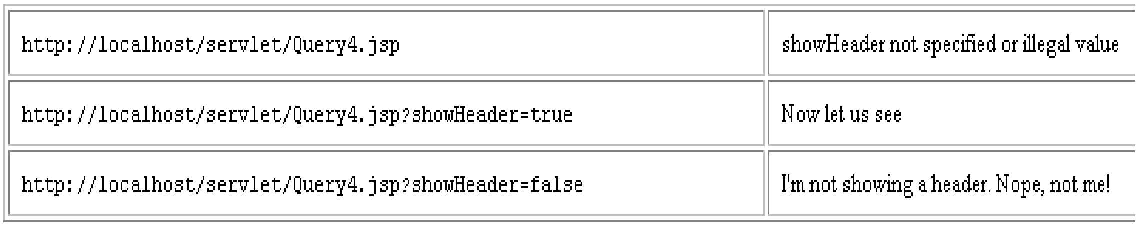
Alas, there is no machine-equivalent to comparison bits in the status register, so the operation is still performed three times. It is unlikely, though possible, that this tag will be used in this fashion.

Example

The following is an example of how switch might be used. The three links at the end will produce the three different types of output:

```
<%@ taglib prefix="cond" uri="condtags6_0.tld" %>

<cond:parameter name="showHeader">
  <cond:equalsIgnoreCase value="true">
    <h2>Now let us see</h2>
  </cond:equalsIgnoreCase>
  <cond:dynamicValue>
    <cond:value value="false"/>
    <cond:equalsIgnoreCase>
      I'm not showing a header.  Nope, not me!
    </cond:equalsIgnoreCase>
  </cond:dynamicValue>
  <cond:else>
    showHeader not specified or illegal value
   </cond:else>
</cond:parameter>
```

The possible outputs would be:



http://localhost/servlet/Query4.jsp	showHeader not specified or illegal value
http://localhost/servlet/Query4.jsp?showHeader=true	Now let us see
http://localhost/servlet/Query4.jsp?showHeader=false	I'm not showing a header. Nope, not me!

Attribute Tag Library

List of tags:

1. <attr:getAttribute **name**="attributeName" scope="[|page|request|session|application]" format="...">default value</attr:getAttribute>

2. <attr:setAttribute **name**="attributeName" value="..." scope="[page|request|session|application]">value</attr:setAttribute>

3. <attr:getParameter **name**="parameterName" format="urlEncode">default value</attr:getParameter>

4. <attr:getRemoteUser>default value</nas:getRemoteUser> <!-- prints out the Servlet's RemoteUser -->

getAttribute tag

Prints out the value of a named attribute whose value is retrieved from the specified *scope*. If no scope is specified, `findAttribute`() is used to find the attribute. If no value is found, the body of the tag is printed instead. The *format* is used ala the *query:field* tag.

setAttribute tag

Sets the value of a named attribute into the specified scope. If no scope is specified, page is assumed. The value is either the value specified in the value attribute, or if none is specifed, the body of the tag is used.

getParameter tag

Prints the value of a named parameter. If no parameter value exists, the body of the tag is printed instead. The format attribute is used ala the query:field tag.

Example

See examples on Conditional and LDAP taglibs

Load-balancing for JSPs

A significant addition to JSPs in iAS 6.0, compared to NAS 4.0, is the support for load-balancing JSPs individually. This is done by assigning a GUID to a JSP, similar to how GUIDs are assigned to servlets, i.e. in the XML descriptor. (See section on "Registered JSPs" on page 86). By assigning a GUID to a JSP, it becomes possible to load-balance JSPs just as servlets would, through the administrative tool.

The conditional family of tags supports switch/case kinds of tags, allowing us to special case when a RowSet is at the end, or when a user should be given management-only types of information, or when a user has requested high-bandwidth content, etc.

JSP Page Caching

iAS 6.0 provides a new feature called JSP Caching, which aids in development of compositional JSPs. This provides functionality to cache JSPs within the java engine, thereby making it possible to have a master JSP which includes multiple JSPs (a la a portal page), each of which can be cached using different cache criteria. Think of a portal page which contains a window to view stock quotes, another to view weather information, and so on. The stock quote window can be cached for 10 minutes, and the weather report window for 30 minutes, and so on.

Note that caching of JSPs is in addition to result caching, so its possible that a JSP can be composed of several included JSPs , each of which has a separate cache criterion, and the composed JSP itself can be cached in the KXS using the result-caching that becomes available as JSPs now have GUIDs (see section on “Registered JSPs” on page 86).

Caching of JSPs uses the custom tag library support provided by JSP 1.1. A typical cache-able JSP page looks as follows:

```
<%@ taglib prefix="ias" uri="CacheLib.tld"%>
<ias:cache>
<ias:criteria timeout="30">
<ias:check class="com.netscape.server.servlet.test.Checker"/>
<ias:param name="y" value="*" scope="request"/>
</ias:criteria>
</ias:cache>
<%! int i=0; %>
<html>
<body>
<h2>Hello there</h2>
I should be cached.
No? <b><%= i++ %></b>
</body>
</html>
```

The <ias:cache> and </ias:cache> delimit the cache constraints. The <ias:criteria > tag specifies the timeout value, and encloses different cache criteria. Cache criteria can be expressed using any or both of the tags, <ias:check> and <ias:param>. The syntax for these tags is as follows:

- <ias:criteria timeout="val" > : This specifies the timeout for the cached element, in seconds. The cache criteria are specified within this and the closing </ias:criteria>

- <ias:check class="classname" /> This is one of the mechanisms of specifying cache criteria. The classname refers to a class that has a method called "check", which has the following signature:

  ```
  public Boolean check(HttpServletRequest req)
  ```

  This returns a boolean indicating whether the element is to be cached or not.

- `<ias:param name="paramName" value="paramValue" scope="request" />` : This is the other mechanism to specify cache criteria.

`paramName` is the name of an attribute, passed in either in the request object (using setAttribute), or in the URI. This is the parameter used as a cache criterion.

`paramValue` is the value of the parameter, which determines whether caching should be performed or not. This can be of the following kinds:

constraint	**meaning**
x = ""	x must be present either as a parameter or as an attribute.
x = "v1 | ... | vk", where vi might be "*"	x is mapped to one of the strings (parameter / attribute). If x=*, then the constraint is true of the current request if the request parameter for x has the same value as was used to store the cached buffer.
x = "1-u", where 1 and u are integers.	x is mapped to a value in the range [1,u]

The scope identifies the source of the attributes that are checked. These can be page, request (default), session, or application.

Example:
The following is an example of a cached JSP page:

```
 <%@ taglib prefix="ias" uri="CacheLib.tld"%>
<ias:cache>
<ias:criteria timeout="30">
<ias:check class="com.netscape.server.servlet.test.Checker"/>
<ias:param name="y" value="*" scope="request"/>
</ias:criteria>
</ias:cache>
```

```
<%! int i=0; %>
<html>
<body>
<h2>Hello there</h2>

I should be cached.

No? <b><%= i++ %></b>
</body>
</html>
```

where Checker is defined as:

package com.netscape.server.servlet.test;

```
import com.netscape.server.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class Checker {
   String chk = "42";
   public Checker()
   {

   }


    public Boolean check(ServletRequest _req)

   {

       HttpServletRequest req = (HttpServletRequest)_req;
       String par = req.getParameter("x");
       return new Boolean(par == null ? false : par.equals(chk));
    }

}
```

Given the above, a cached element is valid for a request with parameter ’x=42’, and ’y’ equal to the value used to store the element. Note that its possible to have multiple sets of <ias:param> and <ias:check> inside an <ias:criteria> block. Also, its possible to have multiple <ias:criteria> blocks inside a JSP.

Now let us see

Chapter 4

Introducing Enterprise JavaBeans

This chapter describes how Enterprise JavaBeans (EJBs) work in the iAS application programming model.

This chapter begins by defining an EJB in terms of its roles and delivery mechanisms. Next it describes the two types of EJBs—entity beans and session beans—and when to use them. Finally, the chapter closes with an overview of designing an object-oriented iAS application using EJBs to encapsulate business logic.

This chapter includes the following sections:

- What Enterprise JavaBeans Do
- What Is an Enterprise JavaBean?
- Session Beans and Entity Beans
- The Role of EJBs in iAS Applications
- Designing an Object-Oriented Application

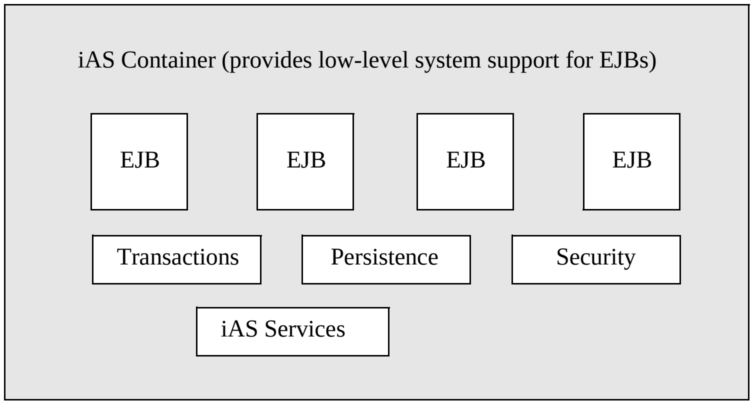
NOTE

If you already know about EJBs and how they are used in iAS, you may want to jump ahead to the chapter containing specific instructions and guidelines for developing EJBs for use with iAS: Chapter 6, "Building Business Entity EJBs" and Chapter 11, "Creating and Managing User Sessions".

What Enterprise JavaBeans Do

In iAS, Enterprise JavaBeans (EJBs) are the workhorses of your applications. If servlets act as the central dispatcher for your application and handle presentation logic, EJBs do the bulk of your application's actual data and rules processing, but provide no presentation or visible user-interface services. EJBs enable you to partition your business logic, rules, and objects into discrete, modular, and scalable units. Each EJB encapsulates one or more application tasks or application objects, including data structures and the methods that operate on them. Typically they also take parameters and send back return values.

EJBs always work within the context of a "container," which serves as a link between the EJBs and the server that hosts them. As an iAS developer, you need not worry about the container for your EJBs. The iAS software environment provides the container for your EJBs. This container provides all the standard container services denoted by the Sun EJB specification, and it provides additional services that are specific to iAS.



In fact, the container handles all actual remote access, security, concurrency, transaction control, and database access. Because the actual implementation details are part of the container and there is a standard, prescribed interface between a container and its EJBs, the bean developer is freed from having to know or handle platform-specific implementation details. Instead, the enterprise bean developer can create generic, task-focused EJBs that can be used with any vendor's products that support the EJB standard.

What Is an Enterprise JavaBean?

The Enterprise JavaBeans architecture is a component-based model for development and deployment of object-oriented, distributed, enterprise applications. An Enterprise JavaBean (EJB) is a single component in such an application. Applications written using EJBs are scalable, encapsulate transactions, and permit secure multiuser access. These applications can be written once and then deployed on any server platform that supports EJBs.

The fundamental characteristics of EJBs are as follows:

- Creation and management of beans is handled at runtime by a container. iAS itself provides the container for enterprise beans you write as part of your server application.
- Mediation of client access is handled by the container and the server where the bean is deployed, freeing the bean designer from having to process it.
- Restricting a bean to use standard container services defined by the EJB specification guarantees that the bean is portable and can be deployed in any EJB-compliant container.
- Including a bean in, or adding a bean to an application made up of other, separate bean elements—a “composite” application—does not require source code changes or recompiling of the bean.
- Definition of a client’s view of a bean is controlled entirely by the bean developer. The view is not affected by the container in which the bean runs or the server where the bean is deployed.

The EJB specification further states that an enterprise bean establishes three “contracts”: client, component, and jar file.

Understanding Client Contracts

The client contract determines the communication rules between a client and the EJB container, establishes a uniform development model for applications that use EJBs, and guarantees greater reuse of beans. In particular the client contract stipulates how an EJB object is identified, how its methods are invoked, and how it is created and destroyed.

The container for EJBs enable you to build distributed applications using your own components and components from other suppliers. iAS provides high-level transaction, state management, multithreading, and resource pooling wrappers, shielding you from having to know the details of many low-level APIs.

An EJB instance is created and managed at runtime by a container class, but the EJB itself can be customized at deployment time by editing its environmental properties. Metadata, such as transaction mode and security attributes, are separate from the bean itself, and are controlled by the container's tools at design and deployment. At run time, a client's access to a bean is controlled by the container and the server where the EJB is deployed.

The EJB container is also responsible for ensuring that a client can invoke the specialized business methods that the EJB defines. While a bean developer implements methods inside the bean, the developer must also provide a "remote interface" to the container that tells the container how clients can call the bean's methods.

Finally, the EJB supplies a "home interface" to the container. The home interface extends the `javax.ejb.EJBHome` interface defined in the EJB specification. This provides a mechanism for clients to create and destroy EJBs. At its most basic, the home interface defines zero or more `create(...)` methods for each way there is to create a bean. In addition, some types of EJBs, known as entity beans, must also define finder methods, one or more for each way there is to look up a bean or a collection of beans.

Understanding Component Contracts

The component contract establishes the relationship between an EJB and its container. As such, it is completely transparent to a client. There are several parts to the component contract for any given bean.

- Life cycle: For EJB session beans, this includes implementation of the `javax.ejb.SessionBean` and `javax.ejb.SessionSynchronization` interfaces. For EJBs entity beans, it includes instead implementation of the `javax.ejb.EntityBean` interface.
- Session context: A container implements the `javax.ejb.SessionContext` interface to pass services and information to a session bean instance when the bean instance is created.
- Entity context: A container implements the `javax.ejb.EntityContext` interface to pass services and information to an entity bean when the bean instance is created.
- Environment: A container implements `java.util.Properties` and makes it available to its EJBs.
- Services information: A container makes its services available to all its EJBs.

Finally, you can extend the component contract to provide additional services specific to your application needs.

Understanding Jar File Contracts

The standard format used to package enterprise Beans is the ejb-jar file. This format is the contract between the Bean Provider and Application Assembler, and between the Application Assembler and the Deployer. With iAS, you can create a jar file containing EJBs using the iAS Deployment Tool (iASDT). For more information, see the *Administration and Deployment Guide*.

The ejb-jar file must contain the deployment descriptor (DD) as well as all the class files for the following:

- The enterprise bean class.
- The enterprise bean home and remote interface.
- If the bean is an entity bean, the primary key class.

In addition, the ejb-jar file must contain the class files for all the classes and interfaces that the enterprise bean class, and the remote home interfaces depend on. For further details on the ejb-jar file contents see Chapter 10, “Packaging for Deployment“.

Session Beans and Entity Beans

There are two kinds of EJBs: entity and session. Each of these bean types is used differently in a server application. An EJB can be an object that represents a stateless service, an object that represents a session with a particular client (and which automatically maintains state across multiple client-invoked methods), or can be a persistent entity object possibly shared among multiple clients.

The following sections describe these two bean types in more detail.

Understanding Session Beans

Session EJBs typically have the following characteristics:

- They execute in relation to a single client.

- Optionally, they handle transaction management according to property settings.
- Optionally, they update shared data in an underlying database.
- They are relatively short-lived.
- They are not guaranteed to survive a server crash, unless you use iAS failover support for stateful session Beans.

A session bean implements business logic. All functionality for remote access, security, concurrency, and transactions is provided by the EJB container. A session EJB is a private resource used only by the client that creates it.

In iAS, an EJB that encapsulates business rules or logic is a session bean. The life duration of a session EJB is usually brief. For example, you might create an EJB to simulate an electronic shopping cart. Each time a user logs into your application, the application creates the session bean to hold purchases for that user. Once the user logs out or finishes shopping, the session bean is removed.

Understanding Entity Beans

Entity EJBs have the following characteristics:

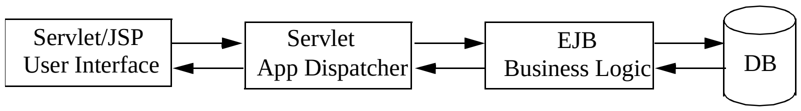
- Representation of data in the backend resource, usually a database.
- Bean-managed transaction demarcation.
- Container-managed transaction demarcation.
- Shared access for any number of users.
- Exists as long as its data is in a database.
- Transparently survives crashes of the EJB server.

The server that hosts EJBs and an EJB container, provides a scalable runtime environment for many concurrently active entity EJBs. Entity EJBs represent persistent data.

The Role of EJBs in iAS Applications

EJBs do the vast majority of business logic and data processing in an iAS application. They function invisibly behind the scenes to make an application work. Even though EJBs are at the heart of iAS applications, users are seldom aware of EJBs, nor do they ever interact directly with them.

When a user invokes an iAS application servlet from a browser, the servlet, in turn, invokes one or more EJBs to do the bulk of your application's business logic and data processing. For example, the servlet may load a Java Server Page (JSP) to the user's browser to request a user name and password, and the servlet also passes the user's input to a session bean to validate the input.



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

For example, suppose a servlet invokes an entity bean that gives a customer service representative access to a parts database. Access to the parts database might mean the ability to browse the database, to queue up items for purchase by the customer, to place the customer order (and permanently reduce the number of parts in the database), and to bill the customer. It might also include the ability to reorder parts when stock is low or depleted.

As part of the customer order process, a servlet creates a session bean that represents a "shopping cart" to keep temporary track of items as a customer orders them. When the order is complete, the data in the shopping cart is transferred to the order database, the quantity of each item in the inventory database is reduced, and the shopping cart session bean is freed.

As this simplified example illustrates, EJBs are invoked by a servlet to handle most of your application's business logic and data processing. Entity beans, are primarily used to handle data access using the Java Database Connectivity (JDBC) API. Session beans provide transient application objects and perform discrete business tasks.

The greatest challenge you face when creating an application that uses EJBs is determining how to break your application into servlets, JSPs, session Beans, and/or entity Beans.

Designing an Object-Oriented Application

Partitioning an iAS application's business logic and data processing into the most effective set of EJBs is a large part of your job as a developer. There are no hard and fast rules for object-oriented design with EJBs, other than that instances of entity beans tend to be long-lived, persistent, and shared among clients, while instances of session beans tend to be short-lived, and used only by a single client. Therefore, what follows in these sections is mostly high-level, iAS-specific advice for improving application speed, making your EJBs modular and shareable, and easing maintenance.

As is the case with all object-oriented development, you must determine what level of granularity you need for your business logic and data processing. Level of granularity refers to how many pieces you break your application into. A high level of granularity—where you divide your application into many, smaller, more narrowly defined EJBs—creates an application that may promote greater sharing and reuse of EJBs among different applications at your site. A low level of granularity creates a more monolithic application that usually executes more quickly.

NOTE

Decomposing an application into a moderate to large number of separate EJBs can create a huge degradation of application performance and more administrator overhead. EJBs, like JavaBeans, are not simply Java objects. They are components with remote call interface semantics, security semantics, transaction semantics, and properties. EJBs are higher level than Java objects.

Planning Guidelines

In general, create iAS applications that balance your need for execution speed with your need for sharing EJBs among applications and clients, and deploying applications across servers:

- Ask your administrator to co-locate EJBs with your presentation logic (servlets and JSPs) on the same server to reduce the number of remote procedure calls (RPCs) when you run your application.
- Create stateless session beans instead of stateful session beans as much as possible. If you must create stateful session beans, have your administrator turn on sticky load balancing for better performance.
- Create session EJBs that are small, generic, and narrowly task-focused. Ideally, these EJBs encapsulate behavior that can be used in many of your applications.

In addition to these general considerations, you must also decide which parts of your applications are candidates for becoming entity beans and which are candidates for becoming session beans.

Using Session Beans

Session beans are intended to represent transient objects and processes, such as updating a single database record, a copy of a document for editing, or specialized business objects for individual clients, such as a shopping cart. These objects are available only to a single client. Because of this, session Beans can maintain client-specific session information, called conversational state. Session Beans that maintain conversational state are called **stateful Session Beans**; Beans that do not are called **stateless Session Beans**.

When the client is done with the session objects, the objects are released. When you design an application, designate each such temporary, single-client object as a potential session bean. For example, in an online shopping application, each customer’s shopping cart is a temporary object. The cart lasts only as long as the customer is selecting items for purchase. Once the customer is done and the order is processed, the cart object is no longer needed and is released.

Like an entity bean, a session bean may access a database through JDBC calls. A session bean, too, can provide transaction settings. These transaction settings and JDBC calls are refereed by the session bean’s container, which is transparent to you as the developer. The container provided with iAS handles the JDBC calls and result sets.

For a complete discussion of using session beans to define temporary objects and rules for single-client access in an iAS application, see Chapter 5, “Using Session EJBs to Manage Business Rules“.

Using Entity Beans

Entity beans most commonly represent persistent data. This data is maintained directly in a database, or accessed through a backend application as an object. A simple example of an Entity Bean would be one that is defined to represent a single row in a database table, and where each instance of the Bean represents a specific row. A more complex example would be an Entity Bean designed to represent complicated views of joined tables in a database where each instance of the Bean would represent a single customer’s shopping cart contents.

Unlike Session Beans, Entity Bean instances can be accessed simultaneously by multiple clients. The container, is responsible for synchronizing the instance's state by using transactions. This delegation of responsibility to the container, means the Bean developer does not need to worry about concurrent access to methods from multiple transactions.

The persistence of an Entity Bean can either be managed by the bean itself, or by the bean's container. When the Entity Bean manages it's own persistence, it's called **Bean-managed persistence**. When the bean delegates this function to the container, it's called **container-managed persistence (CMP)**.

- **Bean-managed Persistence**. The bean developer must implement persistence code (such as JDBC calls) directly in the EJB class methods, if the bean is to manage it's own persistence. The possible downside of this implementation is the loss of portability, if a proprietary interface is used, and also the risk of tieing the bean to a specific database.
- **Container-managed Persistence (CMP)**. The container provider uses the iAS Deployment Tool (iASDT) to generate the bean code to implement the container persistence process. The container manages, transparently to the bean, the persistence state. The bean developer does not need to implement any data access code in the bean's methods. Not only is this method simpler for the bean developer to implement, but it makes the bean fully portable, without any ties to a specific database.

For a complete discussion of using entity beans to define persistent objects and business logic in an iAS application, see Chapter 6, "Building Business Entity EJBs".

Planning for Failover Recovery

Failover recovery is a process whereby a bean can reinstantiate itself after a server crash. Both stateless and stateful Session Beans support failover recovery. The iAS Deployment Tool is used to set the failover properties for session beans, for a description of how to make these settings refer to the *iASDT Administration and Deployment Guide*. For more information about session bean failover recovery see Chapter 5, "Using Session EJBs to Manage Business Rules".

Entity beans support failover recovery with the caveat that the reference to the bean is lost after a server crash. To recover an entity bean, you must create a new reference to it with a finder, see "Using Finder Methods" in Chapter 4, "Introducing Enterprise JavaBeans".

Working with Databases

In iAS, the preferred method for working with databases is through the Java Database Connectivity (JDBC) API in conjunction with transaction attributes.You use the Java Naming and Directory Interface (JNDI) to obtain a database connection. JNDI provides a standard way for the application to find and access database services independent of JDBC drivers.

For a complete discussion of using entity beans to define persistent objects and business logic in an iAS application, see Chapter 8, “Using JDBC for Database Access“.

For a complete description of transaction controls available through session and entity beans, see Chapter 7, “Handling Transactions with EJBs“.

Deploying EJBs

You deploy EJBs with the rest of an application using the iAS Deployment Tool (iASDT).

For more information on how to deploy EJBs using iASDT, see the *Administration and Deployment Guide*.

For information on the meaning of the property settings made by iASDT and how they effect your application, see Chapter 10, “Packaging for Deployment“.

Chapter 5

Using Session EJBs to Manage Business Rules

This chapter describes how to create session Enterprise JavaBeans (EJBs) that encapsulate your application's business rules and business objects.

Specifically, this chapter explains how to use session beans to encapsulate repetitive, time-bound, and user-dependent tasks that represent the transient needs of a single, specific user.

This chapter includes the following sections:

- Introducing Session EJBs
- Session Bean Components
- Additional Session Bean Guidelines

Introducing Session EJBs

Much of a standard, distributed application consists of logical units of code that perform repetitive, time-bound, and user-dependent tasks. These tasks can be simple or complex, and they are often needed in different applications. For example, banking applications must verify a user's account ID and balances before performing a fund transfer of any kind. These kinds of tasks define the business rules and business logic that you use to run your business. Such discrete tasks, transient by nature, are perfect candidates for session Enterprise JavaBeans (EJBs).

Session EJBs are self-contained units of code that represent client-specific instances of generic objects. These objects are transient in nature, created and freed throughout the life of the application on an as-needed basis. For example, the "shopping cart" employed by many web-based, on-line shopping applications is a typical session bean. It is created by the on-line shopping application only when you start choosing items to buy. When you finish selecting items, the price of the items in your cart is calculated, your order is placed, and the shopping cart object is freed. You can continue to browse merchandise in the on-line catalogue, and if you decide the place another order, a new shopping cart is created for you.

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

Session beans can either be “**stateless**” or “**stateful**.” A **stateless** session bean encapsulates a temporary piece of business logic needed by a specific client for a limited time span. A **stateful** session bean is transient, uses a “conversational state” to preserve information about its contents and values between client calls. The conversational state enables the container to maintain information about the state of the session bean and to recreate that state at a later point in program execution when needed.

The defining characteristics of a session bean have to do with its non-persistent, independent status within an application. One way to think of a session bean is as a temporary, logical extension of a client’s application that runs instead on the application server. A session bean:

- executes for a single client.
- updates data in an underlying database.
- is short-lived.

Generally, a session bean does not represent shared data in a database, but obtains a snapshot of data. The bean can, however, update data. Optionally, a session bean can also be transaction-aware. Its operations can take place in the context of a transaction managed by the bean.

A client accesses a session bean through the bean’s remote interface, *`EJBObject`*. An EJB object is a remote Java programming language object accessible from the client through standard Java APIs for remote object calls. The EJB lives in the container from its creation to its destruction, and the container manages the EJB’s life cycle and support services. Where an EJB resides or executes is transparent to the client that accesses it. Finally, multiple EJBs can be installed in a single container. The container provides services that allow clients to look up the interfaces of installed EJB classes through the Java Naming and Directory Interface (JNDI).

A client never accesses instances of a session bean directly. Instead, a client uses the session bean's remote interface to access a bean instance. The EJB object class that implements a session bean's remote interface is provided by the container. At a minimum, an EJB object supports all of the methods of the `java.ejb.EJBObject` interface. This includes methods to obtain the home interface for the session bean, to get the object's handle, to test if the object is identical to another object, and to remove the object. These methods are stipulated by the EJB Specification. (All specifications are accessible from `installdir/iAS/docs/index.htm`, where `installdir` is the location in which you installed iAS.)

In addition, most EJB objects also support specific business logic methods. These are the methods at the heart of your applications.

Session Bean Components

When you code a session bean you must provide the following class files:

- Enterprise Bean remote interface, extending `javax.ejb.EJBObject`
- Enterprise Bean class definition
- Enterprise Bean home interface, extending `javax.ejb.EJBHome`
- Enterprise Bean meta-data (deployment descriptors and other configuration information)

Creating the Remote Interface

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

```
import javax.ejb.*;
import java.rmi.*;
public interface MySessionBean extends EJBObject {
// define business method methods here....
}
```

The remote interface defines the session bean's business methods that a client calls. The business methods defined in the remote interface are implemented by the bean's container at run time. For each method you define in the remote interface, you must supply a corresponding method in the bean class itself. The corresponding method in the bean class must have the same signature.

Besides the business methods you define in the remote interface, the `EJBObject` interface defines several abstract methods that enable you to retrieve the home interface for the bean, to retrieve the bean's handle, which is its unique identifier, to compare the bean to another bean to see if it is identical, and to free, or remove the bean when it is no longer needed.

For more information about these built-in methods and how they are to be used, see the EJB Specification. All specifications are accessible from `installdir/iAS/docs/index.htm`, where *`installdir`* is the location in which you installed iAS.

Declaring vs. Implementing the Remote Interface

A bean class definition must include one matching method definition, including matching method names, arguments, and return types, for each method defined in the bean's remote interface. The EJB specification also permits the bean class to implement the remote interface directly, but recommends against this practice to avoid inadvertently passing a direct reference (via `this`) to a client in violation of the client-container-EJB protocol intended by the specification.

Creating the Class Definition

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

```
import java.rmi.*;
import java.util.*;
import javax.ejb.*;
public class MySessionBean implements SessionBean {

// Session Bean implementation. These methods must always included.

public void ejbActivate() throws RemoteException {
}

public void ejbPassivate() throws RemoteException {
}

public void ejbRemove() throws RemoteException{
}

public void setSessionContext(SessionContext ctx) throws
RemoteException {
}

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

The session bean must also implement one or more `ejbCreate(...)` methods. There should be one such method for each way a client is allowed to invoke the bean. For example:

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

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

All useful session beans also implement one or more business methods. These methods are usually unique to each bean and represent its particular functionality. For example, if you define a session bean to manage user logins, it might include a unique function called `ValidateLogin()`.

Business method names can be anything you want, but must not conflict with the names of methods used in the EJB architecture. Business methods must be declared as `public`. Method arguments and return value types must be legal for Java RMI. The `throws` clause may define application-specific exceptions, and must include `java.rmi.RemoteException`.

There is one interface implementation permitted in a session bean class definition, particularly `javax.ejb.SessionSynchronization`, that enables a session bean instance to be notified of transaction boundaries and synchronize its state with those transactions. For more information about this interface, see the EJB Specification. All specifications are accessible from *`installdir`*`/iAS/docs/index.htm`, where *`installdir`* is the location in which you installed iAS.

Session Timeout

The container removes inactive session beans after they are inactive for a specified (or default) time. This timeout value is set in the bean's XML Deployment Descriptor (DD) file. For more information, see "EJB XML DTD" in "Packaging for Deployment".

Passivation and Activation

The container passivates session beans after they are inactive for a specified (or default) time. This timeout value is set in the bean's deployment descriptor. For more information, see "EJB XML DTD" in "Packaging for Deployment".

For more information about passivation, see the EJB specification. All specifications are accessible from *`installdir`*`/ias/docs/index.htm`, where *`installdir`* is the location in which you installed iAS.

Creating the Home Interface

The home interface defines the methods that enable a client using your application to create and remove session objects. A home interface always extends `javax.ejb.EJBHome`. For example:

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

public interface MySessionBeanHome extends EJBHome {
   MySessionBean create() throws CreateException, RemoteException;
}
```

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

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

All home interfaces automatically define two remove methods for destroying an EJB when it is no longer needed. Do not override these methods.

Additional Session Bean Guidelines

Before you decide what parts of your application you can represent as session beans, you should know a few more things about session beans. A couple of these things are related to the EJB specification for session beans, and a couple are specific to iAS and its support for session beans.

Creating Stateless or Stateful Beans

The EJB specification describes two state management modes for session beans:

- **STATELESS**: the bean retains no state information between method calls, so any bean instance can service any client.
- **STATEFUL**: the bean retains state information across methods and transactions, so a specific bean instance must be associated with a single client at all times.

If you decide to use stateful session beans, plan to co-locate stateful beans with their clients. Also, use sticky load balancing to reduce the number of remote procedure calls, especially for session beans that are passivated and activated frequently or for session beans that use many resources, such as database connections and handles.

Accessing iAS Functionality

You can develop session beans that adhere strictly to the EJB Specification, you can develop session beans that take advantage both of the specification and additional, value-added iAS features, and you can develop session beans that adhere to the specification in non-iAS environments, but that take advantage of iAS features if they are available. Make the choice that is best for your intended deployment scenario.

iAS offers several features through the iAS container and iAS APIs that enable your applications to take programmatic advantage of specific features of the iAS environment. You can embed API calls in your session beans if you plan on using those beans only in the iAS environment.

For example, you can trigger a named application event from an EJB using the `IAppEventMgr` interface, using the following steps and example:

1. First obtain an instance of `com.kivasoft.IContext` by casting `javax.ejb.SessionContext` or `javax.ejb.EntityContext` to `IServerContext`.
2. Next, use the `GetAppEventMgr()` method in the `GXContext` class to create an `IAppEventMgr` object.
3. Finally, trigger the application event with `triggerEvent()`.

```
javax.ejb.SessionContext m_ctx;
....
com.netscape.server.IServerContext sc;
sc = (com.netscape.server.IServerContext) m_ctx;
com.kivasoft.IContext kivaContext = sc.getContext();
IAppEventMgr mgr = com.kivasoft.dlm.GXContext.GetAppEventMgr(ic);
mgr.triggerEvent("eventName");
```

Serializing Handles and References

The EJB Specification indicates that to guarantee serializable bean references, you should use handles rather than direct references to EJBs.

iAS direct references are also serializable. You may wish to take advantage of this extension, but you should be aware that not all vendors support it.

Managing Transactions

Many session beans interact with databases. You can control transactions in beans using settings in the bean's property file. This permits you to specify transaction attributes at bean deployment time. By having a bean handle transaction management you are freed from having explicitly to start, roll back, or commit transactions in the bean's database access methods.

By moving transaction management to the bean level, you gain the ability to place all of a bean's activities—even those not directly tied to database access—under the same transaction control as your database calls. This guarantees that all parts of your application controlled by a session bean run as part of the same transaction, and either everything the bean undertakes is committed, or is rolled back in the case of failure. In effect, bean-managed transactional state permits you to synchronize your application without having to code any synchronization routines.

Committing a Transaction

When a session bean signals that it is time to commit a transaction, the actual commit process is handled by the bean's container. Besides affecting the data your application processes, commit time also affects the state of a session bean. The iAS container implements commit option C as described in the EJB specification.

When a commit occurs, it signals to the container that the session bean has completed its useful work, and should synchronize its state with the underlying data store. The container permits the transaction to complete, and then frees the bean. Result sets associated with a committed transaction are no longer valid. Subsequent requests for the same bean cause the container to issue a load to synchronize state with the underlying data store.

Note that transactions begun in the container are implicitly committed. Also, any participant can roll back a transaction. For more information about transactions, see Chapter 7, " Handling Transactions with EJBs".

Accessing Databases

Many session beans access and even update data. Because of the transient nature of session beans, however, be careful about how that access takes place. In general, use the JDBC API to make your calls, and always use the transaction and security management methods described in Chapter 7, " Handling Transactions with EJBs" to manage transaction isolation level and transaction requirements at the bean level.

For more information see Chapter 8 , "Using JDBC for Database Access."

Session Bean Failover

The session bean failover feature allows conversational state recovery for stateful session beans in the case of an iAS server becoming unavailable due to a system crash or power failure. Supporting failover for stateful session beans is an iAS value-added feature. J2EE programs do not need any modification to support this iAS failover feature. Failover is handled by the container and is defined by the deployer in the deployment descriptor.

Imagine a corporate buyer performing on-line purchasing at an e-commerce web site. After spending hours shopping, the buyer has hundreds of items in their shopping cart (a stateful session bean). The system then has an unexpected fatal problem, and the instance of the iAS server becomes unavailable. Without failover capability, the failure would result in the buyer's shopping cart becoming empty; the state of the stateful session bean would be lost. With the failover feature in place the buyer is unaware of the system failure; the failover mechanism redirects the client to a running instance of the iAS server that has the state of the bean before the failure. The buyer's shopping cart will then have the same selected items as before the failover took place.

Notable features of the failover support for stateful session beans include:

- Failover is a value-added feature that supports J2EE programs.
- Failover is transparent to the client; no special APIs are required for the client to take advantage of this feature.
- Failover is handled by the container and configured by the deployer.
- Distributed Store (DSync) is the enabling mechanism for restoring state after a system failure.
- Performance impact is minimal for stateful session beans that do not need failover support.

How to Configure a Stateful Bean with Failover

Configuring a stateful session bean for failover capability is a combination of configuring the bean with failover and Distributed Store (DSync).

- During installation or during runtime, configure server for DSync.
- During deployment, configure the stateful session bean with failover capability.

In order to fully take advantage of the failover feature, the bean must have configured both failover and DSync. The DSync mechanism saves the conversational state of the session bean during runtime. The failover mechanism, allows the container to detect a system failure and then connect to another running iAS instance with the saved session bean state.

Refer to the *Administration Guide* for details on how to configure a stateful session bean with failover during deployment and how to configure DSync during runtime. Refer to the *Installation Guide* for details on configuring DSync during installation.

How the Failover Process Works

Failover of stateful beans is achieved with a combination of smart stubs and a distributed store. When a bean is deployed as a failover bean, the deployment tool generates special stubs. On a methd invocation, these smart stubs can detect failure and transparently relocate a bean to a new home, potentially in a different engine. Once the bean is relocated, the stub retries the method on the recovered bean. The container guarantees at-most-once semantics when trying a method.

The container uses a distributed store that is based on DSync to maintain the state of a bean. The state of the bean is saved at regular intervals, and automatically reinstated as part of the recovery process.

How often is the State Saved

An container with failover configured, saves the bean states during runtime at regular intervals. The process for saving state includes:

- Saving at regular, configurable time intervals.
- Also saved on transaction boundaries, if the bean participates in transactions.

The regular time interval is configured by the iAS Deployment Tool.

How the State is Saved

The process for saving the state is as follows:

- First, each stateful session bean's `ejbPassivate()` method is called.
- The bean's conversational state is then serialized and saved to the distributed store.

- And finally, the bean's `ejbActivate()` method is called.

NOTE

Saving the state for a bean is quite expensive because of the operations involved.

How the Failover Process Works

Session beans with failover configured, have smart stubs that detect session bean failures and recreate new bean references when a failure occurs. The stubs determine that the Bean's reference has become stale by getting a connect exception from the dead bean. The stub then does a look up of the home and gets the remote interface.

See , "Packaging for Deployment," for a description of the deployment descriptors used by stateful session beans for failover.

Failover Guidelines

Keep in mind the following suggestions and guidelines when implementing failover:

- Keep `ejbPassivate`() and `ejbActivate`() simple.
- Use `obj.remove()` to remove a bean, not `home.remove(handle)`. Association between a bean and it's original home may be preserved after failover.
- Use judgement by carefully weighing the advantages of bean failover against the performance cost of the failover process. Do not make every stateful bean require failover.
- Remember, session bean state is conversational. Use entity beans for transactional data.

Chapter 6

Building Business Entity EJBs

This chapter explains what a business entity EJB is and what entity beans must contain. It also provides additional guidelines for creating entity beans, and for determining the entity beans needs of your applications.

This chapter contains the following sections:

- Introducing Business Entity EJBs
- Entity Bean Components
- Additional Entity Bean Guidelines

All specifications are accessible from *`installdir`*`/ias/docs/index.htm`, where *`installdir`* is the location in which you installed iAS.

Introducing Business Entity EJBs

Often, the heart of a distributed, multi-user application involves interactions with data sources, such as a database, or even with an existing legacy application. Such interactions are often transactional. In most of these cases, the external data source or business object is transparent to the user, or is shielded or buffered from direct user interaction. These protected, transactional, and persistent interactions with databases, document files, and other business objects, are candidates for encapsulation in entity EJBs.

Business EJBs are self-contained, reusable components—with data members, properties, and methods—that represent instances of generic, transactionally aware, persistent data objects that can be shared among clients. *Persistence* refers to the creation and maintenance of a bean throughout the lifetime of the application.

There are two types of persistence management and iAS supports both types as listed below.

- *container-managed persistence* - This is when the container is responsible for the persistence of the beans.
- *bean-managed persistence* - This is when the beans are responsible for their own persistence.

As a developer, you code a bean-managed entity bean by providing database access calls—via JDBC or SQL—directly in the methods of the bean class. Database access calls must be placed in the code for the `ejbCreate(...)`, `ejbRemove()`, `ejbFindXXX()`, `ejbLoad()`, and `ejbStore()` methods. The advantage of using bean-managed persistence is that such beans can be installed into any container without requiring the container to generate database calls.

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

As an iAS application developer, you cannot access the container's entity bean services directly, nor do you ever need to. Instead, know that the container is there to take care of low-level implementation details so that you can focus on the larger role the entity bean plays in your application picture.

Clients access an entity bean through the bean's remote interface. The object that implements the remote interface is called the EJB object. Usually, an entity EJB is shared among multiple clients, and represents a single point of entry to a data resource or business object, such as a database. Regardless of which client access an entity object at a given time, each client's view of the object is both location independent, and transparent to other clients.

Finally, any number of entity beans can be installed in a container. The container implements a home interface for each entity bean. The home interface enables a client to create, look up, and remove entity objects. A client can look up an entity bean's home interface through the Java Naming and Directory Interface (JNDI).

An entity bean:

- represents data in a database.
- supports transactions.
- executes for multiple clients.
- persists for as long as needed by all clients.
- transparently survives server crashes.

Generally, an entity bean represents shared data in a database and is transaction aware. Its operations always take place in the context of transactions managed by the bean's container.

How an Entity Bean Is Accessed

A client, such as a browser or servlet, accesses an entity bean through the bean's remote interface, *EJBObject*. An EJB object is a remote Java programming language object accessible from the client through standard Java APIs for remote object calls. The EJB lives in the container from its creation to its destruction, and the container manages the EJB's life cycle and support services.

A client never accesses instances of an entity bean directly. Instead, a client uses the entity bean's remote interface to access a bean instance. The EJB object class that implements an entity bean's remote interface is provided by the container. At a minimum, an EJB object supports all of the methods of the `java.ejb.EJBObject` interface. This includes methods to obtain the entity bean's home interface, to get the object's handle, to retrieve the entity's primary key, to test if the object is identical to another object, and to remove the object. These methods are stipulated by the EJB Specification. All specifications are accessible from *`installdir`*`/ias/docs/index.htm`, where *`installdir`* is the location in which you installed iAS.

In addition, the remote interface for most EJB objects also support specific business logic methods. These are the methods at the heart of your specific applications.

Entity Bean Components

When you code an entity bean you must provide the following class files:

- Enterprise Bean class
- Enterprise Bean home interface, `javax.ejb.EJBHome`
- Enterprise Bean remote interface, `javax.ejb.EJBObject`

Creating the Class Definition

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

```
import java.rmi.*;
import java.util.*;
import javax.ejb.*;
public class MyEntityBean implements EntityBean {

// Entity Bean implementation. These methods must always included.

public void ejbActivate() throws RemoteException {
}

public void ejbLoad() throws Remote Exception {
}

public void ejbPassivate() throws RemoteException {
}

public void ejbRemove() throws RemoteException{
}

public void ejbStore() throws RemoteException{
}

public void setEntityContext(EntityContext ctx) throws
RemoteException {
}

public void unsetEntityContext() throuws RemoteException {
}

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

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

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

There are really two types of business methods you can implement in an entity bean: internal ones, that are used by other business methods in the bean, but are never accessed outside the bean itself; and external ones, that are referenced by the entity bean's remote interface.

The following sections describe the various methods in an entity bean's class definition in more detail.

The examples in this section assume the following member variable definitions:

```
private transient javax.ejb.EntityContext m_ctx = null;

// These define the state of our bean
private int m_quantity;
private int m_totalSold;
```

Using ejbActivate and ejbPassivate

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

If there are specific application tasks that need to be performed when a bean is first made ready for an application, or that need to be performed when a bean is no longer needed by the application, code those operations within these methods.

Activation is not the same as creating a bean. You can only activate a bean that has already been created. Similarly, passivation is not the same as removing a bean. Passivation merely returns a bean instance to the container pool for later use. `ejbRemove()` is required to actually terminate a bean instance's existence.

The container passivates session beans after they are inactive for a specified (or default) time. This timeout value is set in the bean's property file. For more information, see"EJB XML DTD" in Chapter 11, "Packaging for Deployment".

For more information about `ejbActivate()` and `ejbPassivate()`, see the EJB Specification. All specifications are accessible from *`installdir`*`/ias/docs/index.htm`, where *`installdir`* is the location in which you installed iAS.

For more information about `ejbCreate()` and `ejbRemove()`, see "Using ejbCreate Methods" on page 144.

Using ejbLoad and ejbStore

An entity bean should permit its container to store the bean's state information in a database for synchronization purposes. Use your implementation of `ejbStore()` to store state information in the database, and use your implementation of `ejbLoad()` to retrieve state information from the database. When the container calls `ejbLoad()`, it synchronizes the bean state by loading state information from the database.

The following example shows `ejbLoad()` and `ejbStore()` definitions that methods store and retrieve active data.

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

    itemId = (String) m_ctx.getPrimaryKey();

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

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

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

    if (rs != null) {
        rs.next();
        m_totalSold = rs.getInt(1);
        m_quantity = rs.getInt(2);
    }
}
```

```
public void ejbStore()
            throws java.rmi.RemoteException
{
    String itemId;
    itemId = (String) m_ctx.getPrimaryKey();
    DatabaseConnection dc = null;
    java.sql.Statement stmt1 = null;
    java.sql.Statement stmt2 = null;

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

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

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

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

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

NOTE

For related information about isolation levels in beans that access transactions concurrently with other beans, see “Handling Concurrent Access” on page 150.

Using setEntityContext and unsetEntityContext

A container calls `setEntityContext()` after it creates an instance of an entity bean in order to provide the bean with an interface to the container itself. When you implement this method, use it to store the reference to the container in an instance variable.

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

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

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

Using ejbCreate Methods

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

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

Each `ejbCreate(...)` method must be declared as `public`, return either the entity's primary key type or a collection, and be named `ejbCreate`. The return type can be any legal Java RMI type that can be converted to a number for key purposes. Any arguments must be legal types for Java RMI. The `throws` clause, may define application-specific exceptions, may include `java.rmi.RemoteException`, or `java.ejb.CreateException`.

For each `ejbCreate()` method, the entity bean class may optionally define an `ejbPostCreate()` method to handle entity services immediately following creation Each `ejbPostCreate()` method must be declared as `public`, must return void, and be named `ejbPostCreate`. The method arguments, if any, must match in number and type the arguments of its corresponding `ejbCreate` method. The `throws` clause, may define application-specific exceptions, may include `java.rmi.RemoteException`, or may include `java.ejb.CreateException`.

Finally, an entity bean also implements one or more `ejbRemove()` methods to free a bean when it is no longer needed.

Using Finder Methods

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

The following example shows a definition for `FindByPrimaryKey()`:

```
public String ejbFindByPrimaryKey(String key)
              throws java.rmi.RemoteException,
                     javax.ejb.FinderException
{
    //System.out.println("@@@ myBean.ejbFindByPrimaryKey key = " +
key);
    return key;
}
```

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

Finder methods must be declared as public, and their arguments and return values must be legal types for Java RMI. The return type of each finder method must be the entity bean's primary key type or a collection of objects of the same primary key type. If the return type is a collection, the return type must be `java.util.Enumeration`.

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

Declaring vs. Implementing the Remote Interface

A bean class definition must include one matching method definition, including matching method names, arguments, and return types, for each method defined in the bean's remote interface. The EJB specification also permits the bean class to implement the remote interface's methods, but recommends against this practice to avoid inadvertently passing a direct reference (via `this`) to a client in violation of the client-container-EJB protocol intended by the specification.

Creating the Home Interface

The home interface defines the methods that enable a client accessing your application to create and remove entity objects. A home interface always extends `javax.ejb.EJBHome`. For example:

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

public interface MyEntityBeanHome extends EJBHome {
   MyEntityBean create() throws CreateException, RemoteException;
}
```

As this example illustrates, an entity bean's home interface defines one or more `create` methods. Usually the home interface also defines one or more `find` methods corresponding to the `finder` methods in the bean class.

Defining Create Methods

Each such method must be named `create`, and must correspond in number and type of arguments to an `ejbCreate` method defined in the entity bean class. The return type for each create method, however, does not match the return type of its corresponding `ejbCreate` method. Instead, it must return the entity bean's remote interface type.

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

Defining Find Methods

A home interface can define one or more `find` methods. Each such method must be named `findXXX` (e.g., `findApplesAndOranges`), where `XXX` is a unique continuation of the method name. Each finder method must correspond to one of the finder methods defined in the entity bean class definition. The number and type of arguments must also correspond to the finder method definitions in the bean class. The return type, however, may be different. The return type for a finder method in the home interface must be the entity bean's remote interface type or a collection of interfaces.

Finally, all home interfaces automatically define two remove methods for destroying an EJB when it is no longer needed. Do not override these methods.

Creating the Remote Interface

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

```
import javax.ejb.*;
import java.rmi.*;
public interface MyEntityBean extends EJBObject {
// define business method methods here....
}
```

The remote interface defines the entity bean's business methods that a client calls. The business methods defined in the remote interface are implemented by the bean's container at runtime. For each method you define in the remote interface, you must supply a corresponding method in the bean class itself. The corresponding method in the bean class must have the same signature.

Besides the business methods you define in the remote interface, the `EJBObject` interface defines several abstract methods that enable you to retrieve the home interface for the bean, to retrieve the bean's handle, to retrieve the bean's primary key which uniquely identifies the bean's instance, to compare the bean to another bean to see if it is identical, and to remove the bean when it is no longer needed.

For more information about these built-in methods and how they are to be used, see the EJB Specification. All specifications are accessible from *`installdir`*`/ias/docs/index.htm`, where *`installdir`* is the location in which you installed iAS.

Additional Entity Bean Guidelines

Before you decide what parts of your application you can represent as entity beans, you should know a few more things about entity beans. A couple of these things are related to the EJB specification for entity beans, and a couple are specific to iAS and its support for entity beans.

Accessing iAS Functionality

You can develop entity beans that adhere strictly to the EJB Specification, you can develop entity beans that take advantage both of the specification and additional, value-added iAS features, and you can develop entity beans that adhere to the specification in non-iAS environments, but that take advantage of iAS features if they are available. Make the choice that is best for your intended deployment scenario.

iAS offers several features through the iAS container and iAS APIs that enable your applications to take programmatic advantage of specific features of the iAS environment. You can embed API calls in your entity beans if you plan on using those beans only in the iAS environment.

Serializing Handles and References

The EJB Specification indicates that to guarantee serializable bean references, you should use handles rather than direct references to EJBs.

iAS direct references are also serializable. You may wish to take advantage of this extension, but you should be aware that not all vendors support it.

Managing Transactions

Most entity beans interact with databases. You can control transactions in beans using settings in the bean's property file. This permits you to specify transaction attributes at bean deployment time. By having a bean handle transaction management you are freed from having explicitly to start, roll back, or commit transactions in the bean's database access methods.

By moving transaction management to the bean level, you gain the ability to place all of a bean's activities—even those not directly tied to database access—under the same transaction control as your database calls. This guarantees that all parts of your application controlled by a entity bean run as part of the same transaction, and either everything the bean undertakes is committed, or is rolled back in the case of failure. In effect, bean-managed transactional state permits you to synchronize your application without having to code any synchronization routines.

Committing a Transaction

When a commit occurs, it signals to the container that the entity bean has completed its useful work, and should synchronize its state with the underlying data store. The container permits the transaction to complete, and then returns the bean to the pool for later reuse. Result sets associated with a committed transaction are no longer valid. Subsequent requests for the same bean cause the container to issue a load to synchronize state with the underlying data store.

Note that transactions begun in the container are implicitly committed. Also, any participant can roll back a transaction. For more information about transactions, see Chapter 7, “Handling Transactions with EJBs".

Commit Options B and C

Commit options B and C are supported by iAS CMP.

- *Commit option B* caches the entity bean instance between method invocations. The instance retains it's association with a particular primary key and is not yet returned to the pool of free, unassociated instances. When the instance's stay in the ready cache exceeds the ready cache timeout value, the container will transition the instance back to the free pool.
- *Commit option C* gets an instance of a bean from the free pool at the start of a transaction and transitions the instance back to the free pool at the end of the transaction.

NOTE

If the entity bean has the TX_NOT_SUPPORTED attribute set and the commit option B defined, the bean will perform read only data caching.

The rules of operation for commit option B are a little more complex than for commit option C. Here are some examples to explain the differences.

Example 1

This is the lifecyle for several business method invocations by a single transactional client.

On the first invocation (no ready instance has been cached):

```
ejbActivate -> ejbLoad -> business method-> ejbStore-> (instance is
put in ready cache)
```

On subsequent invocations:

```
(retrieve ready instance)-> ejbLoad -> business method -> ejbStore->
(instance is returned to ready cache)
```

In comparison, the lifecycle for every business method invocation under commit option C looks like this:

```
ejbActivate-> ejbLoad -> business method -> ejbStore -> ejbPassivate
```

If there is more than one transactional client concurrently accessing the same entity EJBObject, the first client will get the ready instance and subsequent concurrent clients will get new instances from the pool. The lifecycle for these new instances will be the same as if they were operating under commit option C.

Example 2

This is the lifecyle for several business method invocations by a single non-transactional client.

On the first invocation (no ready instance has been cached):

```
ejbActivate -> ejbLoad -> (instance is put in ready cache)->
business method
```

On subsequent invocations:

```
(get reference to ready instance) -> business method
```

Multiple non-transactional clients can access the ready instance concurrently. The bean, therefore, must be declared as reentrant.

If a transactional client is using the ready instance, non-transactional clients will block until the transaction is completed.

Handling Concurrent Access

As an entity bean developer, you do not have to be concerned about concurrent access to an entity bean from multiple transactions. The bean's container automatically provides synchronization in these cases. In iAS, the iAS container activates one instance of an entity bean for each simultaneously-occurring transaction that uses the bean. Transaction synchronization is performed automatically by the underlying database during database access calls.

The iAS EJB container implementation does not provide its own synchronization mechanism when multiple transactions try to access an entity bean. It creates a new instance of the entity bean for every new transaction. The iAS container delegates the responsibility of synchronization to the application.

You typically perform this synchronization in conjunction with the underlying database or resource. One possible approach would be to acquire the corresponding database locks in the `ejbLoad()` method, for example by choosing an appropriate isolation level or by using a "select for update" clause. The specifics depend on the database backend that is being used. For more information, see the EJB specification as it relates to concurrent access.

The following example `ejbLoad()` snippet illustrates the use of "select for update" syntax to obtain database locks. This prevents other instances from being loaded at the same time.

```
public void ejbLoad() throws java.rmi.RemoteException
{
....
// Get the lock on the corresponding DB table
    try {
            java.sql.Connection dbConn = ds.getConnection();
            String query = "SELECT accountNum, balance FROM accounts "
                         + "WHERE customerId = ? FOR UPDATE";
            prepStmt = dbConn.prepareStatement(query);
            prepStmt.setString(1, m_customerId);
            resultSet = prepStmt.executeQuery();
            if ((resultSet != null) && resultSet.next()) {
                acctNum = resultSet.getInt(1);
                acctBalance = resultSet.getInt(2);
            } else {
                throw new RemoteException("Database error. "
                                        + "Couldn’t find accout");
            }
        } catch (java.sql.SQLException e)  {
            throw new RemoteException("Database error. "
                                    + "Couldn’t load account");
        } finally {
            try {
                if (resultSet != null)
                    resultSet.close();
                if (prepStmt != null)
                    prepStmt.close();
                if (dbConn != null)
                    dbConn.disconnect();
            } catch (java.sql.SQLException e) {
                System.out.println("Unexpected exception while "
                                 + "closing resources"); }
        }
    }
```

Accessing Databases

Most entity beans work with databases. Always use the transaction and security management methods in `javax.ejb.deployment.ControlDescriptor` to manage transaction requirements. For more information about creating and managing transactions with beans, see Chapter 7, “Handling Transactions with EJBs".

To work with data in the context of a bean-managed transaction, use JDBC. For more information about using JDBC to work with data, see Chapter 8, “Using JDBC for Database Access".

Container Managed Persistence

If you want the container to manage the storage of entity bean state to an underlying resource manager then you can use container managed persistence (CMP) entity beans.

Important iAS CMP goals include:

- Support for the J2EE v 1.2 specification CMP model.
- Support of multiple pluggable persistence managers in the server.
- Provide user with a deployment tool (iASDT) to perform the object relational (O/R) mapping and create the separate CMP deployment descriptor XML files.
- Support for sophisticated custom finder methods

How Container Managed Persistence Works

The EJB container needs two things to support CMP entity beans. First, it needs information on how to map an entity bean into a resource such as a table in a relational database. Second, it needs a CMP runtime environment that will use the mapping information to perform persistence operations on each bean.

iAS supports multiple CMP solutions via an architecture known as pluggable CMP (PCMP). A CMP solution consists of a mapping tool and a runtime environment. The mapping tool is used to generate mapping information for CMP entity beans. Typically this information is stored in an XML file. The CMP runtime environment is responsible for performing persistence operations.

iAS ships with a mapping tool that is embedded in iASDT. Please see the deployment tool documentation for information on how to use iASDT to generate mapping information. iAS also ships with a reference implementation of a CMP runtime environment that utilizes the mapping information generated by the iASDT to perform persistence operations (see below for more information).

Pluggable Container Managed Persistence

iAS supports CMP entity beans via the pluggable container managed persistence architecture (PCMP). The primary benefit of this architecture is that it separates the EJB container from the CMP runtime environment (also known as the persistence manager). The EJB container communicates with the persistence manager through a well defined open API. Using this API, the container is able to ask the persistence manager to perform various persistence operations throughout the entity bean lifecycle. The persistence manager is responsible for performing the persistence operations and may use caching and other techniques to improve performance.

A number of vendors, including Thought, Inc. and Forte, are planning to adapt their CMP products to support PCMP. Please look at the product release notes for a URL link with a list of products that currently support PCMP.

PCMP is an open and well defined API. It is possible to write your own persistence manager to support legacy systems. Please look at the product release notes for a URL link with information on how to obtain the PCMP API specification.

How to Use Persistence Managers in iAS

Pluggable persistence managers allow users to set the persistence policies for a bean at deployment time. In order to use PCMP with a particular bean, you need to set a persistence manager factory class in the xml deployment descriptor. The EJB container will use this factory class to create persistence managers during runtime.

The iASDT should be used to set the persistence manager factory class for a particular bean. Please refer to the iASDT documentation for information on how to modify the deployment descriptor.

If you are using the iAS persistence manager reference implementation the factory class tag should have the value:
`com.netscape.server.ejb.SQLPersistenceManagerFactory`

iAS Persistence Manager Reference Implementation

iAS ships with a simple persistence manager reference implementation. The reference implementation uses mapping information generated by the iASDT. Users may also hand modify the mapping information file produced by iASDT to access additional features. For example, custom finder methods are not supported by iASDT but may be added to the mapping information file by hand.

Please go to the following url to obtain the latest technical documentation on the iAS persistence manager reference implementation. Please look at the product release notes for a URL link with information on how to obtain the latest technical documentation on the iAS persistence manager reference implementation.

Chapter 7

Handling Transactions with EJBs

This chapter describes the transaction support built into the Enterprise JavaBean programming model.

This chapter begins by introducing the EJB transaction model and explains the notion of container and bean managed transactions. Then it explains the semantics of all transaction attributes and use of the Java Transaction API for bean managed transactions. Finally, it discusses the restrictions on various combinations of EJB types and transaction attributes.

This chapter includes the following sections:

- Understanding the Transaction Model
- Specifying Transaction Attributes in an EJB
- Using Bean Managed Transactions

Understanding the Transaction Model

One of the primary advantages of using Enterprise JavaBeans (EJBs) is the support they provide for declarative transactions. In the declarative transaction model, attributes are associated with beans at deployment time and it is the container's responsibility, based on the attribute value, to demarcate and transparently propagate transactional context. The container is also responsible, in conjunction with a Transaction Manager, for ensuring that all participants in the transaction see a consistent outcome.

Declarative transactions free the programmer from explicitly demarcating transactions. They facilitate component-based applications where multiple components, potentially distributed and updating heterogeneous resources, can participate in a single transaction. The EJB specification also supports programmer-demarcated transactions using `javax.transactions.UserTransaction()`.

It is necessary to understand the distinction between global and local transactions in order to understand the iAS support for transactions. Global transactions are managed and coordinated by a Transaction Manager and can span multiple databases and processes. The Transaction Manager typically uses the XA protocol to interact with the database backends. Local transactions, on the other hand, are native to a database and are restricted within a single process. Local transactions can work against only a single backend. Local transactions are typically demarcated using JDBC APIs.

In iAS, all transactions are started by the container or by using `javax.transaction.UserTransaction()`.

The EJB specification requires support for flat (as opposed to nested) transactions. In this model each transaction is decoupled from and independent of other transactions in the system. Flat transactions are by far the most prevalent model and are supported by most commercial database systems.

NOTE

If your application uses global transactions, your should configure and enable the corresponding iAS Resource Managers. See the *Administration and Deployment Guide* for further details.

Specifying Transaction Attributes in an EJB

Transactional attributes can be specified on a bean-wide basis or on a per-method basis for a bean's remote interface. If attributes are specified at both levels, method-specific values take precedence over bean-wide values. These two should be mixed with care since some combinations are invalid as documented in the restrictions section.

Transactional attributes are specified as part of the bean's XML Deployment Descriptor file (see "EJB iAS XML DTD" in Chapter 10, "Packaging for Deployment").

Using Bean Managed Transactions

While it is preferable to use container-managed transactions, you may find that your application requirements necessitate the use of bean-managed transactions. To manage transactions programmatically, refer to the Enterprise JavaBeans Specification, v1.1 for this interface at the following URL:

`http://java.sun.com/products/ejb/javadoc-1.1/javax/ejb/EJBContext.html`

Chapter 8

Using JDBC for Database Access

This chapter describes how to use the Java Database Connectivity (JDBC) API for database access with iPlanet Application Server.

This chapter provides high-level instructions for using the iAS implementation of JDBC in servlets and EJBs, and it indicates specific iAS resources affected by JDBC statements when those resources have clear programming ramifications. In iAS, Enterprise JavaBeans (EJBs) support database access primarily through the JDBC API. iAS supports all of JDBC 2.0 API as well as many of the emerging JDBC 2.0 extensions to JDBC, including result set enhancements, batch updates, distributed transactions, row sets, and JNDI support for datasource name lookups.

While this chapter assumes familiarity with JDBC 2.0, it also describes specific implementation issues that may have programming ramifications. For example, the JDBC specifications do not make it clear what constitute JDBC resources. In the specifications, some JDBC statements—such as any of the `Connection` class methods that close database connections—release resources without specifying exactly what those resources are.

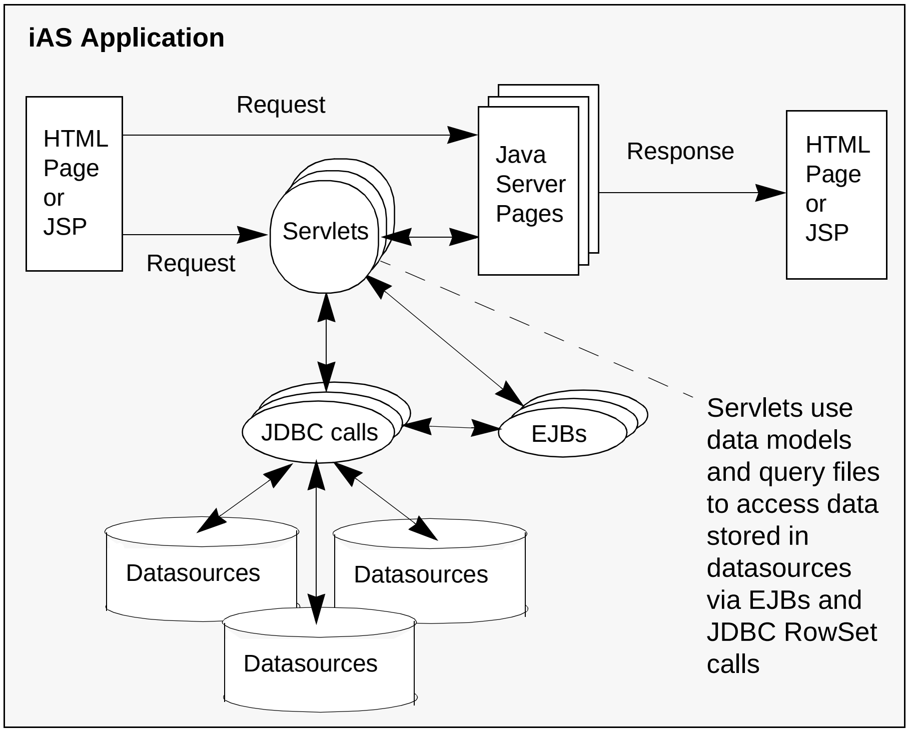
This chapter contains the following sections:

- Introducing JDBC
- Using JDBC in Server Applications
- Handling Connections
- Working with JDBC Features

Introducing JDBC

From a programming perspective, JDBC is a set of Java classes and methods that let you embed database calls in your server applications. That's all you need to know in order to start using JDBC in your server applications.

More specifically, JDBC is a set of interfaces that every server vendor, such as iPlanet, must implement according to the JDBC specifications. iAS provides a JDBC type 2 driver which supports a variety of database backends. This driver processes the JDBC statements in your applications and routes the SQL arguments they contain to your database engines.



JDBC lets you write high-level, easy-to-use code that can operate seamlessly with and across many different databases without your needing to know most of the low-level database implementation details.

Supported Functionality

The JDBC specification is a broad, database-vendor independent set of guidelines that try to encompass the broadest range of database functionality possible in a simple framework.

At a minimum, JDBC assumes that all underlying database vendors support the SQL-2 database access language. Since its original inception, the JDBC specification has grown. It now has three parts:

- JDBC 2.0 describes the core set of database access and functionality that server vendors must implement in order to be JDBC compliant. The iPlanet Application Server (iAS) fully meets this compliance standard. From the database vendor’s perspective, JDBC 2.0 describes a database access model that permits full access to the standard SQL-2 language, those portions of the standard language each vendor supports, and those extensions to the language that each vendor implements.
- JDBC 2.0 describes additional database access and functionality. Much of this functionality involves support for newly-defined SQL-3 features, data types, and mappings. Other parts of JDBC 2.0 extend JDBC 2.0 features. The iAS implementation of JDBC supports most of the JDBC feature enhancements, but omits support for the new SQL-3 data types, such as blobs, clobs, and arrays, which many database vendors do not, as yet, fully support in their relational database management systems. The iAS JDBC implementation also omits support for SQL-3 data type mapping
- JDBC 2.0 Standard Extension API describes advanced support features, many of which offer the promise of improved database performance. The iAS implementation of JDBC currently supports JNDI and rowsets.

Understanding Database Limitations

When you start using JDBC in your server applications, you may encounter situations when you do not get the results you desire or expect. You may think the problem lies in JDBC or in the iAS implementation of the JDBC driver. In fact, the vast majority of the time, the problems actually lie in the limitations of your database engine.

Because JDBC covers the broadest possible spectrum of database support, it enables you to attempt operations not every database supports. For example, most database vendors support most of the SQL-2 language, but no vendor provides fully unqualified support for all of the SQL-2 standard. Most vendors built SQL-2 support on top of their already existing proprietary relational database management systems, and either those proprietary systems offer features not in SQL-2, or SQL-2 offers features not available in those systems. Most vendors, too, have added non-standard SQL-2 extensions to their implementations of SQL in order to support their proprietary features. JDBC provides ways for you to access vendor-specific features, but it's important to realize that these features may not be available for all the databases you use. This is especially true when you build an application that uses databases from two or more different vendors.

As a result, not all vendors fully support all aspects of every available JDBC class, method, and method arguments. More importantly, a set of SQL statements embedded as an argument in a JDBC method call may or may not be supported by the database or databases your server application uses. In order to make maximum use of JDBC, you must consult your database vendors' documentation about which aspects of SQL and JDBC they support. Before you call iAS technical support for a database problem, make sure you first eliminate the possibility that your database is the cause of the problem.

Understanding iAS Limitations

Like JDBC, iAS attempts to support the broadest possible spectrum of database engines and features. In some cases, iAS itself or the iAS JDBC driver may not fully support a particular feature of a database you use, or they may report incorrect information. If you can't access a database feature from your iAS application and you have eliminated the database as the source of the problem, check this section of the documentation and the product release notes to see if the problem you encounter is a documented iAS limitation. If not, document the problem fully and contact iAS technical support.

NOTE

Some JDBC access problems can result if you attempt to access JDBC features that are either partially supported or not supported by the iAS JDBC driver. Almost all of these feature limitations apply to JDBC 2.0.

The following table lists JDBC features that are either partially or completely unsupported in iAS 6.0:

Table 8-1 JDBC Feature Limitations

Feature	Limitation
Connection.setTransactionIsolation	Works only with isolation levels supported by your database vendors.
Connection.getTypeMap	Type maps are not supported.
Connection.setTypeMap	Type maps are not supported.
Connection.cancel	Works only with databases that support it.
PreparedStatement.setObject	Works only with simple data types.
PreparedStatement.addBatch	Works only with supported data manipulation statements that return a count of records changed.
PreparedStatement.setRef	References are not supported.
PreparedStatement.setBlob	Blobs are not supported. Use `setBinaryStream()` instead.
PreparedStatement.setClob	Clobs are not supported. Use `setBinaryStream()` instead.
PreparedStatement.setArray	Arrays are not supported. Use `setBinaryStream()` instead.
PreparedStatement.getMetaData	Not supported.
CallableStatement.getObject	Works only with scalar types. JDBC 2.0 offers a second version of this method that includes a map argument. The map argument is ignored.
CallableStatement.getRef	References are not supported.
CallableStatement.getBlob	SQL3-style blobs are not supported.
CallableStatement.getClob	SQL3-style clobs are not supported.
CallableStatement.getArray	Arrays are not supported.
CallableStatement	Updatable ResultSet is not supported.
ResultSet.getCursorName	Behavior differs depending on database:
	For Oracle, if user does not specify a cursor name with SetCursorName, an empty string is returned.
	For Sybase, if the result set is not updatable, a cursor name is automatically generated by iAS. Otherwise an empty string is returned.
	For ODBC, Informix, and DB2, the driver returns a cursor name if none is specified.

Table 8-1 JDBC Feature Limitations

Feature	Limitation
ResultSet.getObject	Works only with scalar types. JDBC 2.0 offers two other versions of this method that includes a map argument. The map argument is ignored.
ResultSet.updateObject	Works only with scalar types.
ResultSet.getRef	References are not supported.
ResultSet.getBlob	SQL3-style blobs are not supported.
ResultSet.getClob	SQL-style clobs are not supported.
ResultSet.getArray	Arrays are not supported.
ResultSetMetaData.getTableName	Returns an empty string for non-ODBC database access.
DatabaseMetaData.getUDTs	Not supported.

For more information about working with `ResultSet`, `ResultSetMetaData`, and `PreparedStatement`, see the appropriate sections later in this chapter.

Supported Databases

iAS currently connects to many different relational databases. The following tables lists the most commonly used databases that are supported.

Table 8-2 Supported Databases

Database	Notes
Oracle	Support is offered through the Oracle OCI interface. Both Oracle 7 and Oracle 8 database instances are supported in a fully multi-threaded environment. iAS also coexists with all Oracle RDBMS tools and utilities, such as SQL*Plus, Server Manager, and Oracle Backup.
Informix	Support is offered through Informix CLI interface. Both Informix Online Dynamic Server and Informix Universal Server are supported.
Sybase	Support is offered through Sybase CTLIB.
Microsoft SQLServer	Support is offered through the Microsoft ODBC interface. Microsoft SQLServer on Windows NT only is supported.

Table 8-2 Supported Databases

Database	Notes
DB2	Support is offered through the DB2 CLI client interface. DB2 versions 5.2 and 6.1 are supported.
ODBC	iAS does not specifically support or certify any ODBC 2.0 or 3.0 compliant driver set, though they may work.

Both because the databases supported by iAS are constantly updated, and because database vendors consistently upgrade their products, you should always check with iAS technical support for the latest database support information.

Using JDBC in Server Applications

JDBC is part of the iAS run-time environment. In theory, this means that JDBC is always available to you any time you use Java to program an application. In a typical multi-tiered server application, it is theoretically possible to use JDBC to access a database backend from the client, from the presentation layer, in servlets, and in Enterprise Java Beans (EJBs).

In practice, however, it usually makes sense—for security and portability reasons—to restrict database access to the middle layers of a multi-tiered server application. In the iAS programming model, this means placing all JDBC calls in servlets and EJBs, with a strong preference toward EJBs.

There are two reasons for this programming preference. One is that placing all JDBC calls inside EJBs makes your application more modular and more portable. Another is that EJBs provide built-in mechanisms for transaction control. Placing JDBC calls in well-designed EJBs frees you from programming explicit transaction control using JDBC or `java.transaction.UserTransaction` that provide low-level transaction support under JDBC.

NOTE

Make sure to use a globally available datasource to create a global (bean-wide) connection so that the EJB transaction manager can control the transaction.

Using JDBC in EJBs

Placing all your JDBC calls in EJBs ensures a high degree of server application portability. It also frees you from having to manage transaction control with explicit JDBC calls unless you so desire. Because EJBs are components, you can use them as building blocks for many applications with little or no recoding, and maintain a common interface to your database backends.

Managing Transactions with JDBC or javax.transaction.UserTransaction

Using the EJB transaction attribute property to manage transactions is recommended, but not mandatory. There may be times when explicit coding of transaction management using JDBC or `javax.transaction.UserTransaction` is appropriate for your application. In these cases, you code transaction management in the bean yourself. Using an explicit transaction in an EJB is called a bean-managed transactions.

Transactions can be local to a specific method (method-specific) or they can encompass the entire bean (bean-wide).

There are two steps for creating a bean managed transaction:

1. Set the transaction attribute property of the EJB to `TX_BEAN_MANAGED` in the bean's deployment descriptor.
2. Code the appropriate JDBC or transaction management statements in the bean, including statements to start the transaction, and to commit it or roll it back.

It is an error to code explicit transaction handling in EJBs for which the transaction attribute property is not `TX_BEAN_MANAGED`. For more information about handling transactions with JDBC, see the JDBC 2.0 API specification.

Specifying Transaction Isolation Level

You can specify or examine the transaction level for a connection using the methods `setTransactionIsolation()` and `getTransactionIsolation()`, respectively. Note that you can not call `setTransactionIsolation()` during a transaction.

Transaction isolation levels are defined as follows:

Table 8-3 Transaction Isolation Levels

Transaction Isolation Level	**Description**
TRANSACTION_NONE	Transactions are not supported. Only used with `Connection.getTransactionIsolation()`
TRANSACTION_READ_COMMITTED	Dirty reads are prevented; non-repeatable reads and phantom reads can occur.
TRANSACTION_READ_UNCOMMITTED	Dirty reads, non-repeatable reads and phantom reads can occur.
TRANSACTION_REPEATABLE_READ	Dirty reads and non-repeatable reads are prevented; phantom reads can occur.
TRANSACTION_SERIALIZABLE	Dirty reads, non-repeatable reads and phantom reads are prevented.

Before you specify a transaction isolation level for a bean, make sure the level is supported by your relational database management system. Not all databases support all isolation levels. You can test your database programmatically by using the method `supportsTransactionIsolationLevel()` in `java.sql.DatabaseMetaData`, as in the following example:

```
java.sql.DatabaseMetaData db;
if (db.supportsTransactionIsolationLevel(TRANSACTION_SERIALIZABLE) {
    Connection.setTransactionIsolation(TRANSACTION_SERIALIZABLE);
}
```

For more information about these isolation levels and what they mean, see the Java Database Connectivity (JDBC) 2.0 API Specification.

Using JDBC in Servlets

Servlets are at the heart of your iAS server applications. They stand between a client interface, such as an HTML page on a browser, the JSP that generated the HTML, and the EJBs that do the bulk of your application's work.

iAS applications use JDBC embedded in EJBs for most database access. This is the preferred arrangement for database access using iAS because it enables you to take advantage of the transaction control built into EJBs and their containers. Servlets, however, can also provide database access through JDBC.

In some situations, accessing a database directly from a servlet can offer a speed advantage over accessing databases from EJBs. There is less call overhead, especially if your application is spread across servers so that your EJBs are accessible only through the Java Remote Method Interface (RMI). Use direct database service through servlets sparingly. If you do provide database access from servlets, restrict access to those situations where access is very short duration, the transaction is read-only, and you can take advantage of the new JDBC 2.0 rowset class.

If you choose to access a database from a servlet, use the new JDBC 2.0 rowset interface to interact with a database. A rowset is a Java object that encapsulates a set of rows that have been retrieved from a database or other tabular datasource, such as a spreadsheet. The rowset interface provides JavaBean properties that allow a rowset instance to be configured to connect to a datasource and retrieve a set of rows. For more information about working with rowsets, see "Working with Rowsets" on page 180.

Handling Connections

iAS implements the JDBC 2.0 compliant interface java.sql.Connection. The behavior of the connection depends on whether it is a local, global or container managed local connection.

Local Connections

A `Connection` object is called a local connection if its transaction context is not managed by an EJB container. The transaction context in a local connection can not propagate across processes or across datasources; it is local to the current process and to the current datasource.

The transaction context on this type of connection is managed using the methods `setAutoCommit()`, `commit()`, and `rollback()`.

Registering a Local Datasource

The first step in creating a local connection is to register the datasource with iAS. Once the datasource is registered, the datasource can be used to make connections to the listed database using `getConnection()`.

You can register the datasource by creating an XML resource descriptor file that describes the properties of the datasource. Next, register the properties with iAS using the Administration Tool or the provided utility `resReg`. `resReg` takes as its argument, the name of the resource descriptor file describing the datasource.

NOTE

When run, resReg overwrites existing entries.

For example, to register a datasource called SampleDS which connects to an Oracle database using the username kdemo, password kdemo, database ksample and server ksample, create a XML descriptor file like the following, and name it `SampleDS.xml` (use the iAS Deployment Tool to create this XML file):

```
<ias-resource>
   <resource>
      <jndi-name>jdbc/SampleDS</jndi-name>
      <jdbc>
         <database>ksample</database>
         <datasource>ksample</datasource>
         <username>kdemo</username>
         <password>kdemo</password>
         <driver-type>ORACLE_OCI</driver-type>
      </jdbc>
   </resource>
</ias-resource>
```

You can then use this resource descriptor file to register the datasource with the following command:

```
resReg GlobalSampleDS.xml
```

For more information about resource descirptor files, see Chapter 10, “Packaging for Deployment". For more information about the iAS Administration Tool, see the *Administration and Deployment Guide*.

Global Connections

A `Connection` object is called a global connection if its transaction context is managed by EJB container. The transaction context in a global connection can be propagated across datasources. The transaction context is managed implicitly by the EJB container for container-managed transactions, or explicitly in case of bean-managed transactions. For more information about transactions, see Chapter 7, “Handling Transactions with EJBs".

Transaction management methods, e.g. `setAutoCommit()`, `commit()`, and `rollback()`, are disabled for global connections.

Using Resouce Managers

The collection of datasources in which a global transaction participates is known as a resource manager. All resources managers needs to be registered with iAS and be enabled so that they participate in global transactions. Resource managers can be set up at install time or they can also be set up using the iAS Administration Tool (see the *Administration and Deployment Guide*). A global connection must be associated with a resource manager.

Registering a Global Datasource

The first step in creating a local connection is to register the datasource with iAS. Once the datasource is registered, the datasource can be used to make connections to the listed database using `getConnection()`.

You can register the datasource by creating an XML resource descriptor file that describes the properties of the datasource. Next, register the properties with iAS using the Administration Tool or the provided utility `resReg`. `resReg` takes as its argument, the name of the resource descriptor file describing the datasource.

NOTE

When run, resReg overwrites existing entries.

For example, to register a datasource called GlobalSampleDS which connects to an Oracle database using the username kdemo, password kdemo, database ksample and server ksample, create a XML descriptor file like the following, and name it `GlobalSampleDS.xml` (use the iAS Deployment Tool to create this XML file):

```
<ias-resource>
   <resource>
      <jndi-name>jdbc/GlobalSampleDS</jndi-name>
      <jdbc>
         <database>ksample</database>
         <datasource>ksample</datasource>
         <username>kdemo</username>
         <password>kdemo</password>
         <driver-type>ORACLE_OCI</driver-type>
         <resource-mgr>ksample_rm</resource-mgr>
      </jdbc>
   </resource>
</ias-resource>
```

You can then use this resource descriptor file to register the datasource with the following command:

```
resReg GlobalSampleDS.xml
```

For more information about resource descirptor files, see Chapter 10, “Packaging for Deployment". For more information about the iAS Administration Tool, see the *Administration and Deployment Guide*.

Creating a Global Connection

The following code demonstrates how a datasource is looked up and a connection created from it. As is illustrated by the code, the string that is looked up is the same as specified in the <jndi-name> tag in the resource descriptor file.

```
InitialContext ctx = null;
String dsName1 = "jdbc/GlobalSampleDS";
DataSource ds1 = null;

try
{
    ctx = new InitialContext();
    ds1 = (DataSource)ctx.lookup(dsName1);

    UserTransaction tx = ejbContext.getUserTransaction();

    tx.begin();

    Connection conn1 = ds1.getConnection();

    // use conn1 to do some database work -- note that 
conn1.commit(),
    // conn1.rollback() and conn1.setAutoCommit() can not used here

    tx.commit();

} catch(Exception e) {
    e.printStackTrace(System.out);
}
```

Container Managed Local Connections

A `Connection` object is considered a container managed local connection when its' transaction context is managed by the EJB container and global transactions are disabled. In the case of container managed transactions, the transaction context is managed implicitly by the EJB container and in the case of bean managed transactions the transaction context is handled explicitly.

Connection object methods `setAutoCommit()`, `commit()`, and `rollback()` are disabled for this type of connection.

For more information on how to enable or disable global transactions in an EJB container, please refer to the *iAS Administration and Deployment Guide.*

Registering a Container Managed Local Datasource

The registering process for container managed local datasources is the same as for the local and global datasources, see “Registering a Local Datasource,” on page 168.

Working with JDBC Features

While this chapter is not a JDBC primer, it does introduce you to using JDBC in EJBs with iAS 6.0. The following sections describe various JDBC interfaces and classes that have either have special requirements in the iAS environment, or that are new JDBC 2.0 features that you are especially encouraged to use when developing iAS server applications.

For example, “Working With Connections” describes what resources iAS releases when a connection is closed because that information differs among different JDBC implementations. On the other hand, “Pooling Connections” and “Working with Rowsets” offer more extensive coverage because these are new JDBC 2.0 features that offer increased power, flexibility, and speed for your server applications.

Working With Connections

When you open a connection in JDBC, iAS allocates resources for the connection. If you call `Connection.close()` when a connection is no longer needed, the connection resources are freed. Always reestablish connections before continuing database operations after you call `Connection.close()`.

You can use `Connection.isClosed()` to test whether the connection is closed. This method returns false if the connection is open, and returns true only after `Connection.close()` is called.

You can determine if a database connection is invalid by catching the exception that is thrown when a JDBC operation is attempted on a closed connection.

Finally, opening and closing connections is an expensive operation. If your application uses several connections, and if connections are frequently opened and closed, iAS automatically provides connection pooling. Connection pooling provides a cache of connections that are automatically closed when necessary.

NOTE

Connection pooling is an automatic feature of iAS; the API is not exposed.

setTransactionIsolation

Not all database vendors support all levels of transaction isolation available in JDBC. iAS permits you to specify any isolation level your database supports, but throws an exception against values your database does not support. For more information, see “Specifying Transaction Isolation Level” on page 166.

getTypeMap, setTypeMap

The iAS implementation of the JDBC driver does not support type mapping, a new SQL-3 feature that most database vendors also do not support. The methods exist, but currently do nothing.

cancel

`cancel()` is supported for all databases.

Pooling Connections

Two of the costlier database operations you can execute in JDBC are for creating and destroying database connections. Connection pooling permits a single connection cache to be used for connection requests. When you use connection pooling, a connection is returned to the pool for later reuse without actually destroying it. A later call to create a connection merely retrieves an available connection from the pool instead of actually creating a new one.

iAS automatically provides JDBC connection pooling wherever you make JDBC calls. The process of pooling database connections works differently for each type of connection.

- For **local connections**, the database connections are pooled when they are closed by the application.
- For **global connections**, the database connections are tied to the thread that initiated the transaction. These connections are later resused by transactions that get executed on that thread.
- For **container managed local connections**, the `connection.close()` method does not release the connection to the connection pool immediately. Once the transaction that the connection is participating in is finished, the connection is released back to the connection pool by the application server.

In each java engine, each driver (Oracle, Sybase, Informix and DB2) has its own connection pools. Each connection pool can be sized according to the application requirements. See the Administration and Deployment Guide for more information on the connection pool settings (such as maximum number of connections, connection timeout and so on).

Working with ResultSet

`ResultSet` is a class that encapsulates the data returned by a database query. Be aware of the following behaviors or limitations associated with this class.

Concurrency Support

iAS supports concurrency for `FORWARD-ONLY READ-ONLY` result sets. On callable statements, iAS also supports concurrency for `FORWARD-ONLY UPDATABLE` result sets.

iAS also supports concurrency for `SCROLL-INSENSITIVE READ-ONLY` result sets.

`SCROLL-SENSITIVE` concurrency is not supported.

Updatable Result Set Support

In iAS, creation of updatable result sets is restricted to queries on a single table. The `SELECT` query for an updatable result set must include the `FOR UPDATE` clause:

```
SELECT...FOR UPDATE [OF column_name_list]
```

NOTE

You can use join clauses to create read-only result sets against multiple tables; these result sets are not updateable.

For Sybase, the select list must include a unique index column. Sybase also permits you to call `execute()` or `executeQuery()` to create an updatable result set, but the statement must be then be closed before you can execute any other SQL statements.

To use an updatable result set with Oracle 8, you must wrap the result set query in a transaction:

```
conn.setAutoCommit(false);
ResultSet rs =
   stmt.executeQuery("SELECT...FOR UPDATE...");

...

rs.updateRows();

...

conn.commit();
```

For Microsoft SQL Server, if concurrency for a result set is `CONCUR_UPDATABLE`, the `SELECT` statement in the `execute()` or `executeQuery()` methods must not include the `ORDER BY` clause.

getCursorName

One method of `ResultSet, getCursorName()`, enables you to determine the name of the cursor used to fetch a result set. If a cursor name is not specified by the query itself, different database vendors return different information. iAS attempts to handle these differences as transparently as possible. The following table indicates the name of a cursor returned by different database vendors if no cursor name is specified in the initial query.

Table 8-4

Database Vendor	getCursorName Value Returned
Oracle	If a cursor name is not specified with setCursorName(), an empty string is returned.
Sybase	If a cursor name is not specified with setCursorName(), and the result set is not updatable, a unique cursor name is automatically generated by iAS. Otherwise an empty string is returned.
Informix, DB2, ODBC	If a cursor name is not specified with setCursorName(), the driver automatically generates a unique cursor name.

getObject

iAS implements this JDBC method in a manner that only works with scalar data types. JDBC 2.0 adds additional versions of this method that include a map argument. iAS does not implement maps, and ignores the map argument if it is supplied.

getRef, getBlob, getClob, and getArray

References, blobs, clobs, and arrays are new SQL-3 data types. iAS does not implement these data objects or the methods that work with them. You can, however, work with references, blobs, clobs, and arrays using `getBinaryStream()` and `setBinaryStream()`.

Working with ResultSetMetaData

The `getTableName()` method only returns meaningful information for OBDC-compliant databases. For all other databases, this method returns an empty string.

Working with PreparedStatement

`PreparedStatement` is a class that encapsulates a query, update, or insert statement that is used repeatedly to fetch data. Be aware of the following behaviors or limitations associated with this class.

NOTE

You can use the iAS feature `SqlUtil.loadQuery()` to load a `iASRowSet` with a prepared statement. For more information, see the entry for the **SqlUtil** class in the *iAS Foundation Class Reference.*

setObject

This method may only be used with scalar data types.

addBatch

This method enables you to gang a set of data manipulation statements together to pass to the database as if it were a single statement. `addBatch()` only works with SQL data manipulation statements that return a count of the number of rows updated or inserted. Contrary to the claims of the JDBC 2.0 specification, `addBatch()` does not work with any SQL data definition statements such as `CREATE TABLE`.

setRef, setBlob, setClob, setArray

References, blobs, clobs, and arrays are new SQL-3 data types. iAS does not implement these data objects or the methods that work with them. You can, however, work with references, blobs, clobs, and arrays using `getBinaryStream()` and `setBinaryStream()`.

getMetaData

Not all database systems return complete metadata information. See your database vendor's documentation to determine what kind of metadata your database provides to clients.

Working with CallableStatement

`CallableStatement` is a class that encapsulates a database procedure or function call for databases that support returning result sets from stored procedures. Be aware of the following limitation associated with this class. The JDBC 2.0 specfication states that callable statements can return an updatable result set. This feature is not supported in iAS.

getRef, getBlob, getClob, getArray

References, blobs, clobs, and arrays are new SQL-3 data types. iAS does not implement these data objects or the methods that work with them. You can, however, work with references, blobs, clobs, and arrays using `getBinaryStream()` `and setBinaryStream()`.

Handling Batch Updates

The JDBC 2.0 Specification provides for a batch update feature that allows for an application to pass multiple SQL update statements (`INSERT, UPDATE, DELETE`) in a single request to a database. This ganging of statements can result in a significant increase in performance when a large number of update statements are pending.

The `Statement` class includes two new methods for executing batch updates:

- `addBatch()` permits you to add an SQL update statement (`INSERT`, `UPDATE`, `DELETE`) to a group of such statements prior to execution. Only update statements that return a simple update count can be grouped using this method.
- `executeBatch()` permits you to execute a collection of SQL update statements as a single database request.

In order to use batch updates, your application must disable auto commit options:

```
...
// turn off autocommit to prevent each statement from commiting
separately
con.setAutoCommit(false);

Statement stmt = con.createStatement();

stmt.addBatch("INSERT INTO employees VALUES(4671, ’James
Williams’)");
stmt.addBatch("INSERT INTO departments VALUES(560, ’Produce’)");
stmt.addBatch("INSERT INTO emp_dept VALUES( 4671, 560)");

//submit the batch of updates for execution
int[] updateCounts = stmt.executeBatch();
con.commit();
```

To remove all ganged statements from a batch operation before `executeBatch()` is called, (for example, because an error is detected), call `clearBatch()`.

NOTE

The JDBC 2.0 specification erroneously implies that batch updates can include data definition language (DDL) statements such as `CREATE TABLE`. DDL statements do not return a simple update count, and so cannot be grouped for a batch operation. Also, some databases do not allow data definintion statements in transactions.

Creating Distributed Transactions

The JDBC 2.0 Specification provides the capability for handling distributed transactions. A distributed transaction is a single transaction that applies to multiple, heterogeneous databases that may reside on separate server machines.

Distributed transaction support is already built into the iAS EJB container, so if you use EJBs that do not specify the `TX_BEAN_MANAGED` transaction attribute, you get automatic support for distributed transactions in your application.

In servlets and in EJBs that specify the `TX_BEAN_MANAGED` transaction attribute, you can still use distributed transactions, but you must manage transactions using the JTS `UserTransaction` class. For example:

```
InitialContext ctx = null;
String dsName1 = "jdbc/SampleDS1";
String dsName2 = "jdbc/SampleDS2";
DataSource ds1 = null;
DataSource ds2 = null;
```

```
    try {
        ctx = new InitialContext();
        ds1 = (DataSource)ctx.lookup(dsName1);
        ds2 = (DataSource)ctx.lookup(dsName2);

    } catch(Exception e) {
      e.printStackTrace(System.out);
    }

UserTransaction tx = ejbContext.getUserTransaction();

tx.begin();

Connection conn1 = ds1.getConnection();
Connection conn2 = ds2.getConnection();

// do some work here

tx.commit();
```

In this example, `ds1` and `ds2` must be registered with iAS as global datasources. In other words, their datasource properties files must include a `ResourceMgr` entry whose value must be configured at install time.

For example a global database properties file looks like:

```
DataBase=ksample
DataSource=ksample
UserName=kdemo
PassWord=kdemo
DriverType=ORACLE_OCI
ResourceMgr=orarm
```

In this example, `orarm` must be a valid `ResourceMgr` entry and must be enabled to get a global connection successfully. In order to be a valid `ResourceMgr` entry, an resource manager must be listed the registry in `CCS0\RESOURCEMGR`, and the entry itself must have the following properties.

```
DatabaseType (string key)
IsEnabled (integer type)
Openstring ( string type key)
ThreadMode ( string type key)
```

Working with Rowsets

A rowset is an object that encapsulates a set of rows retrieved from a database or other tabular data store, such as a spreadsheet. To implement a rowset, your code must import `javax.sql`, and implement the `RowSet` interface. `RowSet` extends the `java.sql.ResultSet` interface, permitting it to act as a JavaBeans component.

Because a `RowSet` is a JavaBean, you can implement events for the rowset, and you can set properties on the rowset. Furthermore, because `RowSet` is an extension of `ResultSet`, you can iterate through a rowset just as you would iterate through a result set.

You fill a rowset by calling the `RowSet.execute()` method. The `execute()` method uses property values to determine the datasource and retrieve data. The actual properties you must set and examine depends upon the implementation of `RowSet` you invoke.

For more information about the `RowSet` interface, see the JDBC 2.0 Standard Extension API Specification.

Using iASRowSet

iAS provides a rowset class called `iASRowSet`. `iASRowSet` extends `ResultSet`, so call methods are inherited from the `ResultSet` object. `iASRowSet` overrides the `getMetaData()` and `close()` methods of `ResultSet`.

The `RowSet` interface is fully supported except as noted in the following table.

Table 8-5 RowSet Interface Support Exceptions

Method	**Argument**	**Exception Thrown**	**Reason**
setReadOnly()	false	SQLException	iASRowSet is already read-only.
setType()	TYPE_SCROLL_INSENSITIVE	SQLException	`SCROLL_INSENSITIVE` is not supported.
setConcurrency()	CONCUR_UPDATABLE	SQLException	iASRowSet is read-only.
addRowSetListener()	any	None	Not supported.
removeRowSetListener()	any	None	Not supported.
setNull()	any type name	Arguments ignored	Not supported.

Table 8-5 RowSet Interface Support Exceptions

Method	Argument	Exception Thrown	Reason
setTypeMap()	java.util.Map	None	Map is a JDBC 2.0 feature that is not currently supported.

RowSetReader

`iASRowSet` provides a full implementation of the `RowSetReader` class.

RowSetWriter

`iASRowSet` is read-only, but an interface for this class is provided for future expansion. At present, its only method, `writeData()` throws `SQLException`.

RowSetInternal

This internal class is used by `RowSetReader` to retrieve information about the `RowSet`. It has a single method, `getOriginalRow()`, which returns the original result set instead of a single row.

Using CachedRowSet

The JDBC specification provides a rowset class called `CachedRowSet`. `CachedRowSet` permits you to retrieve data from a datasource, then detach from the datasource while you examine, and modify the data. A cached rowset keeps track both of the original data retrieved, and any changes made to the data by your application. If the application attempts to update the original datasource, the rowset is reconnected to the datasource, and only those rows that have changed are merged back into the database.

Creating a RowSet

To create a rowset in an iAS server application:

```
iASRowSet rs = new iASRowSet();
```

Using JNDI

JDBC 2.0 specifies that you can use the Java Naming and Directory Interface (JNDI) to provide a uniform, platform and JDBC vendor-independent way for your applications to find and access remote services over the network. For example, all JDBC driver managers, such as the JDBC driver manager implemented in iAS, must find and access a JDBC driver by looking up the driver and a JDBC URL for connecting to the database. For example:

```
Class.forName("SomeJDBCDriverClassName");
Connection con =

DriverManager.getConnection("jdbc:iAS_subprotocol:machineY:portZ");
```

This code illustrates how a JDBC URL may not only be specific to a particular vendor's JDBC implementation, but also to a specific machine and port number. Such hard-coded dependencies make it hard to write portable applications that can easily be shifted to different JDBC implementations and machines at a later time.

In place of this hard-coded information, JNDI permits you to assign a logical name to a particular datasource. Once you establish the logical name, you need only modify it a single time to change the deployment and location of your application.

JDBC 2.0 specifies that all JDBC datasources should be registered in the jdbc naming subcontext of a JNDI namespace, or in one of its child subcontexts. The JNDI namespace is hierarchical, like a file system's directory structure, so it is easy to find and nest references. A datasource is bound to a logical JNDI name. The name identifies a subcontext, "jdbc", of the root context, and a logical name. In order to change the datasource, all you need to do is change its entry in the JNDI namespace without having to modify a line of code in your application.

For more information about JNDI, see the JDBC 2.0 Standard Extension API.

Chapter 9

Rich Client

This chapter explains the role of the Rich Client infrastructure within the iAS environment. Also described are the programming considerations that need addressing when implementing a Rich Client.

The following topics are presented in this chapter:

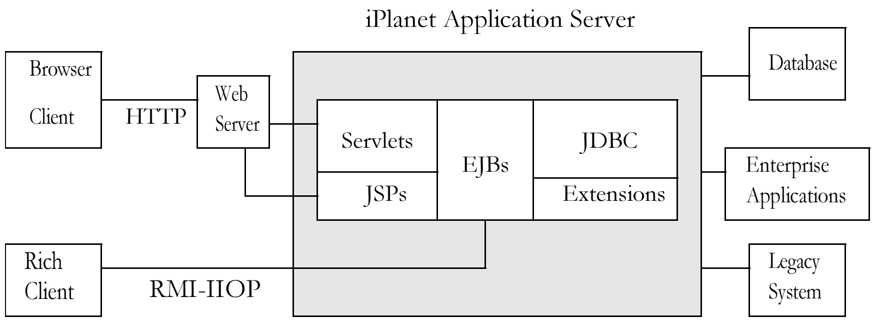
- "Overview of Rich Client"
- "Rich Client Architecture and Use Cases"
- "Value-added Features"
- "Developing for a Rich Client"
- "Sample code"

Overview of Rich Client

What is a Rich Client

The Rich Client is a stand-alone Java program that can directly access EJBs deployed on iAS. Traditionally, clients communicated with iAS through the web-path, i.e. by speaking HTTP to server components such as JSPs and Servlets which in turn had access to EJBs within the context of the server. The J2EE v1.2 specification, however, requires that stand-alone clients be able to talk to iAS using the RMI-IIOP standard. Chapter 9 of the J2EE v1.2 specification also requires that these stand-alone clients operate within the context of an Application Client Container (ACC) that isolates server-specific issues, leaving the clients completely portable. Through its Rich Client infrastructure, iAS allows Java clients to directly access EJBs on iAS. These clients could operate within an ACC that ships with iAS as required by the J2EE ACC specification or more straight forward direct access (non ACC path) the way Java programmers are used to writing them.

The diagram below is a schematic representation of the iAS architecture and illustrates the difference between the Rich Client and web paths. While browser clients communicate with iAS using HTTP through a Web Server, Rich Clients circumvent the Web Server and directly access EJBs as RMI-IIOP clients.



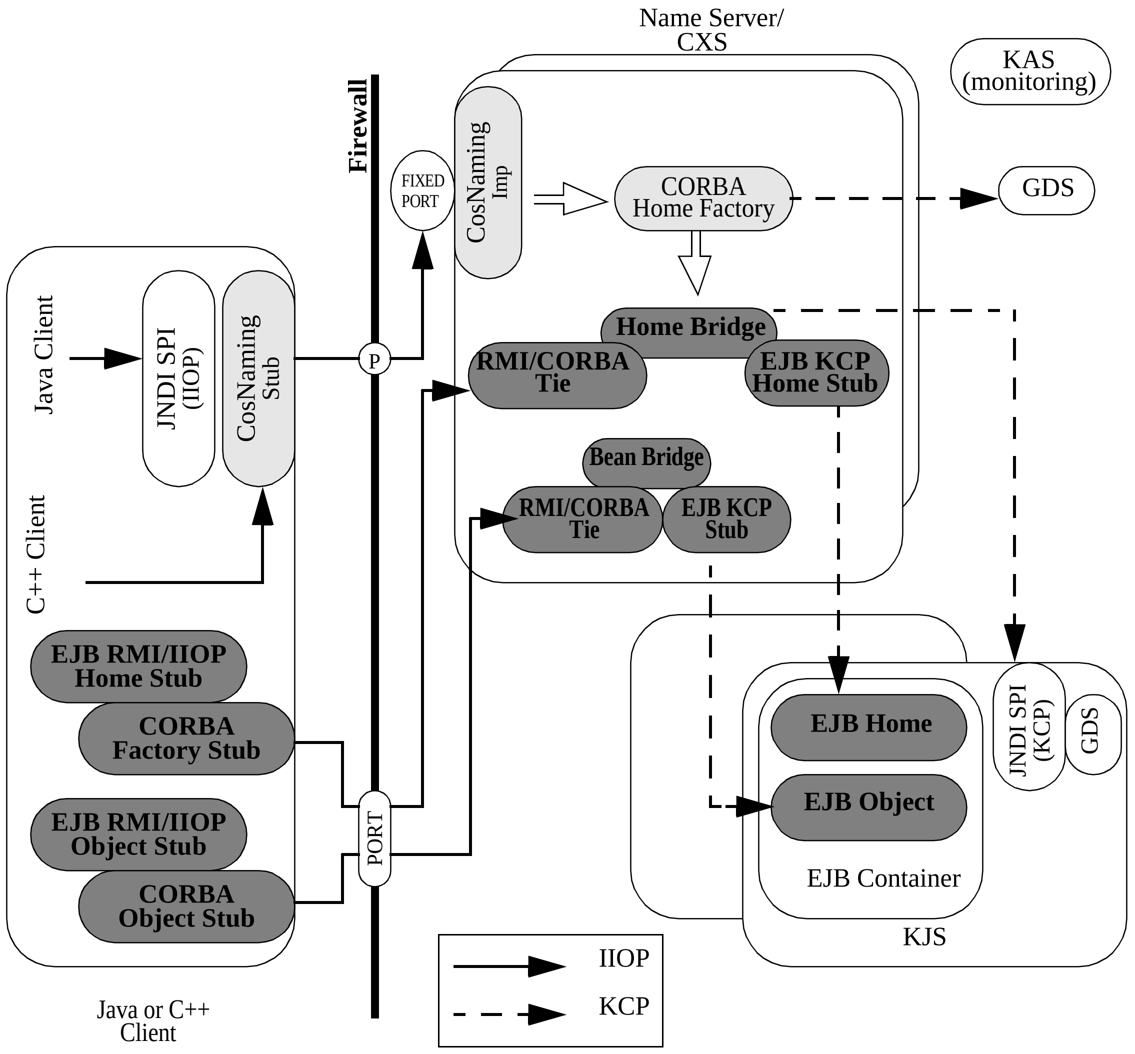
The Rich Client is a first-tier program that executes in its own Java Virtual Machine (JVM), possibly in an ACC. Deploying the Rich Client (through the ACC) requires the specification of deployment descriptors using XML. Refer to the J2EE Specification, v1.2 for more information about application clients and their deployment descriptors.

Rich Client Architecture and Use Cases

Architecture

iAS supports Rich Clients using a Java engine called the CORBA Executive Server (CXS). The CXS acts as a bridge between Rich Clients that use the Internet Inter-ORB Protocol (IIOP) and the EJBs on iAS' Java engine(s). For every EJB on iAS, the CXS instantiates two objects, one each for the EJB home and remote interfaces, that act as bridges between IIOP and iAS' internal communication protocol.

The following diagram illustrates the various components of the Rich Client architecture.



Use Cases

Rich Client connectivity to iAS is primarily expected to be used in any of the following three scenarios:

- From a stand-alone Java (and possibly C++) client that directly communicates with iAS. This is the scenario we have talked about up until now.
- A business-to-business scenario wherein applications in other applications servers can work with EJBs hosted on iAS. These components would use RMI-IIOP (or possibly) IIOP from C++) and obtain references to iAS EJBs.
- From a Web-Server hosted component such as Servlets and JSPs, Microsoft ASP etc., Beans on iAS can be accessed from any Web-Server (like Apache, IIS, NES etc.).

Value-added Features

Load Balancing

The CXS currently uses a simple round-robin scheme for load balancing. It obtains a static engine list at system start-up and uses this static list to pick the target engine for an EJB home to be hosted on. Subsequent lookups for that EJB home, bean creations on that home and business method invocations on the created beans will go to the same target engine. In the current release it's possible to configure multiple CXS but the client will need to manually load balance across them. The next release of iAS will support a client-initiated load balancing scheme.

Failover

When connecting to iAS through the Rich Client path, there are two levels of failover that are available to the client. The first of these is the failover support that iAS provides for stateful session beans hosted on iAS' Java engines. In case a Java engine crashes and the Administrative Server restarts the engine, the state of session beans is restored to that before the crash. The second level of failover support happens in the CXS for both session and entity bean handles / object references. In case of a CXS crash, the states of bridge objects for all EJBs are restored to that before the crash. The handle / object reference failover allows the client to get persistent references that survive client or CXS crashes. For failover to be available for an EJB, an XML deployment descriptor entry is required at deployment time. For more information on deployment descriptor settings, see Chapter 10, “Packaging for Deployment.

Security

Security on the Rich Client path is integrated with iAS' security infrastructure. The CXS uses iAS' security manager to authenticate clients with user information stored in LDAP. Client credentials are passed from the client, through the bridge to EJBs. A client side callback initiates client login (with username and password). The type of the object to be instantiated to obtain this information is specified through an environment setting on the client. In case of authentication failure, the client-side is setup to retry the login process. The number of retries is currently hard-coded to three.

Developing for a Rich Client

Server Side

The server-side work involved in working with the Rich Client infrastructure is almost exactly the same as that for the web-path. EJBs that the Rich Client needs to access are developed and deployed in exactly the same way (see Section “EJB XML DTD,” in Chapter 10, “Packaging for Deployment"). There are however two key additional steps that are required when using the Rich Client path.

1. The CXS needs to be started up and configured using the iAS Administrator Tool (iASAT). The CXS is not setup during iAS installation. Instead, it is configured using iASAT when required. The CXS is configured to listen on an IIOP port and an internal engine port. The IIOP port is used by the client to talk to CXS.
2. The iAS Deployment Tool (iASDT) should be used to obtain a single JAR file corresponding to the application that needs Rich Client connectivity. This JAR file contains RMI/IIOP stubs that are generated from the EJB home and remote interfaces. The iASDT ensures that all the server-side ties and stubs for the application are made available to the CXS.

Client Side

The client-side of the Rich Client system requires the following pieces, at a minimum, in order to be completely up and running. All the classes in the following pieces need to be in the client-side CLASSPATH.

1. The following iAS-specific components that come bundled in one tar file called iasclient.tar (zip file iasclient.zip on NT) on the iAS 6.0 CD:
    a. A J2SDK SE 1.2 with the Javasoft ORB packaged as an extension. In order to be able to use the rich client infrastructure, a client has to necessarily use this JDK. The reason for this is that this JDK has the Javasoft ORB packaged inside it as an extension unlike the pristine JDK.
    b. iAS-specific client-side classes packaged into iasclient.jar
    c. J2EE APIs packaged into javax.jar, jms.jar, mail.jar and servlet.jar
2. The client-side JAR file generated on the server using the iASDT.
3. A security principal class that implements the `com.netscape.ejb.client.IUserPrincipal` interface. This class is instantiated once by the Rich Client system and the `setPrincipal()` method is invoked on this instance whenever a client needs to be logged in. (See “java com.netscape.ejb.client.AppContainer <client ear file> -iasXml <ias xml file> [-resourceXml <resource xml file>]” section.)
4. The RMI/IIOP client needs to be developed and inserted in the client’s CLASSPATH.

The client is invoked through its main() method like any other standalone Java program. Look at the "Sample code" section for an example.

Application Client Container

When using the Application Client Container, a client has a few additional things to take care of.

1. One additional jar, namely iasacc.jar, contained in iasclient.tar/iasclient.zip (mentioned in (1) above) needs to get into the client’s CLASSPATH.
2. A J2EE v1.2-compliant EAR file will need to be created. This EAR file should contain
    a. The RMI/IIOP client mentioned in (4) above.

b. A J2EE v1.2 XML descriptor file (application-client.xml). Refer to the J2EE 1.2 specifications for an explanation.

   See Chapter 11, "Packaging for Deployment".

3. An iAS-specific XML descriptor file (typically ias-application-client.xml). Refer to Appendix C section for an example.
4. An optional resource XML descriptor file (typically resource-application-client.xml) required only if the client needs to access resources such as JDBC drivers. Refer to Appendix C section for an example with explanations.

The command to invoke the client through the Application Client Container looks like this:

```
java com.netscape.ejb.client.AppContainer <client ear file> -iasXml
<ias xml file> [-resourceXml <resource xml file>]
```

Sample code

Sample client code

```
import java.rmi.*;

import javax.naming.*;

import javax.rmi.PortableRemoteObject;

import java.util.Properties;

public static void main(String [] argv) {

  Properties env = new Properties[];

  Context nctx;

  String beanName = "java:comp/env/bean";
```

The beanName string is used later to do a lookup. A name starting with "java:comp" may be used only if the application client container is used. If doing a lookup directly from a standalone client the absolute JNDI name of the bean needs to be used. The absolute name looks like: "ejb/<module-name>/<bean-name>.

```
  BeanHome beanHome;

  BeanRemote beanRef;
  // Environment entries when creating Initial Naming Context
  env.put("java.naming.factory.initial",
          "com.sun.jndi.cosnaming.CNCtxFactory");
```

The rich client requires client to set few properties. The above property specifies that the client code would use CORBA CosNaming SPI from Sun's JNDI implementation to communicate with the CosNaming server running inside the CXS.

```
  // Client security principal
  env.put("com.netscape.ejb.client.PrincipalClass",
          <my principal class name>);
```

If the client code wants to support secure invocations as mentioned earlier IUserPrincipal interface needs to be implemented and the class file name specified in the above property.

```
  // Host and port number of CXS Naming Service
  env.put("java.naming.provider.url",
          "iiop://"+<host>+":"+<port>);
```

The host and port of the CXS the client wants to connect to needs to be specified here. The iiop port number for CXS was probably configured when the CXS was created using iASAT.

```
  try {
    nctx = new InitialContext(env);
```

The creation of initial context triggers the first invocation of the IUserPrincipal callback's setPrinicpal method if the PrincipalClass property is specified.

```
  }
  catch (Exception e) {}
// Get EJB Home
  Object o1 = nctx.lookup(beanName);
  beanHome = (BeanHome)PortableRemoteObject.narrow(o1,
BeanHome.class);


  // Create EJB instance
  beanRef = (BeanRemote)beanHome.create();
  // Invoke business method on beanRef
  beanRef.doSomething();
  // Remove bean instance
  beanRef.remove();
}
```

Sample Principal Class

The IUserPrinicpal interface can be implemented in several ways. The simplest would be to pop up a dialog in the setPrinciapl callback to capture a user/password pair and store them in the username and password private string fields. Then whenever an EJB invocation occurs from the client, the getUserId() and getPassword() methods would be used to set the security context that is propagated by the client.

The CXS will try to authenticate the user/password with the iAS security manager, if an authentication exception occurs in CXS the client-side orb would be notified and would then re-trigger the setPrincipal callback to give another chance to capture the correct user/password information. The orb would re-try the request automatically 3 times after which the Authentication exception is propagated up to the client code.

Another valid implementation of IUserPrincipal could support multiple user identities in the same client JVM. This can be done by using ThreadLocal variables to store username and password. The methods in the IUserPrincipal implementation would then need to be THreadLocal savvy.

```
import com.netscape.ejb.client.IUserPrincipal;
public class Principal implements IUserPrincipal {
  private String username;
  private String password;
  public void setPrincipal() {
    <Pop up GUI to take user name and password>
  }
  public String getUserId() {
    return username;
  }
  public String getPassword() {
    return password;
  }
}
```

Chapter 10

Packaging for Deployment

This chapter describes the contents of iAS modules and how they are packaged to create an iAS application.ear file used for application deployment. The iAS modules include J2EE standard elements and iAS specific elements. Only iAS specific information is detailed in this chapter. Through out this chapter for J2EE specific information, you will find a reference to the appropriate J2EE specification.

The following topics are presented in this chapter:

- Overview of Packaging and Deployment
- Introducing XML DTDs
- Application XML DTD
- Web Application XML DTD
- EJB XML DTD
- Rich Client XML DTD
- Resource XML DTD

Overview of Packaging and Deployment

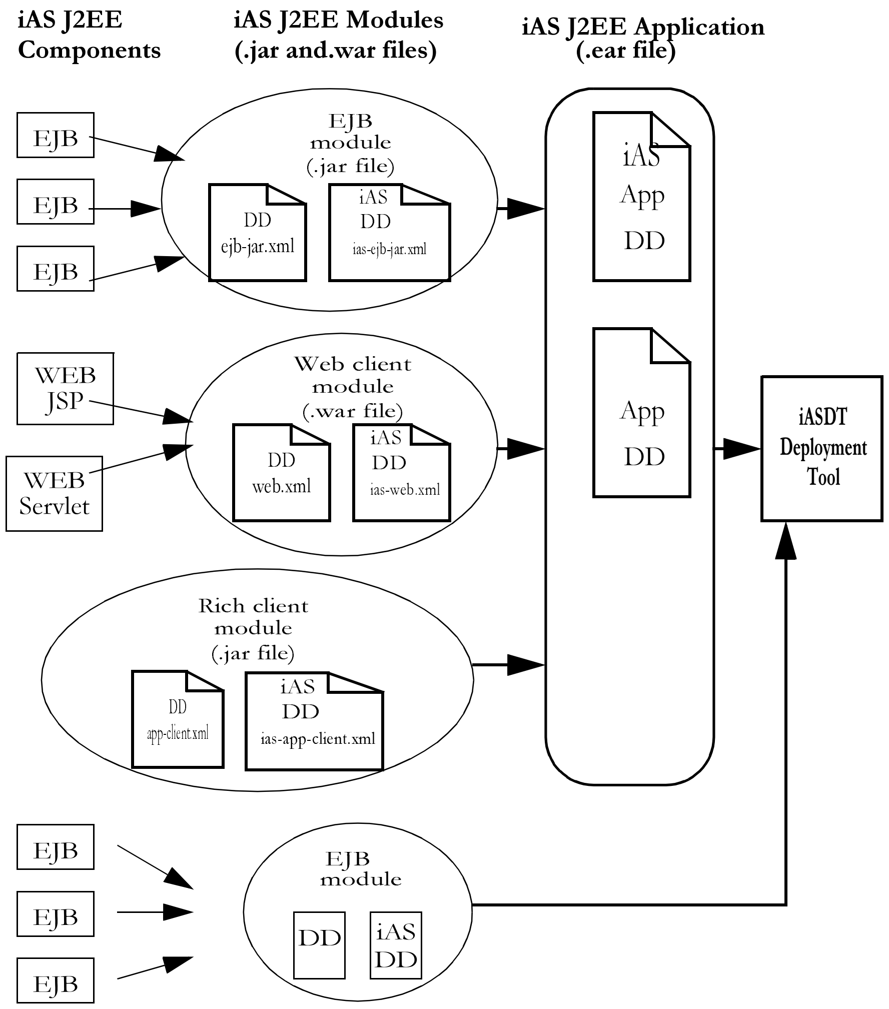
An iAS application is comprised of one or more iAS application modules, an iAS deployment descriptor (DD) and a J2EE application deployment descriptor (DD). All of these items are packaged, using the Java ARchive (JAR) file format, into one file with an extension of .ear (Enterprise ARchive). The deployment descriptors are used by the iAS Deployment Tool (iASDT) to deploy the application components and to register the resources with the iAS.

Each of these application modules consist of either EJB, Web-application, Rich Client, or resource components, as well as an iAS deployment descriptor, and a J2EE deployment descriptor. For example, an EJB module would have one or more EJB components, an EJB deployment descriptor (DD) and an iAS EJB deployment descriptor (DD). There are two DDs for each module; one is a J2EE standard DD and the other is an iAS specific DD specified by this chapter.

Portions of the deployment descriptors for the application modules have been standardized by the J2EE specification, v1.1. For more information on these standards refer to the following specifications:

- Java 2 Platform Enterprise Edition Specification, v1.2, Chapter 8 Application Assembly and Deployment - J2EE:application XML DTD
- Java 2 Platform Enterprise Edition Specification, v1.2, Chapter 9 Application Clients - J2EE:application-client XML DTD
- JavaServer Pages Specification, v1.1, Chapter 7 JSP Pages as XML Documents
- JavaServer Pages Specification, v1.1, Chapter 5 Tag Extensions
- Java Servlet Specification, v2.2 Chapter 13 Deployment Descriptor
- Enterprise JavaBeans Specification, v1.1, Chapter 16 Deployment Descriptor

The diagram below illustrates how components are package into modules and then assembled into an iAS application .ear file ready for deployment.



Introducing XML DTDs

The Document Type Definition (DTD) defines the XML grammar for the deployment descriptors. There are two levels of deployment descriptors; there are application level descriptors, and component level descriptors. Applications are comprised of one or more components, one J2EE application deployment descriptor and one iAS application deployment descriptor.

The J2EE platform provides packaging and deployment facilities. These facilities use Java ARchive (JAR) files as the standard package for components and applications, and XML-based deployment descriptors for customizing parameters. See chapter 7 “Packaging and Deployment” of the Developing Enterprise Applications with the Java 2 Platform, Enterprise Edition, v 1.0 for more information on the J2EE packaging and deployment process.

Application Deployment Descriptor

The deployment descriptor for an application, lists the application’s components as modules. An iAS application will have two application deployment descriptors; one is a standard J2EE application DD and the other is an iAS application DD. The iAS application DD describes iAS specific deployment information for the iAS value-added features.

Component Deployment Descriptors

On the component level, each group of components packaged into a module will also have a J2EE component DD and an iAS component DD. An iAS application supports e web application, EJB, rich client and resource components. For each of these component types, there is a DTD to define the XML elements supported for that type of descriptor.

Creating Deployment Descriptors

All the deployment descriptors (DDs) needed for an iAS application are created using iPlanet Application Server Deployment Tool (iASDT). Refer to the iASDT documentation for further details on these procedures.

Deployment Descriptors

The application server uses deployment descriptors (DD) to properly deploy the application. The DDs are in the form of XML files that contain metadata describing deployment information about the J2EE modules (such as servlets, JSPs and EJBs) that make up your application. The information in each XML file is stored in a registry internal to the application server.

Each application module must have an iAS XML DD file, and a J2EE XML DD file. Additionally, each application component must be associated with a globally unique identifier, or GUID.

The following is a list of all the types of deployment descriptors supported by iAS:

- application DD and iAS application DD
- web application DD and an iAS web application DD
- EJB DD and an iAS EJB DD
- application-client DD and an iAS Rich Client DD
- iAS resource DD

Document Type Definition (DTD)

The Document Type Definition (DTD) describes the structure and properties of a class of XML DD files. Each DD will have exactly one element that completely contains all other elements (or sub elements).

The descriptions of the elements found in the XML files are presented in a table format. These element tables have several fields that describe the purpose of the element and settings for the element's parameters. Some elements are hierarchical, that is they have parameters that are other elements (or sub elements). If a parameter contains an element, the description for that element can be found in another table describing that element.

Table 10-1 Document Type Definition

Element	Name of the element as it appears in XML file, and a description of the element.
Sub Elements	Lists the elements contained by this element.

Table 10-1 Document Type Definition

Repeat Rule	Describes how many times this parameter is defined: “one and only one” means this parameter is not optional, it must be defined one time only. “zero or one” means this parameter is optional, it may be undefined or defined one time only <?> “one or many” means this parameter is not optional, it must be defined at least one time or multiple times <+> “zero or many” means this parameter is optional or may be defined many times <*>
Contains	This field describes the contents of the parameter. The contents may be a string, boolean, an integer or an element. If this field is an element, the definition of the element will be found in another table for that element.
Default	Describes the value of this parameter if it is not defined.
Description	The parameter description for this element.

The iAS Registry

The iAS registry is a collection of application metadata, organized in a tree, that is continually available in active memory or on a readily-accessible directory server. The process by which iAS gains access to servlets, EJBs, and other application resources is called registration, because it involves placing entries in the iAS registry for each item.

You can change some of the information in the registry at run-time using the iAS Administrator Tool (iASAT). For more information about the registry, and the iASAT, see the *Administration Guide*.

GUIDs

A GUID is a 128-bit hexadecimal number that is assigned to an EJB and Servlet, and optionally to JSPs and is automatically generated by the iAS Deployment Tool. The GUID has the following format:

{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}

GUIDs are guaranteed to be globally unique, which makes them ideal for identifying components in a large-scale heterogeneous system such as an iAS application.

GUIDs are normally assigned automatically by the deployment tool, or by iPlanet Application Builder. You can generate a GUID manually by using a utility named `kguidgen`. `kguidgen` is installed by default into the directory `BasePath/bin`. That directory must be listed in your search path (your `PATH` environment variable in order to generate a GUID.

To generate a new GUID, simply run `kguidgen` from a command line or window.

Application XML DTD

The application Document Type Definition (DTD) describes the application Deployment Descriptor (DD). There is a J2EE application DD, defined by the J2EE specification, and an iAS application DD that is defined in this chapter.

The iAS Deployment Tool (iASDT) is used to create the application DD, and also to deploy the application. Refer to the iASDT documentation for further details on these procedures.

J2EE Application DTD

See the Java Platform 2 Enterprise Edition, specification, v1.2 section 8.4 “J2EE:application XML DTD”, for the application DTD description.

iAS Application DTD

This is the IAS 6.0 specific XML DTD for the Application ear file

Table 10-2 ias-app

ias-app	**The root element of the iAS application deployment descriptor.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
role-mapping	zero or more	elements	none	This field create the mapping between the role name as it is known in the AppComponent, and then map them onto one or more LDAP defined user, group, etc. The deployment code simply treats each role-impl as an opaque string that's interpreted by the security infrastructure

Table 10-3 role-mapping

role-mapping	**Maps role names to LDAP user, groups.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
role-name	one and only one	string	none	The name of the role as referred to in the <security-role> element
role-impl	one and only one	elements	none	The string used to represent an LDAP group / user thing that makes up a particular role-name. A role imple could be any number of groups and / or users.

Table 10-4 role-impl

role-impl	**.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
group	zero or many	string	none	The LDAP specific string that corresponds to a particular LDAP group
user	zero or many	string	none	The LDAP specific string that corresponds to a particular LDAP user

Sample Application XML DD File

```
<?xml version="1.0"?>

<!DOCTYPE application PUBLIC ’-//Sun Microsystems, Inc.//DTD J2EE
Application 1.2//EN’
’http://java.sun.com/j2ee/dtds/application_1_2.dtd’>

<application>

<description>Application description</description>

  <display-name>estore</display-name>

  <module>

    <ejb>estoreEjb.jar</ejb>

  </module>

  <module>

    <web>

      <web-uri>estore.war</web-uri>

      <context-root>estore</context-root>

    </web>

  </module>

  <security-role>

    <description>the customer role</description>

    <role-name>customer</role-name>

  </security-role>

</application>
```

Web Application XML DTD

This section provides an overview of web applications, the web application module and the web application deployment descriptor (DD). The DDs are created using iPlanet Application Server Deployment Tool (iASDT). Refer to the iASDT *Administration and Deployment Guide* for further details on these procedures.

Web Application Overview

Web applications run on web servers and may consist of servlets, JSPs, JSP Tag libraries, html pages, classes and other resources. The location of a web application is rooted at a specific path within the web server. An instance of a web application must only be run on one Virtual Machine (VM) at any one time, unless the application is marked as distributable by its deployment descriptor. When marked as distributable, the application may run on more than one VM at any one time and must follow a more restrictive set of rules outlined by the Java Servlet 2.2 specification.

The web application is a composite of the following items:

- Servlets
- JavaServer Pages (JSPs)
- Utility Classes
- Static documents (html, images, sounds, etc.)
- Client side applets, beans and classes
- Descriptive meta information bundling all the above items together

The web application is created by first assembling all the needed Web components into a Web application module along with module deployment descriptors (DD), and then packaging this web application module with all other modules that will be used by the J2EE application along with the application deployment descriptors (DD) into the final web application that is ready for deployment. A complete description of the J2EE assembly and deployment is available in the J2EE specification, chapter 8. See the URL:**`http://java.sun.com/j2ee.`**

Web Application XML DTD

This section provides the XML Document Type Definition (DTD) for the iAS specific Web application deployment descriptor (DD). Refer to section 8.4 of the J2EE specification for the standard J2EE application DD.

The Web application DD supports the definition of elements that provide the following information:

- servlet information
- session information
- EJB reference information

- Resource reference information
- Specifying Servlet Information

Element for specifying an iAS Web Application

Table 10-5 <ias-web-app>

ias-web-app	**The root element of the iAS web application deployment descriptor.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
servlet	zero or more	elements	none	Contains servlet configuration information.
session-info	one and only one	elements	none	Specifies session information.
ejb-ref	zero or more	elements	none	Specifies storage place foe ABOLUTE JNDI name of the corresponding J2EE XML `ejb-ref` entry.
resource-ref	zero or more	elements	none	Specifies storage place for the ABSOLUTE JNDI name that the ejb-link in the corresponding J2EE XML file ejb-ref entry.
nlsinfo	one and only one	elements	none	Descriptor for nls settings.
role-maping	zero or many	elements	none	LDAP role mapping descriptor.

Elements for Specifying Servlet Information

servlet	**Contains configuration information about a servlet.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
servlet-name	one and only one	string	none	The name of the servlet. This name must match the “servlet name” parameter in the J2EE web app XML exactly.

servlet	**Contains configuration information about a servlet.**			
guid	one and only one	string	none	A string representing the guid for the servlet.
servlet-info	zero or one	elements	none	Optional characteristics of the servlet.
validationRequired	zero or one	boolean	“false”	Specifies if the input parameter needs to be validated.
error-handler	zero or one	string	none	Describes the error handler for the servlet.
parameters	zero to more	elements	none	Describes all the input parameters to be validated.
param-group	zero to more	elements	none	Each parameter group is represented by an event source name and the associated parameters.

servlet-info	**Describes optional characteristics of the servlet.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
sticky	zero or one	boolean	“false”	If sticky is “true”, the servlet exhibits session affinity and is only load-balanced if no session exists. Once a session is created in a given engine, subsequent requests for sticky servlets will continue to be routed to that same engine.
encrypt	zero or one	boolean	“false”	Optional flag indicating whether communications to the servlet are encrypted (“true”) or not (“false”).
caching	zero or one	elements	none	Specifies caching criteria for the servlet.
number-of-singles	zero or one	integer	"10"	The number of objects in the servlet pool when `SingleThread` mode is used.
disable-reload	zero or one	boolean	"false"	This is used to disable reloading of servlets when dirty. Legal values are "true" or "false".

servlet-info	Describes optional characteristics of the servlet.			
server-info	zero or many	elements	none	Optional information about the server including enabling / disabling of the server and loadbalancing.

Element	Repeat Rule	Contains	Default	Description
validation-required	one and only one	boolean	"false"	Specifies whether or not the input parameters should be verified.

caching	Describes caching criteria for the servlet. Caching can be disabled by not defining the caching element.			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
cache-timeout	one and only one	integer	none	Sets the time-out for the caching of the servlet (in seconds). If the value is 0, caching is disabled.
cache-size	one and only one	integer	none	Sets the size of the cache. A value of “0” disables caching.
cache-criteria	one and only one	string where the syntax is any value of arg in the input parameter list	none	Criteria expression containing a string of comma-delimited descriptors, each descriptor defining a match with one of the input parameters to the servlet.

Examples for Setting Cache Criteria

The following examples provide some common usages and settings for the cache-criteria element.

Example 1

```
<cache-criteria>EmployeeCode</cache-criteria>
```

This means caching is enabled if "EmployeeCode" is in the input parameter list.

Example 2

```
<cache-criteria>stock=NSCP</cache-criteria>
```

This means caching is enabled is the value of the "stock" input parameter is "NSCP"

Example 3

```
<cache-criteria>*</cache-criteria>
```

This means that caching is enabled whenever the value of the input parameters is the same as the value cached.

Example 4

```
<cache-criteria>dept=sales|marketing|support</cache-criteria>
```

This means caching is enabled if the value of the "dept" parameter is either of sales or marketing or support.

Example 5

```
<cache-criteria>salary=40000-60000</cache-criteria>
```

This means caching is enabled when the value of the input parameter "salary" is between 40000 and 60000.

server-info	**Optional extra information about the server, including enabling/disabling of the server, and load-balancing.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
server-ip	one and only one	string	none	Server’s IP address.
server-port	one and only one	string	none	Executive Server’s port number.
sticky-lb	zero or many	boolean	"sticky" setting of the servlet-info	Sets sticky load balancing. Legal values are "true" or "false". If set will override setting of the servlet-info.
enable	zero or many	boolean	"true"	Specifies if the server is enabled or not. Legal values are "true" or "false".

parameters	**Describes all the input parameters to be validated.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
param	zero or more	elements	none	Specifies each parameter by name and the rules that are applied to it for validation.

param	**Each parameter is represented by a name and the rules that are applied to it for validation.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
param-name	one and only one	string	none	Name of the input parameter.
input-fields	one and only one	elements	none	This describes the details of the input parameter.

input-fields	**Describes the details of the input parameter.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
input-required	zero or one	boolean	none	Specifies whether the input parameter is required to exist, i.e. whether the field should be part of the input list.
input-rule	zero or one	string	none	Specifies the input rule being applied for validation on the input parameter.
format	zero to one	string in date/time format	none	Specifies the format for date/time to be applied for validation on the input parameter.
in-session	zero to one	string	none	Specifies if the parameter is in cache (session) for validation.

input-fields	**Describes the details of the input parameter.**			
param-error-handler	zero or one	string	none	Specifies the error handler for the parameter.

param-group	**Description: Each parameter group is represented by an event source name and the associated parameters.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
param-group-name	one and only one	string	none	Name of the parameter group.
param-input	one or more	string	none	Name of the parameter input associated with the parameter group.

Elements for Specifying Session Information

session-info	**Specifies session information.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
impl	one and only one	string of either “distributed” or “lite”	none	A session can either be distributed, fault-tolerant or a lightweight local-session only.
timeout-type	zero or one	string of either “last-access” or “creation”	‘last-access’	Session timeouts are normally measured in “time since last access”. Alternatively, an absolute timeout can be specified as “time since session creation”
timeout	zero or one	positive integer	30 minutes	This is the number of session timeout seconds before a timeout. If unspecified, a system-wide default session timeout is used
secure	zero or one	boolean	“false”	This specifies that the session can only be visible to a secure (HTTPS) server.

session-info	**Specifies session information.**			
domain	zero or one	string name of the domain that set the cookie	none	This specifies that the domain for the application. This is used to set the cookie for the session for that domain. The domain string argument must contain at least 2 or 3 periods (3 period-domains apply to domains like acme.co.uk). If the domain is set to acme.com, then the session is visible to foo.acme.com, bar.asme.com, etc.
path	zero or one	String value of the URL for the session cookie starting with "/".	The URL that created the cookie.	This specifies the path for the session cookie. A non-existent path implies that the same path as the one that sets the cookie is used. For example, the path "/phoenix" will match "/phoenix/types/bird.html" and "/phoenix/birds.html".
scope	zero or one	String identifying the other application.	none	Grouping name that selects which other applications can access this session. For example, if the domain is set to acme.com, then the session is visible to foo.acme.com, bar.asme.com, etc.
dsync-type	zero or one	string of either "dsync-local" or "dsync-distributed"	none	Specifes the type of DSync session.

Elements for Specifying EJB Reference Information

ejb-ref	**This is the storage place for the ABSOLUTE jndi name that the ejb-link in the corresponding J2EE XML file ejb-ref entry.**			
Sub Element	**Repeat Rule**	**Contains**	**Default**	**Description**
ejb-ref-name	one and only one	string	none	The ejb-link in the corresponding J2EE XML file ejb-ref entry.
jndi-name	one and only one	string	none	The ABSOLUTE jndi name.

Elements for Specifying Resource Reference Information

resource-ref	**This is the storage place for the ABSOLUTE jndi name that the resource-ref in the corresponding J2EE XML file resource-ref entry.**			
Parameter	**Repeat Rule**	**Contains**	**Default**	**Description**
resource-ref-name	one and only one	string	none	This is the name of the resource-ref in the corresponding J2EE XML file resource-ref entry.
jndi-name	one and only one	string	none	The ABSOLUTE jndi name.

Elements for Specifying nls Settings

nlsinfo	**The nlsinfo element contains configuration information about application nls settings..**			
Sub Element	**Repeat Rule**	**Contains**	**Default**	**Description**
locale-charset-map	zero or many	elements	none	Contains locale and corresponding character set.
default-locale	one and only one	string	none	Default locale.

locale-charset-map	Descriptor for locale and corresponding charater set.			
Sub Element	**Repeat Rule**	**Contains**	**Default**	**Description**
locale	one and only one	string	none	Name of locale.
charset	one and only one	string	none	Default locale.

Elements for Specifying Role Mapping

role-mapping	Descriptor for mapping roles to LDAP user, groups.			
Sub Element	**Repeat Rule**	**Contains**	**Default**	**Description**
role-name	one and only one	string	none	Name of the role as referred to in the <security-role> element.
role-impl	one and only one	elements	none	The string used to represent a LDAP group / user thing that makes up a particular role-name. A role-impl could be any number of groups and / or users.

role-impl	Descriptor for role implementation.			
Sub Element	**Repeat Rule**	**Contains**	**Default**	**Description**
group	zero or many	string	none	LDAP specific string that corresponds to a particular LDAP group.
user	zero or many	string	none	LDAP specific string that corresponds to a particular LDAP user.

EJB XML DTD

This section provides the EJB Document Type Definition (DTD) that is used by the EJB deployment descriptors. The DDs are created using iPlanet Application Server Deployment Tool (iASDT). Refer to the iASDT documentation for further details on how to create the DDs.

EJB Jar File Contents

The standard format used to package enterprise Beans is the ejb-jar file. This format is the contract between the Bean Provider and Application Assembler, and between the Application Assembler and the Deployer.

The ejb-jar file must contain the deployment descriptor (DD) as well as all the class files for the following:

- The enterprise bean class.
- The enterprise bean home and remote interface.
- If the bean is an entity bean, the primary key class.

In addition, the ejb-jar file must contain the class files for all the classes and interfaces that the enterprise bean class, and the remote home interfaces depend on.

Specifying Parameter Passing Rules

When a servlet or EJB calls another bean that is co-located within the same process, iAS does not perform marshalling of all call parameters by default. This optimization allows the co-located case to execute far more efficiently than if strict “by-value” semantics were used. In certain cases, however, you may want to ensure that parameters passed to a bean are always passed by value. The iAS supports the ability to mark a bean or even a particular method within a bean as optionally requiring pass by value semantics. The parameter passing method used by the EJB is defined by the “pass-by-value” element. See the “pass-by-value” element description in the “session” or “entity” element table in this section. Because this option can decrease performance by greatly increasing call overhead, the default value is “false” if this entry is unspecified.

EJB iAS XML DTD

The following is the iAS specific XML DTD for EJB JAR files.

ias-ejb-jar	**The root element of the iAS web application deployment descriptor.**			
Sub Element	**Repeat Rule**	**Contains**	**Default**	**Description**
enterprise-beans	one and only one	element	none	The enterprise-beans element contains the declarations of one or more enterprise beans.

enterprise-beans	**The enterprise-beans element contains the declarations of one or more enterprise beans.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
session	one or the other	element	none	An element that declares all the ias specific session bean related deployment information
entity	one or the other	element	none	An element that declares all the ias specific entity bean related deployment information

session	**Declares all the ias specific session bean related deployment information. The ejb-name MUST be matched 1 to 1 with the ejb-name declared in the J2EE XML file.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
ejb-name	one and only one	string	none	The name of the ejb.
guid	one and only one	string	none	The guid of the ejb in question.
pass-timeout	one and only one	positive integer	none	Passivation timeout in seconds used by the container. This value can be changed during runtime by iASAT.

session	**Declares all the ias specific session bean related deployment information. The ejb-name MUST be matched 1 to 1 with the ejb-name declared in the J2EE XML file.**			
pass-by-value	one and only one	boolean	none	If “true”, marshalling of all call parameters to the EJB will be performed. If “false” and the beans are co-located, strict “by-value” semantics are not guaranteed.
session-timeout	one and only one	positive integer	none	
ejb-ref	zero or more	elements	none	This is a storage place for the ABSOLUTE jndi name that the ejb-link in the corresponding J2EE XML file ejb-ref entry.
resource-ref	zero or more	elements	none	This is a storage place for the ABSOLUTE jndi name that the resource-ref in the corresponding J2EE XML file resource-ref entry.
failoverrequired	zero or one	boolean	none	Indicates whether failover is required.

entity	**Declares all the ias specific entity bean related deployment information. The ejb-name MUST be matched 1 to 1 with the ejb-name declared in the J2EE XML file.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
ejb-name	one and only one	string	none	The name of the ejb.
guid	one and only one	string	none	The guid of the ejb in question.
pass-timeout	one and only one	positive integer	none	Passivation timeout in seconds used by the container. This value can be changed during runtime by iASAT.
pass-by-value	one and only one	boolean	none	If “true”, marshalling of all call parameters to the EJB will be performed. If “false” and the beans are co-located, strict “by-value” semantics are not guaranteed.
persistence-manager	zero or one	elements	none	Specifies persistence information.

entity	**Declares all the ias specific entity bean related deployment information. The ejb-name MUST be matched 1 to 1 with the ejb-name declared in the J2EE XML file.**			
pool-manager	zero or one	elements	none	Descriptor for cache pool attributes.
ejb-ref	zero or more	elements	none	This is a storage place for the ABSOLUTE jndi name that the ejb-link in the corresponding J2EE XML file ejb-ref entry.
resource-ref	zero or more	elements	none	This is a storage place for the ABSOLUTE jndi name that the resource-ref in the corresponding J2EE XML file resource-ref entry.
failover-required	zero or one	boolean	"false"	This indicates whether failover is required.
iiop	zero or one	boolean	"false"	Indicates if the bean is rich client enabled.
role-mapping	zero or many	elements	none	This descriptor creates the role mapping.

persistence-manager	**Defines all the persistence manager specific information.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
factory-class-name	one and only one	string	none	Name of the persistence manager factory class.
properties-file-location	one and only one	string	none	Location in jar file of properties file.

pool-manager	**Defines all the pool manager specific information.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**

pool-manager	**Defines all the pool manager specific information.**			
comit-option	one and only one	string value of "COMMIT_OPTION_C" or "COMMIT_OPTION_B"	"COMMIT_OPTION_C"	Option B: Between transactions, the Container caches a "ready" instance. Option C: Between transactions the Container will not cache a "ready" instance. (see EJB v1.1 spec. section 9.1.10 for more details.)
ready-pool-timeout	one and only one	positive integer	infinite	Ready pool timeout used by the container. This value can be changed during runtime by iASAT.
ready-pool-maxsize	one and only one	positive integer or "0" for infinite	infinite	Maximum size of the ready cache in number of entries. This value can be changed during runtime by iASAT.
free-pool-maxsize	one and only one	positive integer or "0" for infinite	infinite	Maximum size of the instance free pool in number of entries. This value can be changed during runtime by iASAT.

ejb-ref	**This is the storage place for the ABSOLUTE jndi name that the ejb-link in the corresponding J2EE XML file ejb-ref entry.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**

ejb-ref	**This is the storage place for the ABSOLUTE jndi name that the ejb-link in the corresponding J2EE XML file ejb-ref entry.**			
ejb-ref-name	one and only one	string	none	The ejb-link in the corresponding J2EE XML file ejb-ref entry.
jndi-name	one and only one	string	none	The ABSOLUTE jndi name.

resource-ref	**This is the storage place for the ABSOLUTE jndi name that the resource-ref in the corresponding J2EE XML file resource-ref entry.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
resource-ref-name	one and only one	string	none	This is the name of the resource-ref in the corresponding J2EE XML file resource-ref entry.
jndi-name	one and only one	string	none	The ABSOLUTE jndi name.

role-mapping	**Descriptor for mapping roles to LDAP user, groups.**			
Sub Element	**Repeat Rule**	**Contains**	**Default**	**Description**
role-name	one and only one	string	none	Name of the role as referred to in the <security-role> element.
role-impl	one and only one	elements	none	The string used to represent a LDAP group / user thing that makes up a particular role-name. A role-impl could be any number of groups and / or users.

role-impl	**Descriptor for role implementation.**			
Sub Element	**Repeat Rule**	**Contains**	**Default**	**Description**

role-impl	**Descriptor for role implementation.**			
group	zero or many	string	none	LDAP specific string that corresponds to a particular LDAP group.
user	zero or many	string	none	LDAP specific string that corresponds to a particular LDAP user.

Rich Client XML DTD

The Rich Client is an iAS specific type of J2EE client. A Rich Client supports the standard J2EE Application Client specifications and in addition supports direct access to iAS. For more information on Rich Client, refer to Chapter 10 "Rich Client".

A Rich Client jar file will contain two deployment descriptors (DDs) that are generated by the iASDT. One of these DDs is specified by the J2EE application-client XML Document Type Definition (DTD), that can be found in the J2EE Specification, v1.0 Chapter 9 Application Clients. The other DD contains iAS specific Rich Client elements; these elements are described in section <$paratext>" of this chapter.

For a sample Rich Client DD file, refer to section <$paratext>" in Appendix C, " Sample Deployment Files".

iAS Rich Client XML DTD

The ias-java-client-jar element is the root element of the Rich Client deployment descriptor.

Elements for Specifying EJB Reference Information

ejb-ref	**This is the storage place for the ABSOLUTE jndi name that the ejb-link in the corresponding J2EE XML file ejb-ref entry.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**

ejb-ref	**This is the storage place for the ABSOLUTE jndi name that the ejb-link in the corresponding J2EE XML file ejb-ref entry.**			
ejb-ref-name	one and only one	string	none	The ejb-link in the corresponding J2EE XML file ejb-ref entry.
jndi-name	one and only one	string	none	The ABSOLUTE jndi name.

Elements for Specifying Resource Reference Information

resource-ref	**This is the storage place for the ABSOLUTE jndi name that the resource-ref in the corresponding J2EE XML file resource-ref entry.**			
Sub Elements	Repeat Rule	Contains	Default	Description
resource-ref-name	one and only one	string	none	This is the name of the resource-ref in the corresponding J2EE XML file resource-ref entry.
jndi-name	one and only one	string	none	The ABSOLUTE jndi name.

Resource XML DTD

Each iAS resource, such as a JDBC datasource, Java Mail and JMS, has a resource XML file. This XML file contains entries that are used to register the resource with iAS. These entries define the way iAS connects to that resource. These files are generated by the iAS Deployment Tool (iASDT). This section describes the resource XML file entries. For information on how to generate these files, refer to the iASDT documentation.

Datasource XML DTD

This section describes the XML DTD for the iAS datasource.

ias-Datasource-jar	**This is the root element of the resource deployment descriptor**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**

ias-Datasource-jar	**This is the root element of the resource deployment descriptor**			
ias-resource	one and only one	element	none	Common element for all the resource deployment descriptors.

ias-resource	**Descriptor used for all the resources.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
resource	one and only one	elements	none	Common element for all the resource deployment descriptors.

resource	**Descriptor used for all the resources.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
jndi-name	one and only one	string	none	The absolute jndi name of the resource factory (i.e. jdb / foo).
jdbc	one or the other	elements	none	Descriptor for the JDBC data source.
jms	one or the other	string	none	Descriptor for the jms datasource.
mail	one or the other	string	none	Descriptor for the mail datasource.
url	one or the other	string	none	Descriptor for the url datasource.

jdbc	**Descriptor for the jdbc datasource.**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
database	one and only one	string	none	Name of the database to connect to.
datasource	one and only one	string	none	Name to assign to the datasource.
username	one and only one	string	none	Valid user name for this database.

jdbc	**Descriptor for the jdbc datasource.**			
password	one and only one	string	none	Valid password for the user name listed.
driver-type	one and only one	string field containing of one of the following: ORACLE_OCI (Oracle) DB2_CLI (DB2) INFORMIX_CLI (Informix) SYBASE_CTLIB (Sybase) ODBC (ODBC)	none	Backend-specific JDBC driver.
resource-mgr	zero or one	string	none	If this attribute is set, the datasource is available for distributed transactions by way of the resource manager listed. If this attribute is not specified, the datasource is only valid for a local database. The value must be a name that you created for a resource manager under the RESOURCEMGR key.

Rich Client Datasource XML DTD

ias-javaclient-resource	**Root element for the Rich Client datasource XML DD..**			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
jdbc	one or the other	elements	none	Descriptor for Rich Client JDBC settings.
jms	one or the other	string	none	Not yet defined.
jndi-name	one and only one	string	none	The ABSOLUTE jndi name.

jdbc	Descriptor for JDBC settings.			
Sub Elements	**Repeat Rule**	**Contains**	**Default**	**Description**
driverClass	one and only one	elements	none	
connectUr1	one and only one	string	none	.
userName	one and only one	string	none	
password	one and only one	string	none	

Chapter 11

Creating and Managing User Sessions

This chapter describes how to create and manage a session that allows user and transaction information to persist between interactions.

This chapter contains the following sections:

- Introducing Sessions
- How to Use Sessions

Introducing Sessions

The term *user session* refers to a series of user-application interactions that are tracked by the server. Sessions are used for maintaining user-specific state, including persistent objects (like handles to EJBs or database result sets) and authenticated user identities, among many interactions. For example, a session could be used to track a validated user login followed by a series of directed activities for that particular user.

The session itself resides in the server. For each request, the client transmits the session ID in a cookie or, if the browser does not allow cookies, the server automatically writes the session ID into the URL.

iAS supports the servlet standard session interface `HttpSession` for all session activities. This interface enables you to write portable, secure servlets.

Additionally, iAS provides an additional interface `HttpSession2`, which gives support for a servlet security framework as well as sharing sessions between servlets and older iAS components (AppLogics).

Behind the scenes, there are two session styles, distributable sessions and local sessions. The main difference between them is that distributable sessions, as the name implies, can be distributed among multiple servers in a cluster, while local sessions are sticky (i.e., bound to an individual server). Sticky load balancing is automatically set for servlets of an application that is configured to use the local session model. You determine which session style to use in the application configuration file.

Sessions and Cookies

A cookie is a small collection of information that can be transmitted to a calling browser, and then retrieved on each subsequent call from that browser so that the server can recognize calls from the same client. The cookie is returned with each subsequent call to the site that created it unless it expires.

Sessions are maintained automatically by a session cookie that is sent to the client when the session is first created. The session cookie contains the session ID, which identifies the client to the browser on each successive interaction. If a client does not support or allow cookies, the server rewrites URLs such that the session ID appears in URLs from that client.

Sessions and Security

The iAS security model is based on an authenticated user session. Once a session has been created, the application user can be authenticated and “logged in” to the session. Each step of an interaction, from the servlet that receives the request to the EJB that generates content to the JSP that formats the output, can be aware that the user has been properly authenticated.

Additionally, you can specify that a session cookie only gets passed on a secured connection (like HTTPS), so the session can only remain active on a secure channel.

For more information about security, see Chapter 12, “Writing Secure Applications".

How to Use Sessions

To use a session, you first create a session using the HttpServletRequest method getSession(). Once the session is established, you can examine and set its properties using the provided methods. You can set a session to time out after being inactive for a certain time, or you can invalidate it manually.

You can also bind objects to the session, thus storing them for use by other components.

Creating or Accessing a Session

To create a new session, or to gain access to an existing session, use the HttpServletRequest method getSession(), as in the following example:

```
HttpSession mySession = request.getSession();
```

getSession() returns the valid session object associated with this request, identified in the session cookie which is encapsulated in the request object. If you call this method with no arguments, a session is created if there is not already a session associated with the request. If you call this method with a Boolean argument, then the session is created only if the argument is true.

This example shows the doPost() method from a servlet that only performs the servlet's main functions if the session is present (note that the false parameter to getSession() prevents the servlet from creating a new session if one does not already exist):

```
public void doPost (HttpServletRequest req,
                    HttpServletResponse res)
            throws ServletException, IOException
{
    if ( HttpSession session = req.getSession(false) )
    {
        // session retrieved, continue with servlet operations
    }
    else
        // no session, return an error page

    }
}
```

For more information about getSession(), see the servlet specification.

Examining Session Properties

Once a session ID has been established, you can use methods in the `HttpSession` interface to examine properties in the session, and methods in the `HttpServletRequest` interface to examine properties in the request that relate to the session.

Use the following methods to examine properties in the session:

Table 11-1 HttpSession Methods

HttpSession method	Description
`getCreationTime()`	Returns the time at which this session was created in milliseconds since January 1, 1970, 00:00:00 GMT.
`getId()`	Returns the identifier assigned to this session. An HTTP session's identifier is a unique string that is created and maintained by the server.
`getLastAccessedTime()`	Returns the last time the client sent a request carrying the identifier assigned to the session, or -1 if the session is new. Time is expressed as milliseconds since January 1, 1970, 00:00:00 GMT.
`isNew()`	Returns a Boolean value indicating whether this session is considered to be new. A session is considered to be new if it has been created by the server and not received from the client as part of this request. This means that the client has not "acknowledged" or "joined" the session and may not ever return the appropriate session identification information when it makes its next request.

For example:

```
String mySessionID = mySession.getId();
if ( mySession.isNew() ) {
    log.println(currentDate);
    log.println("client has not yet joined session " + mySessionID);
}
```

Use the following methods to inspect properties in the request object that relate to the session:

Table 11-2 HttpServletRequest Methods

HttpServletRequest method	Description
`getRemoteUser()`	Gets the name of the user making this request. This information may be provided by HTTP authentication. Returns null if there is no user name information in the request.
`getRequestedSessionId()`	Returns the session ID specified with this request. This may differ from the session ID in the current session if the session ID given by the client was invalid and a new session was created. Returns null if the request does not have a session associated with it.
`isRequestedSessionIdValid()`	Checks whether this request is associated with a session that is currently valid. If the session used by the request is not valid, it is not returned via the `getSession()` method.
`isRequestedSessionIdFromCookie()`	Returns true if the session ID for this request was provided from the client as a cookie, or false otherwise.
`isRequestedSessionIdFromURL()`	Returns true if the session ID for this request was provided from the client as part of a URL, or false otherwise.

For example:

```
if ( request.isRequestedSessionIdValid() ) {
    if ( request.isRequestedSessionIdFromCookie() ) {
        // this session is maintained in a session cookie
    }
    // any other tasks that require a valid session
} else {
    // log an application error
}
```

Binding Data to a Session

You can bind objects to sessions in order to make them persistent across multiple user interactions. The following `HttpSession` methods provide support for binding objects to the session object:

Table 11-3 HttpSession Methods

HttpSession method	Description
`getValue()`	Returns the object bound to a given name in the session, or null if there is no such binding.
`getValueNames()`	Returns an array of the names of all the values bound into this session.
`putValue()`	Binds the specified object into the session with the given name. Any existing binding with the same name is overwritten. For an object bound into the session to be distributed, it must implement the `Serializable` interface. Note that iAS RowSets and JDBC ResultSets are not serializable, and can not be distributed.
`removeValue()`	Unbinds an object in the session with the given name. If there is no object bound to the given name, this method does nothing.

Binding Notification with HttpSessionBindingListener

Some objects may require that you know when they are placed into, or removed from, a session. You can obtain this information by implementing the `HttpSessionBindingListener` interface in those objects. When your application stores data in or removes data from the session, the servlet engine checks whether the object being bound or unbound implements `HttpSessionBindingListener`. If it does, methods in the interface automatically notify the object that it has been bound or unbound.

Invalidating a Session

You can specify that the session invalidates itself automatically after being inactive for a certain amount of time. Alternatively, you can invalidate a session manually with the `HttpSession` method `invalidate()`.

Note: The session API does not provide an explicit API for logging out from a session, so any implementation of a "Logout" must call the session.invalidate() API.

Invalidating a Session Manually

To invalidate a session manually, simply call the following method:

```
session.invalidate();
```

All objects bound to the session are also removed.

Setting a Session Timeout

Session timeout is set using the ias-specific Deployment Descriptor. Refer to the session-info element in the section "Packaging for Deployment"

Controlling the Type of Session

You can control the type of session by setting the appropriate elements in the iAS-specific XML. Refer to the element session-info in Chapter 10, “Packaging for Deployment".

Sharing Sessions with AppLogics

Servlet programmers can use the iAS feature interface `HttpSession2` to share distributable sessions between AppLogics and servlets. Sharing sessions is useful when you want to migrate an application from NAS 2.x to iAS. `HttpSession2` interface adds security and direct manipulation of distributable sessions.

Additionally, if you establish a session in an AppLogic using `loginSession()` and you want to be able to access that session from a servlet, you must call the method `setSessionVisibility()` in the `AppLogic` class in order to instruct the session cookie to be transmitted to servlets as well as AppLogics. It is important to do this before calling `saveSession()`.

For example, in an AppLogic:

```
domain=".mydomain.com";
path="/"; //make entire domain visible
isSecure=true;
if ( setSessionVisibility(domain, path, isSecure) == GXE.SUCCESS )
    { // session is now visible to entire domain }
```

For more information about `setSessionVisibility()`, see the entry for the `AppLogic` class in the *Foundation Class Reference*. For more information about sharing sessions between AppLogics and servlets, see the *Migration Guide*.

Chapter 12

Writing Secure Applications

This chapter describes how to write a secure J2EE application for the iPlanet Application Server (iAS) with components that perform user authentication and authorization for access to servlets and EJB business logic.

The following sections are included in this chapter:

- iAS Security Goals
- iAS Specific Security Features
- iAS Security Model
- Overview of Security Responsibilities
- Common Security Terminology
- Container Security
- Programmatic Security
- Declarative Security
- User Authentication by Servlets
- User Authorization by Servlets
- User Authorization by EJBs
- User Authentication for Single Sign-on
- User Authentication for Rich Client
- Guide to Security Information

iAS Security Goals

In an enterprise computing environment, there are many security risks. The goal of iAS is to provide highly secure, interoperable distributed, component computing based on the J2EE security model. The security goals for iAS include:

- Full compliance with the J2EE v1.2 security model (*see the J2EE specification, v1.2 Chapter 3 Security*)
- Full compliance with the EJB v1.1 security model (*see the Enterprise JaveBean specification v1.1 Chapter 15 Security Management*). This includes EJB role based authorization.
- Full compliance with the Java Servlet v2.2 security model (*see the Java Servlet specification, v2.2 Chapter 11 Security*). This includes Servlet role based authorization.
- Support for single signon across all applications on iAS.
- Security support for Rich Clients.
- Use LDAP as the backend for security and allow administration of users during runtime.
- Implement declarative iAS specific XML based role mapping information.
- iAS specific XML files with declarative security will be created by the iAS Deployment Tool.
- Backward compatibility with the applogic security API s.

iAS Specific Security Features

iAS supports the J2EE v 1.2 security model, as well as the following features that are specific to iAS.

- Single Signon across all applications on iAS.
- Security for Rich Clients.
- iAS specific XML based role mapping information.
- A GUI based deployment tool (iASDT) used to build the XML files that contain the security information.
- Administration of users to LDAP during runtime.
- LDAP is used as the backend for security.

iAS Security Model

Secure applications require the client to first be authenticated as a valid user of the application, and secondly to have the authorization to access the EJB business logic. iAS supports security for web clients and rich clients.

Web clients use a browser and a web server to communicate using HTTP with servlets running on iAS. These clients require communication with servlets and JSPs to extend the functionality of the web server.

Applications with secure web containers and secure EJB containers may enforce the following security processes for web clients:

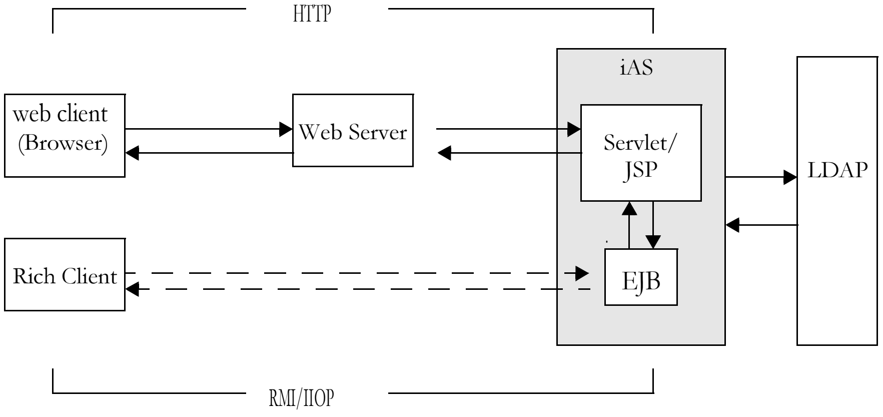
- authenticate the caller
- authorize the caller for access to the URL
- authorize the caller for access to the EJB business methods

Rich clients, on the other hand, communicate over a bridge using RMI/IIOP to directly access EJBs running on iAS. Rich clients directly invokes bean methods.

Applications with secure EJB containers may enforce the following security processes for rich clients:

- authorize the caller for access to the EJB business methods

The diagram below shows the iAS security model.



Web Client Authentication and URL Authorization

Secure web containers may have authentication and authorization properties. These containers support three types of authentication - basic, certificate and form based. When a web client requests the main application URL, the web server is responsible for collecting the user's authentication information (e.g. username and password) from the web client and passing it on to iAS.

iAS consults the security policies (derived from the deployment descriptor) associated with the web resource to determine the security roles that are permitted access to the resource. The web container tests the user's credentials against each role to decide if it can map the user to the role. The Lightweight Directory Access Protocol (LDAP) server, an enterprise wide directory service for managing information about users, groups and roles, is used to get the user's credentials.

Web Client Invocation of Enterprise Bean Methods

Once the web client has been authenticated and authorized by the web container and the JSP performs a remote method call to the EJB, the user's credentials (gathered during the authentication process) are used to establish a secure association between the JSP and the bean. A secure EJB container has a deployment descriptor with authorization properties that are used to enforce access control on the bean method. The EJB container uses role information received from the LDAP server to decide whether it can map the caller to the role and allow access to the bean method.

Rich Client Invocation of Enterprise Bean Methods

For rich clients a secure EJB container will consult with it's security policies to determine if the caller has the authority to access the bean method. This process is the same for rich clients and web clients.

Overview of Security Responsibilities

A primary goal of the J2EE platform is to isolate the developer from the details of the security mechanisms and facilitate the secure deployment of an application in diverse environments. This goal is address by providing mechanisms for the specification of application security requirements declaratively and outside the application.

Application Developer

Programmatic security is supplied by the developer.

- Specifies security levels.
- Verifies security permission levels when secure operations are being accessed.

Application Assembler

The application assembler or application component provider must identify all the security dependencies embedded in a component including:

- The names of all the role names used by the components to call isCallerInRole or isUserInRole.
- References to all the external resources accessed by the components.
- References to all inter-component calls made by the component.
- Recommended that the assembler identifies which method calls of which components feature parameters and return values which should be protected for confidentiality and or integrity. The deployment descriptor is used for this purpose (9.4.3)

Application Deployer

The deployer takes the security view of the components, provided by the assembler, and uses them to secure in the application a particular enterprise environment. The deployer uses the iASDT to map the view provided by the assembler to the policies and mechanisms that are specific to the operational environment. The security mechanisms configured by the deployer are implemented by the containers on behalf of the components which are hosted in the containers.

- Assigns groups of users to security levels
- Refines the privileges required to access the methods of components, and to define the correspondence between the security attributes presented by the callers and the container privileges.

Common Security Terminology

Authentication

Authentication is the process that verifies that the user is who the user claims to be. For example, the user may enter her username and password in a Web browser, and if those credentials match the her permanent profile stored in the LDAP server then she is authenticated. She is now associated with a security identity for the remainder of the session.

Authorization

Authorization is the process of permitting a user to perform desired operations, after he has been authenticated. For instance, a human resources application may authorize managers to view personal employee information for all employees, but allow employees to only view their own personal information.

Role Mapping

A client may be defined in terms of a security role. For example, a company might use its employee database to generate both a company-wide phone book application, and to generate payroll information. Obviously, while all employees might have access to phone numbers and email addresses, only some employees would have access to salary information. Employees with the right to view or change salaries might be defined as having a special security role.

A role is different from a user group in that a role defines a function in an application, while a group is a set of users that are related in some way. For example, members of the groups `astronauts`, `scientists`, and (occasionally) `politicians` all fit into the role of `SpaceShuttlePassenger`.

The EJB security model describes roles (as distinguished from user groups) as being described by an application developer and independent of any particular domain. Groups, by contrast, are specific to a deployment domain. The role of the deployer is to map roles into one or more groups.

In iAS, roles correspond to user groups configured in the Directory Server. LDAP groups can contain both users and other groups. Future versions of iAS will provide specific support for roles.

Container Security

The component containers are responsible for providing security for the J2EE application. There are two forms of security provided by the container:

- Programmatic security
- Declarative security

Programmatic Security

Programmatic security is when an EJB or servlet uses method calls to the security API, specified by the J2EE security model, to make business logic decisions based on the security role of the caller or remote user. Programmatic security should only be implemented when declarative security alone is insufficient in meeting the needs of the application's security model.

The J2EE Specification, v1.2 defines programmatic security as consisting of two methods of the EJB EJBContext interface and two methods of the servlet HttpServletRequest interface. iAS supports these interfaces as specified by this specification. For further details on programmatic security see *section 3.3.6 Programmatic Security*, *of the J2EE Specification,v1.2* .

Declarative Security

Declarative security is when the security mechanism for an application is declared and handled externally to the application. Deployment descriptors (DDs) are used by iAS to describe the J2EE application's security structure, including security roles, access control, and authentication requirements.

The deployment descriptors for security aware applications, web-app containers, and EJB containers, have security elements in the form of XML tags to express the security characteristics of the application. Security characteristics include authentication and authorization.

iAS supports the Document Type Descriptors (DTDs) specified by J2EE v1.2, and has additional security elements included in the iAS deployment descriptors.

Declarative security is the responsibility of the application deployer. The XML DDs are generated by the iAS Deployment Tool. See the *Administration and Deployment Guide* for details.

Application Level Security

The application XML DD contains authorization descriptors for all the roles a user can have when accessing the application's servlets and EJBs. On the application level, all roles used by any of the application's containers must be listed in this file. These roles are described by the role-name element in the application's XML DD file. These role names are scoped to the EJB XML DDs (ejb-jar files) and to the servlet XML DDs (web-war files).

Servlet Level Security

A secure web container needs to authenticate web client users and to authorize their access to the servlet. Once the user has been authenticated and authorized the servlet passes on user credentials to an EJB to establish a secure association with the bean.

EJB Level Security

The EJB container is responsible for authorizing access to a bean method by using the security policy laid out in the EJB XML deployment descriptor.

User Authentication by Servlets

The three web based login mechanisms required by J2EE Specification, v1.2 are supported by iAS. These three mechanisms include: HTTP basic authentication, SSL mutual authentication, and form-based login.

The web application deployment descriptor `login-config` element, describes the authentication method to be used, the realm name to be used for this application by the HTTP basic authentication, and the attributes that are needed by the form login mechanism.

The syntax for the `login-config` element is as follows:

<!ELEMENT login-config (auth-method?,realm-name?,from-login-config?)>

For further details regarding web application deployment descriptor elements, refer to *Chapter 13 Deployment Descriptor of the Java Servlet Specification, v2.2.*

HTTP Basic Authentication

HTTP basic authentication (RFC2068) is supported by iAS. HTTP basic authentication protocol indicates the HTTP realm for which access is being negotiated. Because passwords are sent with base64 encoding, this type of authentication is not very secure.

SSL Mutual Authentication

Secure Socket Layer (SSL) 3.0 and the means to perform mutual (client & server) certificate based authentication is a requirement of the J2EE Specification, v1.2. This security mechanism provides end user authentication using HTTPS (HTTP over SSL).

The iAS SSL mutual authentication mechanism (also known as HTTPS authentication) supports the following cipher suites:

```
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
```

Form Based Login

The look and feel of the login screen cannot be controlled with the HTTP browser's built in mechanisms. J2EE introduces the ability to package a standard HTML or Servlet/JSP based form for logging in. The login form is associated with a web protection domain (an HTTP realm) and is used to authenticate previously unauthenticated users.

In order for the authentication to proceed appropriately, the action of the login form must always be "`j_security_check`".

The following is an HTML sample showing how the form should be coded into the HTML pages:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_user_name">
<input type="password" name="j_password">
</form>
```

User Authorization by Servlets

Servlets can be configured to only permit access to user's with the appropriate level of authorization. This is done by using the iAS Deployment Tool to generate deployment descriptors (DD) for the application Ear file and servlet War files.

Defining Roles

All role names for the entire application are declared in the application XML DD. The *`security-role`* and *`role-name`* elements in the application XML DD will declare all role names permitted by this application.These security roles are scoped to the J2EE web application deployment descriptor.

The *`security-role`* element is a sub element of the `application` element in the application XML DD. The syntax for the *`security-role`* element is as follows:

```
<!--

The security-role element defines a security role which is global to
the application. There are two sub elements; the first is a
description of the security role, and the second is the name of the
security role.

<!ELEMENT security-role (description?, role-name)>

The role-name element contains the name of a role.

<!ELEMENT role-name (#PCDATA)>
```

Referencing Security Roles

For each servlet, the web application DD will declare all roles authorized to have access to it. The *`security-rol-ref`* and *`role-link`* elements in the web-app XML DD will link the authorized roles to the role name on the application level.

The application assembler is responsible for linking all the security role references declared in the security-role-ref elements to the security roles defined in the security-role elements.

The application assembler links each security role reference to a security role using the role-link element. The value of role-link element must be the name of one of the security roles defined in a security-role element.

The following deployment descriptor example shows how to link the Sudety role reference named payroll to the security role named payroll-department.

```
<!ELEMENT security-role-ref (description?, role-name, role-link)>
```

```
<!ELEMENT role-link (#PCDATA)>
```

Defining Method Permissions

On the servlet level define method permissions using the *`auth-constraint`* element of the web-app XML DD.

The auth-constraint element on the resource collection must be used to indicate the user roles that should be permitted to this resource collection. The role used here must appear in a security-role-ref element.

```
<!ELEMENT auth-constraint (description?, role-name*)>
```

Sample web application DD

The security section of a sample web application deployment descriptor might look as follows:

```
<web-app>

  <display-name>A Secure Application</display-name>

  <security-role>
    <role-name>manager</role-name>
  </security-role>

  <servlet>
    <servlet-name>catalog</servlet-name>
    <servlet-class>com.mycorp.CatalogServlet</servlet-class>

    <init-param>
      <param-name>catalog</param-name>
      <param-value>Spring</param-value>
    </init-param>

    <security-role-ref>
      <role-name>MGR</role-name> <!-- role name used in code -->
      <role-link>manager</role-link>
    </security-role-ref>
  </servlet>

  <servlet-mapping>
    <servlet-name>catalog</servlet-name>
    <url-pattern>/catalog/*</url-pattern>
  </servlet-mapping>

  <web-resource-collection>
    <web-resource-name>SalesInfo</web-resource-name>
    <urlpattern>/salesinfo/*</urlpattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>

    <user-data-constraint>
      <transport-guarantee>SECURE</transport-guarantee>
    </user-data-constraint>

    <auth-constraint>
      <role-name>manager</role-name>
    </auth-constraint>
  </web-resource-collection>

</web-app>
```

User Authorization by EJBs

EJBs can be configured to only permit access to user’s with the appropriate level of authorization. This is done by using the iAS Deployment Tool to generate deployment descriptors (DD) for the application Ear file and EJB jar files.

Defining roles

The application assembler can define one or more roles in the deployment descriptor. The application assembler then assigns groups of methods of the enterprise bean's home and remote interfaces to the security roles to define the security view of the application.

The deployer assigns the user groups and user accounts defined in the operational environment to security roles defines by the application assembler.

The application assembler is responsible for the following:

- Define each security role using a security-role element
- Use the role-name element to define the name of the security role
- Optionally use the description element to provide a description of a security role

The Security roles defined by the security-role elements are scoped to the ejb-jar file level and apply to all the enterprise beans in the ejb-jar files. (The J2EE specification does not say anyway to define global roles, which are global to container).

The following is an example of security role definition in a deployment descriptor:

...

```
<assembly-descriptor>

    <security-role>

        <description>

             This role includes the employees of the enterprise who
             are allowed to access the employee self service
             application. This role is allowed to access only
             her/his information
        </desciption>
        <role-name>employee<role-name>
        </security-role>
        <security-role>
             <description>
                 This role should be assigned to the personnel
                 authorized to perform administrative functions
                 for the employee self service application. This
                 role does not have direct access to
                 sensitive employee and payroll information
             </desciption>
             <role-name>admin<role-name>
        <security-role>
```

... <assembly-descriptor>

Done by iASDT. Need to provide APIs for adding roles and role members.

Defining method permissions

The application assembler defines the method permissions relation in the deployment descriptor using the method permission elements as follows:

Each `method-permission` element includes a list of one or more security roles and a list of one or more methods. All the listed security roles are allowed to invoke all the listed methods. Each security role in the list is identified by the `role-name` element, and each method (or a set of methods, as described below) is identified by the method element. An optional description can be associated with a method-permission element using the description element.

The method permissions relation is defines as the union of the all method permissions defined in the individual method permission elements.

A security role or a method may appear in multiple `method-permission` elements.

The following example illustrates how security roles are assigned method permissions in the deployment descriptor.

```
...
<method-permission>
    <role-name>employee</role-name>
    <method>
        <ejb-name>EmployeeService</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>

<method-permission>
    <role-name>employee</role-name>
    <method>
         <ejb-name>AardvarkPayroll</ejb-name>
         <method-name>findByPrimaryKey</method-name>
    </method>
    <method>
         <ejb-name>AardvarkPayroll</ejb-name>
         <method-name>getEmployeeInfo</method-name>
    </method>
    <method>
         <ejb-name>AardvarkPayroll</ejb-name>
         <method-name>updateEmployeeInfo</method-name>
```

```
   </method
</method-permission>

...
```

No interaction here, iASDT will convert these into security elements.

Security role references:

The Bean provider is responsible for declaring in the security-rol-ref elements of the deployment descriptor all the security role names used in the enterprise bean code.

Application assembler is responsible for linking all the security role references declared in the security-role-ref elements to the security roles defined in the security-role elements.

The application assembler links each security role reference to a security role using the role-link element.

The value of role-link element must be the name of one of the security roles defined in a security-role element.

The following deployment descriptor example shows how to link the Sudety role reference named payroll to the security role named payroll-department.

```
...
<enterprise-beans>
            ...
   <entity>
      <ejb-name>AardvarkPayroll</ejb-name>
         <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
         ...
         <security-role-ref>
             <description>
```

This role should be assigned to the employees of the payroll department. Members of this role have access to anyone’s payroll record. The role has been linked to the payroll-department role.

```
                    </description>
               </security-role-ref>
          ....
          </entity>
          ...
</enterprise-bean>
```

Realm

An application can specify that it wants to use for authentication and authorization.

User Authentication for Single Sign-on

The single sign-on across applications on iAS is supported by the iAS servlets and JSPs. This feature allows multiple applications that require the same user sign-on information, to share this information between them, rather than have the user sign-on separately for each application. These applications are created to authenticate the user one time, and when needed this authentication information is propagated to all other involved applications.

An example application using the single sign-on scenario, could be a consolidated airline booking service that searches all airlines and provides links to different airline web sites. Once the user signs on to the consolidated booking service, her user information can be used by each of the individual airline sites without requiring her to sign on again.

How to Configure for Single Sign-on

The iAS specific deployment descriptor (DD) for the web container has an element, called session-info, that has fields to specify the authentication for the servlets and JSPs within the container. The DD is created by the iAS Deployment Tool (iASDT). This section concentrates on how the security fields for the session-info element in the DD work together to perform the single sign-on authentication. For details on how to create an iAS specific web container DD, see the *Administration and Deployment Guide*. For a complete description of all the session-info fields, refer to Chapter 10, “Packaging for Deployment".

The session-info element has the following fields that are involved with the authentication process:

Table 12-1 Security Fields for single sign-on

domain	This field specifies the domain to which the cookie is sent back from the browser. By default (i.e. if the user does not specify a domain), the domain of the URL that sets the cookie is assumed to be the domain that the cookie is sent back to. The user can set the domain to any domain that he wishes the cookie to be sent back to. The domain must have at least two periods, and sometimes may have three (e.g. ".acme.com" or ".acme.co.in").
path	This field specifies the path for the session cookie; this is the minimum path the URL must have for the cookie to be sent back from the browser. For example, setting the path to "/phoenix" will send the cookie back when either of the following URLs is accessed: http://my.foo.com/phoenix/birds.html or http://my.foo.com/phoenix/bees.html The path must begin with a "/". If the path is not set, the default path is assumed to be that of the URL setting the cookie.
scope	This specifies a grouping name that "associates" applications sharing the same user session; i.e. signing on into one application automatically allows the user to access the other application without signing on to it. These grouped applications should have the same value for this scope field in their respective iAS specific web XML DD files.

Single Sign-on Example

Consider two applications hosted on iAS named AirlineSearch and AirlineBooking. Both are part of the myairlines.com domain and require the user to be authenticated to access resources within these two applications. AirlineSearch allows the user to search on different airlines available in her part of the country, and AirlineBooking allows her to make bookings using her special preferences such as for seating among other things.

The ias-web.xml for both AirlineService and AirlineBooking will contain the following:

```
<session-info>
   <path>/iASApp</path>
   <scope>AirlineSignon</scope>
</session-info>
```

Now the user first accesses the services provided by the AirlineService application by saying:

```
http://www.myairlines.com/iASApp/AirlineService/showFlights
```

showFlights could be a servlet that shows all flights at the time the user requested. This requires the user to log in. Once the user has seen all the flights, she decides to book tickets and accesses:

```
http://www.myairlines.com/iASApp/AirlineService/bookFlights
```

This provides the service to book flights based on the user’s preferences, which could already be available from the previous accesses and from the sign-on information provided to the previous AirlineService application.

Since both these applications are within the same domain, the domain field is not set in this example. But it is easy to see how this can be extended to share sign-on information between multiple domains.

User Authentication for Rich Client

Security on the Rich Client path is integrated with iAS' security infrastructure. The CXS uses iAS' security manager to authenticate clients with user information stored in LDAP. Client credentials are passed from the client, through the bridge to EJBs. A client side callback initiates client login (with username and password).

The type of the object to be instantiated to obtain this information is specified through an environment setting on the client. In case of authentication failure, the client-side is setup to retry the login process. The number of retries is currently hard-coded to three.

For more information on elements in the Rich Client deployment descriptor see section “Rich Client XML DTD,” in Chapter 10, "Packaging for Deployment".

Guide to Security Information

Each of the following types of information is shown with a short description, the location where the information resides, how to create the information, how to access the information, and where to look for further information.

- User Information
- Security Roles

User Information

User name, password, etc.

Location:
Directory Server

How to Create:
Create using Mission Console, or programmatically using the LDAP SDK. See the *Administration and Deployment Guide* for more details.

Security Roles

Role that defines a function in an application, made up of a number of users and/or groups. LDAP groups function as roles in iAS.

Location:
Directory Server

How to Create:
Use the iAS Deployment Tool (iASDT).

How To Access:
Test for a user’s membership in a role using `isCallerInRole()`.

Chapter 13

Taking Advantage of iAS Features

This chapter describes how to implement iAS features in your application.

iAS provides many additional features to augment your servlets for use in a iAS environment. These features are not a part of the official servlet specification, though some, like the servlet security paradigm described in Chapter 12, “Writing Secure Applications" are based on emerging Sun standards and will conform to those standards in the future.

This chapter contains the following sections:

- Accessing the Servlet Engine
- Caching Servlet Results
- Using Application Events
- Sending and Receiving Email from iAS
- iAS Application Builder Features

Accessing the Servlet Engine

The servlet engine controls all servlet functions, including instantiation, destruction, service methods, request and response object management, and input and output. The servlet engine in iAS is a special class called an AppLogic. AppLogics are iAS components that interact with the core server. In previous releases of iAS, AppLogics were a part of the application model, though for current and future releases they are solely available to access iAS internal features.

Each servlet is scoped in an AppLogic. You can access the AppLogic instance controlling your servlet using the method `getAppLogic()` in the iAS feature interface `HttpServletRequest2`. When you do this, you also gain access to server context. These activities are necessary to take advantage of other iAS features, as described in the other sections in this chapter.

Accessing the Servlet's AppLogic

To access the controlling AppLogic, cast the request object as an `HttpServletRequest2`. This interface provides access to the AppLogic via the method `getAppLogic()`, which returns a handle to the superclass.

The following example servlet header shows how to access the AppLogic instance:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.kivasoft.applogic.*;
import com.kivasoft.types.*;
import com.netscape.server.servlet.extension.*;

public class AppLogicTest extends HttpServlet {

    public void service(HttpServletRequest req,
                        HttpServletResponse res)
        throws ServletException, IOException
    {
        HttpServletRequest2 req2 = (HttpServletRequest2)req;
        AppLogic al = req2.getAppLogic();
        //al is now a handle to the superclass
...
```

Accessing the Server Context

Some iAS features, such as Application Events (see “Using Application Events” on page 255), require an `IContext` object. `IContext` defines a view of the server context. For more information about `IContext`, see the entry for the `IContext` interface in the *iAS Foundation Class Reference*.

To obtain an `IContext` from a servlet, the standard servlet context can be cast to `IServerContext`, and from there, a `com.kivasoft.IContext` instance can be obtained, as in the following example:

```
ServletContext ctx = getServletContext();
com.netscape.server.IServerContext sc;
sc = (com.netscape.server.IServerContext) ctx;
com.kivasoft.IContext kivaContext = sc.getContext();
```

Alternatively, you can access the underlying AppLogic instance from a servlet, as described in “Accessing the Servlet’s AppLogic” on page 252, and obtain the context from the AppLogic’s context member variable, as in the following example:

```
HttpServletRequest2 req2 = (HttpServletRequest2)req;
AppLogic al = req2.getAppLogic();
com.kivasoft.IContext kivaContext = al.context;
```

From an EJB, the standard `javax.ejb.SessionContext` or `javax.ejb.EntityContext` can be cast to `IServerContext`, and from there, a `com.kivasoft.IContext` instance can be obtained, as in the following example:

```
javax.ejb.SessionContext m_ctx;
....
com.netscape.server.IServerContext sc;
sc = (com.netscape.server.IServerContext) m_ctx;  /
com.kivasoft.IContext kivaContext;
kivaContext = sc.getContext();
```

Caching Servlet Results

iAS has the ability to cache the results of a servlet in order to make subsequent calls to the same servlet faster. iAS caches the results of a request (i.e. a servlet’s execution) for a specific amount of time, so that if another call for that data happens, iAS can just return the cached data rather than having to perform the operation again. For example, if your servlet returns a stock quote that you only want to update every 5 minutes, you could set the cache to expire after 300 seconds.

Whether to cache results, and how to cache them, depends on the type of data involved. It makes no sense to cache the results of a quiz submission, for example, because the input to the servlet is different each time. However, you could cache a high-level report showing demographic data taken from quiz results and updated once an hour.

You can define how an iAS servlet handles memory caching by editing specific fields in the servlet’s configuration file. In this way, you can create programmatically standard servlets that still take advantage of this valuable iAS feature. For more information on servlet configuration files, see Chapter 10, “Packaging for Deployment ".

Set the following variables in the `ServletData` section of the servlet's configuration file:

Table 13-1 Servlet Cache Settings

Name	Type	Value
`CacheTimeOut`	Int	Optional. Elapsed time (in seconds) before the memory cache for this servlet is released.
`CacheSize`	Int	Optional. Size (in kb) of the memory cache for this servlet.
`CacheCriteria`	Str	Optional criteria expression containing a string of comma-delimited descriptors. Each descriptor defines a match with one of the input parameters to the servlet.

You use the `CacheCriteria` field to set criteria used to determine whether the results of a servlet should be cached. The `CacheCriteria` field contains a test for one or more fields in the request. This allows you to conditionally cache results based on the value or presence of one or more form fields. If the tests succeed, the servlet results are cached.

Use the follow syntax for the `CacheCriteria` field:

Table 13-2 CacheCriteria Field

Syntax	Description
arg	Test succeeds for any value of *arg* in the input parameter list. For example, if the field is set to `"EmployeeCode"`, the results are cached if the request contains a field called `EmployeeCode`.
arg=v	Test whether *arg* matches *v* (a string or numeric expression). For example, if the field is set to `"stock=NSCP"`, the results are cached if the request contains a field called `stock` that has the value `NSCP`. Assign an asterisk (`*`) to the argument to cache a new set of results every time the servlet runs with a different value. For example, if the criteria is set to `"EmployeeCode=*"`, the results are cached if the request object contains a field called `EmployeeCode` and the value is different from the value used to create the currently cached result.
arg=v1 | v2	Test whether *arg* matches any values in the list (*v1*, *v2*, and so on). For example: `"dept=sales|marketing|support"`

Table 13-2 CacheCriteria Field

Syntax	Description
arg=n1-n2	Test whether *arg* is a number that falls within the given range. For example: `"salary=40000-60000"`
Syntax	Description

Using Application Events

In an iAS environment, you can create and use named events. The term *event* is widely used to refer to user actions, such as mouse clicks, that trigger code. However, the events described in this section are not caused by users. Rather, an event is a named action that you register with the iAS. The event occurs either when a timer expires or when the event is activated from application code at run time.

Events are stored persistently in the iAS, and are removed only when your application explicitly deletes them. Typical uses for events include periodic backups, reconciling accounts at the end of the business day, or sending alert messages. For example, you can set up an event that sends an email to alert your company's buyer when inventory levels drop below a certain level.

Each event has a name, a timer (optional), and one or more actions to take when the event is triggered. Application events have the following characteristics:

- Each event can cause the execution of one or more actions, which can include sending email or calling another application component.
- Actions can be synchronous or asynchronous with the calling environment.
- Multiple actions can be configured to execute concurrently with one another, or serially, one after the other.
- Multiple actions are executed in a specific order (the order in which they are registered).
- Request data can be passed to an application event in an `IValList` object.

You can set up events to occur at specific times or at intervals, such as every hour or once a week. You can also trigger an event by calling the event by name from code. When an event's timer goes off or it is called from code, the associated action occurs.

The Application Events API

iAS uses two interfaces to support events:

- The `IAppEventMgr` interface manages application events. This interface defines methods for creating, registering, triggering, enabling, disabling, enumerating, and deleting events.
- The `IAppEventObj` interface represents the defined events an application supports. This interface defines methods not only for getting or setting attributes of an event, but also for adding, deleting, or enumerating actions of the event.

These two interfaces are described in the following sections.

The IAppEventMgr Interface

You can perform any of the following administrative tasks with an event by using the associated methods in the `IAppEventMgr` interface:

Table 13-3

Method	Description
createEvent()	Creates an empty application event object.
deleteEvent()	Removes a registered event from iAS.
disableEvent()	Temporarily disables a registered event.
enableEvent()	Enables a registered event.
enumEvents()	Enumerates through the list of registered events.
getEvent()	Retrieves the `IAppEventObj` for a registered event.
registerEvent()	Registers a named event for use in applications.
triggerEvent()	Triggers a registered event.

The IAppEventObj Interface

The event's behavior is determined by its attributes, which define how and when an event executes, and its actions, which define what the event does when it is triggered. To set and examine attributes and actions, use the methods in the `IAppEventObj` interface:

Table 13-4IAppEventObj Methods

Method	Description
addAction()	Appends an action to an ordered list of actions.
deleteActions()	Deletes all actions added to this `IAppEventObj`.
enumActions()	Enumerates the actions added to this `IAppEventObj`.
getAttributes()	Retrieves the list of attributes of an `IAppEventObj`.
getName()	Retrieves the name of the `IAppEventObj`.
setAttributes()	Sets a list of attribute values for the `IAppEventObj`.
Method	Description
addAction()	Appends an action to an ordered list of actions.

Creating a New Application Event

Follow these steps to create a new application event and register it with iAS. For more information about the interfaces and methods described in each step, see the entries for `IAppEventMgr` and `IAppEventObj` in the *iAS Foundation Class Reference*.

NOTE

A `com.kivasoft.IContext` object is required to create an event. For more information, see "Accessing the Server Context" on page 252.

1. You first need the context parameter, an `IContext` object which provides access to iAS services. Obtain the context parameter from the server context `IServerContext`, as in the following example:

```
ServletContext ctx = getServletContext();
com.netscape.server.IServerContext sc;
sc = (com.netscape.server.IServerContext) ctx;
com.kivasoft.IContext ic = sc.getContext();
```

2. Next, use the `GetAppEventMgr()` method in the `GXContext` class to create an `IAppEventMgr` object:

```
IAppEventMgr mgr = com.kivasoft.dlm.GXContext.GetAppEventMgr(ic);
```

3. After creating the `IAppEventMgr` object, you can create an application event—an instance of `IAppEventObj`—by calling `createEvent()` on the `IAppEventMgr` object, as in the following example:

```
IAppEventObj evtObj = mgr.createEvent("myEvent");
```

4. You now have an empty event in the system. Create an `IValList` object to hold the event's attributes, and then set the attributes to the requirements you have for this event. Finally, assign the attributes to your event. For example, this event executes at 5:00 AM every morning:

```
IValList atts = GX.CreateValList();
atts.setValString(GXConstants.GX_AE2_RE_KEY_TIME, "5:0:0 */*/*");
evtObj.setAttributes(atts);
```

5. You can assign actions to the event in order to cause it to do something when it is triggered. This procedure is similar to creating event attributes; first create an IValList object to contain the actions, then assign them to the event. For example, this event runs a servlet called `myServlet`:

```
IValList action = GX.CreateValList();
action.setValString(GXConstants.GX_AE2_RE_KEY_SERVLET,
"myServlet");
evtObj.addAction(action);
```

6. You must then register the event, or make iAS aware of it, by calling `registerEvent()`. Further, you must also instruct iAS to enable the event for access by calling `enableEvent()`. The following example shows the registration and enabling of an event:

```
if (mgr.registerEvent("myEvent", evtObj) != GXE.SUCCESS)
    return streamResult("Cannot register RepGenEvent<br>");
```

7. Once the event is registered and enabled, you can trigger it by hand if you want. To trigger the event, use the `IAppEventMgr` method `triggerEvent()`, as in the following example:

```
mgr.triggerEvent("myEvent");
```

Sending and Receiving Email from iAS

iAS supports email transactions through the `IMailbox` interface. For more information, see the entry for the `IMailbox` interface in the *iAS Foundation Class Reference*.

In order for email applications to work, you must have access to an SMTP server if you want to send email and a POP server if you want to receive email.

Security in Email

Security is often a concern when sending or receiving email. If the application generates and sends email using user input to set the address or content, then there is a risk of propagating inappropriate messages or mailing to incorrect recipients. Be sure to validate all user input before incorporating it in email.

Accessing the Controlling AppLogic

Using the `IMailbox` interface from a servlet requires access to the AppLogic instance that controls the servlet's functions. For more information, see "Accessing the Servlet Engine" on page 251.

For example, before you can use the `IMailbox` interface, create a handle to the AppLogic instance that controls your servlet:

```
HttpServletRequest2 req2 = (HttpServletRequest2)req;
AppLogic al = req2.getAppLogic();
```

The examples in this section assume that the above code exists in your servlet.

Receiving Email

To receive email, your application must have access to a POP server.

Before retrieving messages, you can use `retrieveCount()` to see how many messages are waiting in the specified inbox on the mail server. By checking first, you can avoid attempting to retrieve messages if the mailbox is empty. You can also use this technique when you need to know how many messages are waiting in order to construct a loop that iterates through them one by one.

To retrieve messages, call `retrieve()`. Depending on the parameters you pass to this method, you can customize the retrieval process in the following ways:

- Retrieve all messages
- Retrieve only unread messages
- Delete messages from the mail server as they are retrieved

Only those messages received before the last call to `open()` are retrieved. You can not open a mailbox session, leave it open, and continuously receive email messages. Instead, you must open a new session each time you want to retrieve new email.

After retrieving messages, you can return the mailbox to its original state by calling `retrieveReset()`. This method undeletes and unmarks any messages that were affected by the previous `retrieve()` call.

To receive email

1. Create an instance of `IMailbox` by calling `createMailbox()` from the servlet's controlling AppLogic instance ("Accessing the Controlling AppLogic" on page 259). In this call, you specify valid user information and the name of the POP server you want to access. For example:

```
IMailbox mb;

mb = al.createMailbox("mail.myOrg.com","myUserName",

"pass7878","sid@blm.org");
```

2. Open a session on your POP server by calling `open()` with the `OPEN_RECV` flag. For example:

3. result = mb.open(GX_MBOX_OPEN_FLAG.OPEN_RECV);

4. To find out whether you have messages, call `retrieveCount()`. For example:

```
int mbCount = mb.retrieveCount();
```

5. To retrieve messages, instantiate an `IValList` object to contain the email messages, then call `retrieve()`. For example, the following code retrieves the latest unread messages and does not delete them from the mailbox:

```
IValList messages = GX.CreateValList();
messages = mb.retrieve(true, false);
```

   Only the messages received before the call to `open()` are retrieved.

6. To undo changes, call `retrieveReset()`. For example:

```
 result = mb.retrieveReset();
```

7. To close the session, call `close()`. For example:

```
mb.close();
```

You can have only one mail server session open at a time. For example, suppose you open a session with the `OPEN_RECV` flag, then want to send email. You must first close the existing session, then open another one with the `OPEN_SEND` flag.

Example

The following code retrieves email in a servlet:

```
// Create mailbox object to connect to a POP mail server
IMailbox recvMB;
public void recvMail()
{
   // Only check messages received after the last open
   boolean Latest = true;
   // Remove retrieved messages from the mail server
   boolean Delete = true;

   // Create an IMailbox instance
   HttpServletRequest2 req2 = (HttpServletRequest2)req;
   AppLogic al = req2.getAppLogic();
   IMailbox recvMB;
   recvMB = al.createMailbox(recvhost, user, pswd, useraddr);

   if (recvMB != null)
   {
      if (recvMB.open(GX_MBOX_OPEN_FLAG.OPEN_RECV))
      {
         // Count the number of new messages
         int numMsgs = recvMB.retrieveCount();
         if(numMsgs > 0)
         {
            IValList mesgList;
            // Retrieve the new messages
            mesgList = recvMB.retrieve(Latest,Delete);

            // Use IValList methods to iterate through
            // the returned IValList. The keys in the
            // IValList are the message numbers. The
            // values are the email messages as strings
         }
         recvMB.close();
      }
   }
}
```

Sending Email

To send email, your application must have access to an SMTP server. Construct the email address and message separately, then use the `send()` method to send the email out through the server.

You can send email to a single recipient or to a group of recipients. To send email to a group, use one of the following techniques:

- Pass the email addresses to `send()` as an array.
- Use a loop to send a series of messages one at a time.

You can populate an address array dynamically using the results of a query, in which each row returned by the query is one email address. Use a loop to iterate through the rows in the query's result set and assign the data to successive elements of the array.

To send email

1. Create an instance of `IMailbox` by calling `createMailbox()` from the servlet's controlling AppLogic instance ("Accessing the Controlling AppLogic" on page 259). In this call, you specify valid user information and the name of the POP server you want to access. For example:

   ```
   IMailbox mb;
   mb = al.createMailbox("mail.myOrg.com",
                         "myUserName",
                         "pass7878",
                         "sid@blm.org");
   ```

2. Open a session on your SMTP server by calling `open()` with the `OPEN_SEND` flag. For example:

   ```
   int result = mb.open(GX_MBOX_OPEN_FLAG.OPEN_SEND);
   ```

3. To send the message, call `send()`. Pass a single email address or an array of addresses to this method, along with the text of the message. For example:

   ```
   java.lang.String[] ppTo = {"sal@dat.com","sid@blm.com",null};
   ```

   ```
   int mbSend = mb.send(ppTo,"Testing email");
   ```

4. To close the session, call `close()`. For example:

   ```
   mb.close();
   ```

You can have only one mail server session open at a time. For example, suppose you open a session with the `OPEN_SEND` flag, then want to retrieve your email. You must first close the existing session, then open another one with the `OPEN_RECV` flag.

Example

The following code sends email in a servlet:

```
// Define the string parameters that will be passed
// to IMailbox methods
String sendhost = "smtp.kivasoft.com";
String recvhost = "pop.kivasoft.com";
```

```
String user = "eugene";
String pswd = "eugenesSecretPassword";
String useraddr = "eugene@kivasoft.com";
String sendTo[] = {"friend@otherhost.net", null};
String mesg = "Hi Friend, How are you?";
public void sendMail()
{
   // Create an IMailbox instance
   HttpServletRequest2 req2 = (HttpServletRequest2)req;
   AppLogic al = req2.getAppLogic();
   IMailbox sendMB;
   sendMB = al.createMailbox(sendhost, user, pswd, useraddr);

   if (sendMB != null) // sendMB successfully created
   {
      // Open a session with the mail server
      if (sendMB.open(GX_MBOX_OPEN_FLAG.OPEN_SEND))
      {
         // Send a mail message
         sendMB.send(sendTo,mesg);
         // Close the mailbox session
         sendMB.close();
      }
   }
}
```

iAS Application Builder Features

iAS includes APIs that were designed for use by the code generated from iAS Application Builder (iAB) wizards. These APIs are available to servlet programmers, although we recommend that you use iAB to create servlets that use these features.

The features presented here include:

- Validating Form Field Data
- Creating Named Form Action Handlers
- Example Validation and Form Action Handler

Validating Form Field Data

You can set a servlet to automatically check form fields for certain types of values. Additionally, you can create a named error handler to control application flow if the validation fails.

In short, you specify the rules for validation in a servlet's configuration file, and then call the `HttpServletRequest2` method `validate()` to test the fields against the validation rules. If validation fails, you can let iAS generate an error page automatically, or you can provide an error handler method in your servlet to produce an error message.

Validation Methods

iAS provides a method called `validate()` that validates all the form fields configured in the servlet's configuration file. This method is defined in the iAS feature interface `HttpServletRequest2`. To use this interface, cast the standard request object to it, as in the following example:

```
public void service (HttpServletRequest req,
                     HttpServletResponse res)
throws ServletException, IOException
{
    HttpServletRequest2 req2 = (HttpServletRequest2) req;
```

```
There are two method signatures for validate(), as shown here:
```

```
public boolean validate(HttpServletResponse response)
               throws IOException;
```

```
public boolean validate(HttpServletResponse response,
                        HttpServlet servlet,
                        String errHandlerName)
               throws ServletException, IOException;
```

In the first form, if a validation error occurs, the system automatically generates an error response page. In the second form, you pass a servlet and method name in the `servlet` and `errHandlerName` parameters, respectively, that correspond to a named error handler that you write. (See "Error Handlers" on page 266)

`validate()` returns true if the input matches the validation rule, or false otherwise.

Validation Rules

Input data are validated based on the rules configured for each form field in the servlet's configuration file, in the `Parameters` section. This section resides in the `ServletData` section. The `Parameters` section enables you to specify the following information for each form field:

Table 13-5 Servlet Form Fields

Name	Type	Value
`inputRequired`	Str	Optional, indicates whether an input value must be supplied for that parameter. Valid values are `y` or `n`, the default is n.
`inputRule`	Str	Optional, specifies how the named parameter should be validated. `inputRule` indicates the type of data the value should be validated against.

You can also specify a flag `inputRequired` (set to `y` or `n`) for each parameter. This flag indicates whether the specified value must exist in the input stream. If the variable is not present, validation fails.

For more information about the `Parameters` section in servlet configuration files, see "Web Application XML DTD" in Chapter 10, "Packaging for Deployment".

The following types of data can be checked:

Table 13-6 Data to Validate

Data to Validate	Validation Rule	Description
Number	VALIDATE_NUMBER	Data is numerical.
Integer	VALIDATE_INTEGER	Data is an integer.
Positive integer	VALIDATE_POSITIVE_INTEGER	Data is a positive integer.
Alphabetic	VALIDATE_ALPHABETIC	Data is alphabetic, a-z and/or A-Z.
US phone number	VALIDATE_US_PHONE	Data consists only of 10 digits and optionally the characters `(`, `)`, and `-`.
International phone number	VALIDATE_INTL_PHONE	Data consists only of numbers and the characters `(`, `)`, `+`, and `-`.

Error Handlers

You can create methods in your servlet to handle specific validation failures. These methods are called error handlers, and generally follow this method signature:

```
public void myErrorHandler(HttpServletRequest req,
                           HttpServletResponse res);
```

Use this method to generate an error page if the `validate()` method returns false, indicating a validation error. (For information on the `validate()` method, see "Validation Methods" on page 264).

You can examine the type of error using an error vector of type `Vector`, using the following methods in the `HttpServletRequest2` interface:

Table 13-7 Error Handler Vector Methods

Error Handler Vector Method	Description
getErrorCodes()	Returns a vector of error codes that correspond to the input variables that failed to validate. An error code is associated with each type of data validated.
getErrorMsgs()	Returns a vector of error messages that correspond to the input variables that failed to validate.
getErrorVars()	Returns a vector of the input variables that failed to validate.

The following table shows the error code and message associated with each validation rule:

Table 13-8 Validation Rules

Validation Rule	Error Code	Error Message
VALIDATE_NUMBER	ERROR_NUMBER	Wrong number format!
VALIDATE_INTEGER	ERROR_INTEGER	Wrong integer format!
VALIDATE_POSITIVE_INTEGER	ERROR_POSITIVE_INTEGER	Wrong positive integer format!
VALIDATE_ALPHABETIC	ERROR_ALPHABETIC	Wrong alphabetic format!
VALIDATE_US_PHONE	ERROR_US_PHONE	Wrong US phone format!
VALIDATE_INTL_PHONE	ERROR_INTL_PHONE	Wrong international phone format!

Table 13-8 Validation Rules

Validation Rule	Error Code	Error Message
VALIDATE_EMAIL	ERROR_EMAIL	Wrong email format!
VALIDATE_SSN	ERROR_SSN	Wrong social security format!
VALIDATE_DATE	ERROR_DATE	Wrong date format!
VALIDATE_DAY	ERROR_DAY	Wrong day format!
VALIDATE_MONTH	ERROR_MONTH	Wrong month format!
VALIDATE_YEAR	ERROR_YEAR	Wrong year format!
VALIDATE_US_ZIPCODE	ERROR_ZIP	Wrong zip code format!

Example Validation Rules

The following example shows the `Parameters` section from a servlet configuration file. This section describes a form consisting of several parameters, including a name, social security number, an address, an email address, and a US phone number:

```
"Parameters"    NTV {
        "name”  NTV {
                "inputRequired"       Str     "y",
        },
        "zip"   NTV {
                "inputRequired"       Str     "y",
                "inputRule"           Str     "VALIDATE_US_ZIPCODE
",
        },
        "ssn"   NTV {
                "inputRequired"       Str     "y",
                "inputRule"           Str     "VALIDATE_SSN",
        },
        "email"  NTV {
                "inputRequired"       Str     "n",
                "inputRule"           Str     "VALIDATE_EMAIL",
        },
        "phone"  NTV {
                "inputRequired"       Str     "y",
                "inputRule"           Str     "VALIDATE_US_PHONE",
        }
}
```

In this example, the user does not need to supply an email address but must supply a name, zip code, a social security number, and a US phone number in the form. The zip code, social security number, phone number, and email (if it is present) are checked for valid data.

Note that this validation does not fail if the email value is missing, only if it is present and does not match the `VALIDATE_EMAIL` rule.

Creating Named Form Action Handlers

You can create methods that handle particular buttons on a form. This enables you to build in a level of modularity to your servlet to handle requests cleanly.

Form handlers are used in code generated by Netscape Application Builder (NAB). Usage consists of two methods in the `HttpServletRequest2` interface, coupled with entries in the servlet's configuration file.

Use the following methods in `service()` (generic servlets) or `doGet()` or `doPost()` (HTTP servlets) to handle requested actions. These methods reside in the `HttpServletRequest2` interface. Note that you must also configure form action handlers in the `FormActionHandlers` section of the servlet's configuration file.

Table 13-9 Named Form Action Handlers

Method	**Description**
dispatchAction()	Calls a method corresponding to a form action in a servlet.
formActionHandlerExists()	Determines whether the servlet has form actions defined on it. This method is used in code generated by iAB, and typically is not necessary for non-generated code.

Example Validation and Form Action Handler

This example shows a servlet and its configuration file. The servlet performs some validation on the incoming request and then passes it to a form action handler called `submitHandler()`.

Servlet Configuration File

```
NTV-ASCII   {
    "DispatchServlet"    NTV    {

        "ServletRegistryInfo"    NTV    {
            "type"                 Str    "j",
            "enable"               Str    "y",
            "encrypt"              Str    "n",
            "lb"                   Str    "y",
            "descr"               Str    "Testing action dispatch",
            "group"                StrArr ["Actions"],
            "guid"                 Str
                              "{6952A1AC-FED2-1687-9BB6-080020A1689
6}",
        },

        "ServletRunnerInfo"    NTV    {
            "ServletClassPath"       Str
                "com.netscape.server.servlet.test.TestDispatchServl
et",
        },

        "ServletData"    NTV    {
            "FormActionHandlers"    NTV    {
                "submitAction"     Str    "submitHandler",
            },
        },
    },
}
```

```
Servlet Source Code
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * An HTTP Servlet that responds to the GET and HEAD methods of the
 * HTTP protocol.  It returns a form to the user that gathers data.
 * The form POSTs to another servlet.
 */

public class TestDispatchServlet extends HttpServlet {


    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
                throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Get the user’s session and shopping cart
        out.println("<html>"
                  + "<head><title> TestDispatch </title></head>"
                  + "<body bgcolor=\"#ffffff\">\n"
                  + "<br>");

        out.println("<form action=\""
                  + response.encodeUrl("/servlet/DispatchServlet")
                  + "\" method=\"post\">"
                  + "<strong>Please Update your account information"
                  + "</strong><br><br><br>"
                  + "<table>"
                  + "<tr>"
                  + "<td><strong>Your Name:</strong></td>"
                  + "<td><input type=\"text\" name=\"personname\""
                  + "\" size=\"19\"></td>"
                  + "</tr>"

                  + "<tr>"
                  + "<td><strong>Account ID:</strong></td>"
                  + "<td><input type=\"text\" name=\"accountID\""
                  + "\" size=\"19\"></td>"
                  + "</tr>"

                  + "<tr>"
                  + "<td><strong>Your Password:</strong></td>"
                  + "<td><input type=\"password\"
name=\"password1\""
                  + "\" size=\"19\"></td>"
                  + "</tr>"

                  + "<tr>"
                  + "<td><strong>Match Password:</strong></td>"
                  + "<td><input type=\"password\"
name=\"password2\""
                  + "\" size=\"19\"></td>"
                  + "</tr>"

                  + "<tr>"
                  + "<td></td>"
                  + "<td><input type=\"submit\"
name=\"submitAction\""
                  + "value=\"Submit Information\"></td>"
                  + "</tr>"

                  + "</table>"
                  + "</form>"
                  + "</td></tr></table></body>"
                  + "</html>");
        out.close();
    }


    public void doPost (HttpServletRequest request,
                        HttpServletResponse response)
                throws ServletException, IOException
    {
        HttpServletRequest2 newReq = (HttpServletRequest2) request;
        if( newReq.validate(response)) {
            newReq.dispatchAction(response,this);
        }
    }


    public int submitHandler ( HttpServletRequest request,
                               HttpServletResponse response)
               throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String p1 = request.getParameter("password1");
        String p2 = request.getParameter("password2");

        if( p1 == null ||
            p2 == null ||
            ! p1.equals(p2) )
        {
            out.println("<html>"
                      + "<head><title> TestDispatch </title></head>"
                      + "<body bgcolor=\"#ffffff\">\n"
                      + "<br><br>Your password does not match! "
                      + "Please try again"
                      + "</body>"
                      + "</html>");
            out.close();
            return HttpServletRequest2.ERROR_USER;
        }
        out.println("<html>"
                  + "<head><title> TestDispatch </title></head>"
                  + "<body bgcolor=\"#ffffff\">\n"
```

```
                    + "<br><br>Your Account information: <br><br>"
                    + "<br>Your name: "
                    + request.getParameter("personname")
                    + "<br>Account ID: "
                    + request.getParameter( "accountID")
                    + "<br><br>Updated successfully in the database"
                    + "</body>"
                    + "</html>");
          out.close();
          return HttpServletRequest2.NO_ERROR;
      }
  }
```

Using the Java Message Service

This appendix describes how to use the Java Message Service (JMS) API. iAS 6.0 allows the integration of third party JMS providers into its Java environment, and provides two added value features: Connection Pooling and User identity mapping.The following topics are included in this chapter:

- About the JMS API
- Enabling JMS and Integrating Providers
- Using JMS in Applications
- JMS Administration
- Sample Applications
- Future of JMS in iAS

About the JMS API

The Java Message Service is a Java2 Enterprise Edition API. JMS provides a set of standard Java language interfaces to Enterprise Messaging Systems, often referred to as “message oriented middleware.” These interfaces are implemented by JMS providers. iAS supports the JMS provider for IBM MQ Series.

The JMS web page at java.sun.com describes the purpose of JMS as follows:

Enterprise messaging provides a reliable, flexible service for the asynchronous exchange of critical business data and events throughout an enterprise. The JMS API adds to this a common API and provider framework that enables the development of portable, message based applications in the Java programming language.

iAS 6.0 also includes JMS Connection Pooling and User Identity Mapping. These are provided through an adminstrative framework and iAS specific code is not required. Applications can use these features transparently, maintaining component portability.

JMS Messaging Styles

JMS supports two messaging styles:

- Point-to-point allows two programs to communicate by sending and receiving messages through a Destination called a Queue.
- Publish/subscribe allows several messaging programs to communicate through a Destination called a Topic. Messages are sent by publishing to a Topic. Messages are received by "subscribers."

Regardless of messaging style, the link between applications and the JMS provider is the connection object. Applications get their connection objects from Connection Factories.

In order maximize portability of applications between JMS providers, provider specific aspects of messaging are encapsulated in administered objects. JMS administered objects implement one of the following four JMS interfaces, two for each messaging style:

1. Destination
   - Queue
   - Topic
2. ConnectionFactory
   - QueueConnectionFactory
   - TopicConnectionFactory

JMS providers supply classes that implement these interfaces. Administration tools are used to create and configure instances of the administered object classes, and to configure them to the requirements of the deployment. Administrators use the tools to set provider specific parameters.

This programming model allows JMS programs to be written that are completely independent of the provider. Applications look up the administered objects by name using the Java Naming and Directory Interface (JNDI).

The following sample program looks up its connection factory and destination, and sends a simple text message to a queue (exception handling has been omitted for clarity):

```
// Use JNDI to find the connection factory and the destination
Context ctx = new InitialContext();

QueueConnectionFactory factory;

factory = (QueueConnectionFactory) ctx.lookup
("java:comp/env/jms/theFactory");Queue queue = (Queue)
ctx.lookup("java:comp/env/jms/theQueue");


// create a connection, session, sender and the message
QueueConnection conn;

conn = factory.createQueueConnection("myUserName", "myPassword");

QueueSession session = connection.createQueueSession (false,
Session.AUTO_ACKNOWLEDGE);

QueueSender sender = session.createSender(queue);

TextMessage msg = session.createTextMessage();

msg.setText("Hello from a simple Java Message Service Application");


// start up the connection, send the message

connection.start();

sender.send(msg);

connection.stop();


// now close all resources to insure that native resources are
released

sender.close();

session.close();

connection.close();
```

Note that this application did not hard code the names of the resources, but instead used J2EE resource references, as described in the section on application deployment. Applications should reference objects in the JMS subcontext directly, since the iAS deployment manager does not support iAS JMS beta JMS resource references.

Enabling JMS and Integrating Providers

IAS6.0 includes the software required to integrate JMS providers, but it must be enabled. This section describes how to integrate a JMS provider with iAS, and to enable the JMS connection pooling and user identity mapping features.

Running the program jms/bin/jms_setup during iAS installation enables JMS. This program prompts the user for information about the JMS provider, and performs the appropriate setup including:

- Set up the system Java class path and dynamic library paths
- Enable the object pooling infrastructure in the iAS runtime
- Adapt the JMS administration programs to the iAS installation location

The JMS provider software should have been installed previously. See the vendor's installation instructions.

Enabling JMS Connection Pooling

To enable the JMS Connection Pooling, simply run the program `jms_reg`.

Enabling the IBM MQ Provider

The `jms_setup` is preconfigured to use the MQ Series JMS provider. For example when entering:

```
#./jms_setup
```

The response is:

```
Are you using the IBM MQ v5.1 as message provider [Y] : Y
```

Enabling the Sun JMQ Provider

To set up using the Sun JMQ:

```
#./jms_setup
```

```
Are you using the IBM MQ v5.1 as message provider [Y] : n
```

Enter message provider C library absolute path (one each time, if done, hit return only) :

Enter message provider Java jar file absolute path (one each time, if done, hit return only):

```
/opt/SUNWjmq/lib/jmq.jar
```

```
/opt/SUNWjmq/lib/jmqadmin.jar
```

Using JMS in Applications

The support for JMS included with iAS 6.0 is based entirely on the standard J2EE APIs. Application components using the added value features will be portable with other J2EE environments. This section discusses some issues that programmers should consider when using JMS in applications deployed on iAS 6.0.

JNDI and Application Component Deployment

JMS objects are stored by administration tools in the jms subcontext of the iAS root JNDI namespace. The JMS subcontext does not support creation of subcontexts of itself. Links to the components application context are established at application deployment time.

When an InitialContext is created with the default parameters, JMS objects may be referenced by names beginning with `jms/`. Greater flexibility can be achieved by using J2EE resource references. This was demonstrated in the sample code shown on page 2, where the name looked up for the factory was `java:comp/env/jms/theFactory`. In the iAS JMS beta, JMS resource references are not supported. JMS objects should be referenced directly.

Connection Factory Proxy

iAS 6.0 supports the JMS connection pooling and user identity maps. The ConnectionFactoryProxy class functions by interposing between the application and the JMS provider's connection factory. There are two proxy classes, one for each messaging style:

- QueueConnectionFactoryProxy
- TopicConnectionFactoryProxy.

The APIs presented by the proxy classes are the standard JMS APIs: QueueConnectionFactory and TopicConnectionFactory. Only administrators need be concerned with the proxies, which may be used transparently to application code.

A simple administration program configures ConnectionFactoryProxies. The proxies handle connection pooling and user id mapping. JMS operations are forwarded to a connection obtained by the proxy from a provider factory specified by the administrator.

Connection Pooling

Setting up a JMS connection can be network intensive and therefore expensive. Connection pooling facilitates the re-use of JMS connections. When pooling is enabled, and an application closes a connection, the proxy returns the connection to the pool instead of closing the provider connection. When a subsequent application attempts to create a connection using the same username and password, the proxy will re-use the connection.

User Identity Mapping

The Connection Factory Proxy also provides user identity mapping. JMS providers do not use the same security infrastructure as the application server and thus have different user namespaces. User identity mapping provides administrators flexibility in designing their security infrastructure.

Two forms of mapping are provided by the connection factory proxy classes:

- Default username
- Explicit user id map

As with connection pooling, this functionality is implemented by the proxy classes within the standard JMS API. When using this user identity mapping, the deployment depends on the iAS user security mechanisms to control access to the messaging system.

About Default Username

Default username and password enable multiple application users to share a single messaging system provider user id and password.

When a proxy is created, the administrator may define a default user name and password for the proxy. Applications invoking the no argument create connection

method pass these values to the provider factory when creating a connection. For example, when the application calls:

```
connection = proxy.createQueueConnection();
```

if a default user name has been configured, the iAS implementation of the proxy obtains its JMS Connection with:

```
connection = providerFactory.createQueueConnection (defaultUserName,
defaultPassword);
```

About Explicit User ID Map

An explicit user id map may also be used. The map contains a list of entries, each referenced by a unique user id key and containing two values:

- jmsUserName
- jmsPassWord.

The administrator creates the map using a provided tool called jmsuadm. The values in the entry are used when creating connections.

For example, when an application creates a connection using the proxy with:

```
connection = proxy.createQueueConnection(userString,
passWordString);
```

the iAS proxy implementation looks up an entry for the given `userString` in the map. If it finds an entry, the proxy passes `jmsUserName` and `jmsPassWord` values from that entry to the jms provider factory, ignoring the application provided password.

That is, the proxy effectively executes:

```
connection = providerFactory.createConnection (entry.jmsUserName,
entry.jmsPassWord);
```

If no entry matching userString is found in the user identity map, the application provided values are passed through to the providerFactory.

ConnectionFactoryProxies and Application Created Threads

A servlet can create Java threads, but it is not recommended. User created threads will not be known to the JMS connection pooling infrastructure.

Applications must not invoke the create connection or connection close methods from user created threads. Attempting to do so will result in:

`javax.jms.IllegalStateException`.

This is not implemented in JMS beta. In beta, applications that attempt to create or close connections from application created threads will crash KJS.

JMS Features Not Supported

iAS 6.0 does not support the JMS XAConnection and server session pools features described in Chapter 8 of the JMS specification. These features will be supported in a subsequent iAS release.

JMS Administration

The JMS API depends on administered objects for portability. Provider specific aspects of a deployment are encapsulated in Administered objects allowing application code to be portable.

In the iAS environment JMS administration consists of four tasks

- Creating JMS provider factories and destinations
- Creating user id maps
- Creating ConnectionFactoryProxies
- Modifying the connection pooling parameters in the iAS registry.

JMS Object Administration Tools

Each JMS product should include an administration program. This tool creates objects and binds them to names in the iAS JNDI. This section describes the Java properties and system paths required to configure a tool for working with the JMS JNDI Context. Consult your provider's documentation for descriptions of how specific tools are configured. (A script for launching the administration tool for IBM MQ JMS for iAS is described in the next section.)

To access the JMS context, create the InitialContext using the following property values:

Table A-1 Java Property Names and Values

Java Property Name	**Property value**
Java.naming.factory.initial	com.netscape.server.jndi.ExternalContextFactory
Java.naming.provider.url	/jms

JNDI Properties for JMS Administration Tools

For the Java classes required to access the JMSContext, include the following three jar files in the Java runtime classpath:

- `GX_ROOTDIR/classes/java/jms.jar`
- `GX_ROOTDIR/classes/java/javax.jar`
- `GX_ROOTDIR/classes/java/kfcjdk11.jar`

where `GX_ROOTDIR` is the location of the iAS installation. Example:

`/usr/iPlanet/iAS6/ias`).

On Solaris, the following directory must be included in the `LD_LIBRARY_PATH`

`$GX_ROOTDIR/gxlib`

JMS Object Administration for IBM MQ

The `mqjmsadm` script that launches the IBM MQ JMS administration program is included in iAS. It is located in `GX_ROOTDIR/jms/bin`. The administration program is a Java class. mqjmsadm is an interactive command line program that accepts input from the administrator, or from an input file.

Operation is described in the MQSeries documentation for JMSAdmin. mqjmsadm handles the JNDI configuration automatically, so it is not necessary to use the `-cfg` option.

For example, a connection factory and queue could be created with the following `mqjmsadm` session:

```
# mqjmsadm
```

Response:

```
5648-C60 (c) Copyright IBM Corp. 1999. All Rights Reserved.

Starting MQSeries Classes for Java(tm) Message Service
Administration

Connected to LDAP server on localhost port 389

InitCtx> define q(theQueue) queue(SYSTEM.DEFAULT.LOCAL.QUEUE)

InitCtx> define qcf(theFactory)

InitCtx> display ctx

Contents of InitCtx

a  aQueue                    com.ibm.mq.jms.MQQueue

a  theProviderFactory        com.ibm.mq.jms      .
MQQueueConnectionFactory

2 Object(s)

0 Context(s)

2 Binding(s), 2 Administered

InitCtx> end
```

The JMS context does not support subcontexts, so using JMSAdmin commands to manipulate sub-contexts will generate error messages.

Connection Factory Proxy Administration

Connection Factory Proxies are created with the command jmspadm (jms proxy administrator). This command (shell script for Unix or BAT file for NT) launches a java program that creates connection factory proxies with given parameters, and binds them in JNDI. The proxy parameters are set by command line arguments.

The command performs three operations on proxies:

- Creating a proxy
- Deleting a proxy
- Listing proxy parameters

Creating a Proxy:

To create a proxy enter:

```
jmspadm proxyName factoryName <-p or +p> <-u user password> <-m
userMapNam>
```

The first two arguments are required:

- JNDI name to be given to the new proxy
- JNDI name for the connection factory to be proxied.

Since JMS objects may only be found in the jms subcontext, if the supplied names do not begin with `jms` , that string is prepended. For example, the following two commands have the same result:

- `jmspadm theFactory theProviderFactory`
- `jmspadm jms/theFactory jms/theProviderFactory`

Using the provider specific tool, create the factory before running jmspadm, to make the factory class available.

The remaining arguments are optional. They are used control the operation of the proxy at runtime. The default settings are:

- Connection pooling is on. Disable connection pooling by using `-p`.
- No default userid and password. Set them by using `-u` .
- No identity map. Setting the JNDI name of a user id map to be used by the proxy is discussed below.

Deleting a Proxy

The syntax to delete a proxy is:

```
jmspadm -d proxyName
```

Listing Proxy Parameters

To list all proxies stored in JNDI use the command: jmspadm –l.

User ID Map Administration

To create a user identity map the administrator must prepare an xml file

Once this file is ready, use the command `jmsuadm`. Again there are three variations to the command:

- `jmsuadm mapName mapFileName` reads the given file and creates a user id map.
- `jmsuadm -d mapName` deletes the map.
- `jmsuadm –l` lists the names of maps.

For security purposes, the contents of the map cannot be listed. Administrators should protect the input files carefully.

The input file format is XML. The public name for the data type description (DTD) is:

```
-//Sun Microsystems, Inc.//DTD iAS JMS User Identity Map 1.0//EN
```

The following example input file contains the mappings for two JMS users:

```
<?xml version="1.0" encoding="iso8859-1"?>

<!DOCTYPE jms-user-id-map PUBLIC "-//Sun Microsystems, Inc.//DTD iAS
JMS User Identity Map 1.0//EN" "TODO: fill this in" >

  <jms-user-id-map>

        <user>

                <name>bob</name>

                <jms-name>jmsuser</jms-name>

                <jms-password>secret</jms-password>

        </user>

        <user>

                <name>nancy</name>

                <jms-name>jmsuser2</jms-name>

                <jms-password>private</jms-password>

        </user>
```

```
</jms-user-id-map>
```

Each user element must contain each of the three elements noted above:

- name
- jms-name
- jms-password,

  although empty values are allowed:

  (`<jms-name></jms-name>`).

Connection Pooling Configuration

Certain parameters for the JMS Connection pool are stored in the iAS registry. If desired, these may be adjusted using the kregedit program in the iAS bin directory.

The parameters are stored in the key:

```
SOFTWARE\iPlanet\ApplicationServer\6.0\CCS0\POOLS\JMSConnectionPool
```

Following are the parameter names and default values:

Table A-2 Parameter Names and Default Values for Connection Pooling

Parameter	**Default Value**	**Description**
MaxPoolSize	20	maximum # of pooled JMS connections
SteadyPoolSize	10	# of steady state connections
MaxWait	32 seconds	time client will wait for connection
UnusedMaxLife	300 seconds	time unused connections are deleted
DebugLevel	1	0-turns off logging 1-logs callback messages 2-logs all messages (see kjs log file)
MonitorInterval	60	Time between messages

Connections are deleted when they are closed if the number of connections in the pool is between SteadyPoolSize and MaxPoolSize. Connections are kept in the pool up to UnusedMaxLife, when the number of open connections is less than SteadyPoolSize.

Sample Applications

JMS sample applications can be found in the directory

```
GX_ROOTDIR/jms/samples
```

See the README files in for more information

Future of JMS in iAS

Default JMS Provider

A future release of the J2EE standard will require that the environment include a JMS provider.

Message Driven Enterprise Java Beans

J2EE and iAS 6.0 do not currently support application components that receive scalable messages. A future release of J2EE will include support for "Message Driven Enterprise Java Beans," which are activated in response to the receipt of JMS messages. The application framework allows for scalable message receipt.

Using JMS in distributed transactions

A future release of iAS will support JMS resources in global transactions.

```
(javax.jms.XAConnection)
```

Appendix B

Dynamic Reloading

This appendix describes dynamic loading issues relating to the iAS 6.0 Java class loader.

Some iAS components, namely servlets and JSPs, can be dynamically reloaded into the server while it is running. This allows you to make changes to your application without restarting. You can also change some attributes in application or servlet configuration files, namely those that are not uploaded to the registry and those that do not change the session information.

Additionally, you can instruct iAS to make other classes dynamically reloadable by changing values in the iAS registry.

NOTE

Enterprise JavaBeans (EJBs) are not dynamically reloadable. If you make a change to an EJB, you must restart the server in order to effect the change.

This appendix contains the following sections:

- How Dynamic Reloading Works
- Summary of Related Registry Entries

How Dynamic Reloading Works

By default, you can make changes to servlets or JSPs while iAS is running, and iAS notices the change within 10 seconds and instantiates the new component. EJBs can not be reloaded dynamically.

You can change the duration before iAS notices the change by editing the registry entry `SYSTEM_JAVA\GX_TASKMANAGER_PERIOD`.

You can make individual classes other than servlets and JSPs dynamically reloadable by entering the class name into a semicolon-separated list in the registry entry `SYSTEM_JAVA\GX_VERSIONABLE`. By default, all servlets and JSPs are reloadable.

You can make a set of classes dynamically reloadable depending on whether they implement a certain interface by entering the interface name into the registry entry `SYSTEM_JAVA\GX_VERSIONABLE_IF_IMPLEMENTS`. By the same token, you can make all classes that extend a certain class dynamically reloadable by entering the superclass name into the iAS registry entry `SYSTEM_JAVA\GX_VERSIONABLE_IF_EXTENDS`.

Finally, you can make all classes (other than EJBs) dynamically reloadable by setting the iAS registry entry `GX_ALL_VERSIONABLE` to 1 (the default is 0). This setting is normally used for backward compatibility with older versions of iAS.

If you start iAS from the command line, you can turn off dynamic reloading completely by using the `-d` option.

Summary of Related Registry Entries

The following registry entries control how dynamic reloading works in iAS. Note that these entries are not relevant if the server is started with the `-d` option.

NOTE

iAS does not support dynamic reloading of EJBs.

Table B-1 Registry Entries

Registry Entry	Default Value	Description
`GX_TASKMANAGER_PERIOD`	10	The maximum number of seconds before iAS notices that a class has been changed.
`GX_ALL_VERSIONABLE`	0	Set to 1 to indicate that all classes (except EJBs) are dynamically reloadable. This is normally used for backward compatibility with older versions of iAS.
`GX_VERSIONABLE`	null	Semicolon-separated list of classes that are explicitly reloadable.

For example, the default value for `SYSTEM_JAVA\GX_VERSIONABLE_IF_IMPLEMENTS` includes the classes for generic servlets and HTTP servlets, which makes all servlets and JSPs reloadable. You could add `com.kivasoft.applogic.AppLogic` to this registry entry to make all AppLogics reloadable as well.

Appendix C

Sample Deployment Files

This appendix contains sample iAS Deployment Descriptor (DD) files, used for application and component deployment. The following sample DD XML files are included in this appendix:

- "Application DD XML Files"
- "Web Application DD XML Files"
- "EJB-jar DD XML Files"
- "Rich Client DD XML Files"
- "Resource DD XML Files"

Application DD XML Files

The application deployment descriptor (DD) gives a top level view of all the application's contents. There are two types of application DDs; one is the J2EE application DD, and the other is the iAS application DD. These descriptors are XML files specified by Document Type Definitions (DTDs).

The J2EE application DD is described by the *J2EE specification, v2.1 Section 8.4 "J2EE:application XML DTD"*. The iAS application DD is described by the iAS web application DTD described in this document's *Chapter 10, "Packaging for Deployment"*.

Sample Application DD XML File

This section provides an example of a J2EE application deployment descriptor (DD) XML file. The J2EE application DD that follows, has a file name of `application.xml`.

```
<?xml version="1.0"?>

<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application 1.2//EN'
'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>


<application>
  <description>Application description</description>
  <display-name>estore</display-name>
  <module>
    <ejb>estoreEjb.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>estore.war</web-uri>
      <context-root>estore</context-root>
    </web>
  </module>
  <security-role>
    <description>the customer role</description>
    <role-name>customer</role-name>
  </security-role>
</application>
```

Sample iAS Application DD XML File

To be supplied.

Web Application DD XML Files

The web application deployment descriptor (DD) conveys the elements and configuration information of a web application between Developers, Assemblers, and Deployers. These descriptors are XML files specified by Document Type Definitions (DTDs).

The Web application ARchive (WAR) file contains a J2EE web application deployment descriptor (DD) and an iAS web application deployment descriptor. The J2EE web application DD is described by the *Java Servlet Specification, v2.2 Chapter 13 Deployment Descriptors*. The iAS application DD is described by the iAS web application DTD described in this document's "Chapter 10, "Packaging for Deployment".

Sample Web Application DD XML File

This section provides an example of a J2EE web application deployment descriptor (DD) XML file. The web application DD that follows, has a file name of `web.xml`.

```
<?xml version="1.0"?>
    <!DOCTYPE web-app>
    <web-app>
      <description>no description</description>
      <display-name>DukesPetStoreWebTier</display-name>
      <servlet>
        <description>no description</description>
        <display-name>centralJsp</display-name>
        <servlet-name>webTierEntryPoint</servlet-name>
        <jsp-file>Main.jsp</jsp-file>
        <load-on-startup>-1</load-on-startup>
      </servlet>
      <servlet-mapping>
        <servlet-name>webTierEntryPoint</servlet-name>
        <url-pattern>/control/*</url-pattern>
      </servlet-mapping>
```

```
<session-config>
  <session-timeout>54</session-timeout>
</session-config>
<welcome-file-list>
  <welcome-file>/index.html</welcome-file>
</welcome-file-list>
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/errorpage.jsp</location>
</error-page>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>MySecureBit0</web-resource-name>
    <description>no description</description>
    <url-pattern>/control/placeorder</url-pattern>
    <http-method>POST</http-method>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>no description</description>
    <role-name>customer</role-name>
  </auth-constraint>
  <user-data-constraint>
    <description>no description</description>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>MySecureBit1</web-resource-name>
    <description>no description</description>
    <url-pattern>/Main.jsp/signin</url-pattern>
    <http-method>POST</http-method>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>no description</description>
    <role-name>customer</role-name>
  </auth-constraint>
  <user-data-constraint>
    <description>no description</description>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>MySecureBit1</web-resource-name>
    <description>no description</description>
    <url-pattern>/control/signin</url-pattern>
    <http-method>POST</http-method>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>no description</description>
    <role-name>customer</role-name>
  </auth-constraint>
  <user-data-constraint>
    <description>no description</description>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<security-constraint>
```

```
                  <web-resource-collection>
                    <web-resource-name>MySecureBit0</web-resource-name>
      <description>no description</description>
      <url-pattern>/Main.jsp/placeorder</url-pattern>
      <http-method>POST</http-method>
      <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
      <description>no description</description>
      <role-name>customer</role-name>
    </auth-constraint>
    <user-data-constraint>
      <description>no description</description>
      <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <login-config>
    <auth-method>FORM</auth-method>
    <realm-name>default</realm-name>
    <form-login-config>
      <form-login-page>/estore/login.jsp</form-login-page>
      <form-error-page>/estore/error.html</form-error-page>
    </form-login-config>
  </login-config>
  <security-role>
    <description>the customer role</description>
    <role-name>customer</role-name>
  </security-role>
  <ejb-ref>
    <description>no description</description>
    <ejb-ref-name>account</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>com.sun.estore.account.ejb.AccountHome</home>
    <remote>com.sun.estore.account.ejb.Account</remote>
  </ejb-ref>
  <ejb-ref>
    <description>no description</description>
    <ejb-ref-name>order</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>com.sun.estore.order.ejb.OrderHome</home>
    <remote>com.sun.estore.order.ejb.Order</remote>
  </ejb-ref>
  <ejb-ref>
```

```
    <description>no description</description>
    <ejb-ref-name>mailer</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.sun.estore.mail.ejb.MailerHome</home>
    <remote>com.sun.estore.mail.ejb.Mailer</remote>
  </ejb-ref>
  <ejb-ref>
    <description>no description</description>
    <ejb-ref-name>estorekeeper</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.sun.estore.control.ejb.EStorekeeperHome</home>
    <remote>com.sun.estore.control.ejb.EStorekeeper</remote>
  </ejb-ref>
  <ejb-ref>
    <description>no description</description>
    <ejb-ref-name>catalog</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.sun.estore.catalog.ejb.CatalogHome</home>
    <remote>com.sun.estore.catalog.ejb.Catalog</remote>
  </ejb-ref>
  <ejb-ref>
    <description>no description</description>
    <ejb-ref-name>cart</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.sun.estore.cart.ejb.ShoppingCartHome</home>
    <remote>com.sun.estore.cart.ejb.ShoppingCart</remote>
  </ejb-ref>
  <ejb-ref>
    <description>no description</description>
    <ejb-ref-name>inventory</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.sun.estore.inventory.ejb.InventoryHome</home>
    <remote>com.sun.estore.inventory.ejb.Inventory</remote>
  </ejb-ref>
</web-app>
<?xml version="1.0"?>
```

Sample iAS Web-app DD XML File

This section provides an example of an iAS web application deployment descriptor (DD) XML file. The iAS web application DD that follows, has a file name of `ias-web.xml`.

```
<!DOCTYPE web-app>
<ias-web-app>
  <servlet>
    <servlet-name>webTierEntryPoint</servlet-name>
        <guid>{Deadbeef-AB3F-11D2-98C5-000000000000}</guid>
  </servlet>
  <ejb-ref>
    <ejb-ref-name>account</ejb-ref-name>
    <jndi-name>ejb/estoreWar/account</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>order</ejb-ref-name>
    <jndi-name>ejb/estoreWar/order</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>mailer</ejb-ref-name>
    <jndi-name>ejb/estoreWar/mailer</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>estorekeeper</ejb-ref-name>
    <jndi-name>ejb/estoreWar/estorekeeper</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>catalog</ejb-ref-name>
    <jndi-name>ejb/estoreWar/catalog</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>cart</ejb-ref-name>
    <jndi-name>ejb/estoreWar/cart</jndi-name>
  </ejb-ref>
  <ejb-ref>
```

```
        <ejb-ref-name>inventory</ejb-ref-name>
        <jndi-name>ejb/estoreWar/inventory</jndi-name>
      </ejb-ref>
    </ias-web-app>
```

EJB-jar DD XML Files

The ejb-jar file contains a deployment descriptor (DD) in the format defined by the *Enterprise JavaBeans Specification, v1.1* and an iAS ejb DD in the format defined by *Chapter 10, "Packaging for Deployment"* of this document.

Sample EJB-jar DD XML File

This section provides an example of a J2EE ejb deployment descriptor (DD) XML file. The ejb-jar DD that follows, has a file name of `ejb-jar.xml`.

```
<?xml version="1.0"?>
<ejb-jar>
  <description>no description</description>
  <display-name>Ejb1</display-name>
  <enterprise-beans>
    <session>
      <description>no description</description>
      <display-name>TheMailer</display-name>
      <ejb-name>TheMailer</ejb-name>
      <home>com.sun.estore.mail.ejb.MailerHome</home>
      <remote>com.sun.estore.mail.ejb.Mailer</remote>
      <ejb-class>com.sun.estore.mail.ejb.MailerEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <ejb-ref>
```

```
        <ejb-ref-name>account</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <home>com.sun.estore.account.ejb.AccountHome</home>
        <remote>com.sun.estore.account.ejb.Account</remote>
        <ejb-link>TheAccount</ejb-link>
      </ejb-ref>
      <ejb-ref>
        <ejb-ref-name>order</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <home>com.sun.estore.order.ejb.OrderHome</home>
        <remote>com.sun.estore.order.ejb.Order</remote>
        <ejb-link>TheOrder</ejb-link>
      </ejb-ref>
      <resource-ref>
        <description>description</description>
        <res-ref-name>MailSession</res-ref-name>
        <res-type>javax.mail.Session</res-type>
        <res-auth>Application</res-auth>
      </resource-ref>
    </session>
    <session>
      <description>no description</description>
      <display-name>TheEstorekeeper</display-name>
      <ejb-name>TheEstorekeeper</ejb-name>
      <home>com.sun.estore.control.ejb.EStorekeeperHome</home>
      <remote>com.sun.estore.control.ejb.EStorekeeper</remote>
      <ejb-class>com.sun.estore.control.ejb.EStorekeeperEJB
             </ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
```

```
  <env-entry-name>sendConfirmationMail</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>false</env-entry-value>
</env-entry>
<ejb-ref>
  <ejb-ref-name>account</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.sun.estore.account.ejb.AccountHome</home>
  <remote>com.sun.estore.account.ejb.Account</remote>
  <ejb-link>TheAccount</ejb-link>
</ejb-ref>
<ejb-ref>
  <ejb-ref-name>order</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.sun.estore.order.ejb.OrderHome</home>
  <remote>com.sun.estore.order.ejb.Order</remote>
  <ejb-link>TheOrder</ejb-link>
</ejb-ref>
<ejb-ref>
  <ejb-ref-name>mailer</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.sun.estore.mail.ejb.MailerHome</home>
  <remote>com.sun.estore.mail.ejb.Mailer</remote>
  <ejb-link>TheMailer</ejb-link>
</ejb-ref>
<ejb-ref>
  <ejb-ref-name>catalog</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.sun.estore.catalog.ejb.CatalogHome</home>
  <remote>com.sun.estore.catalog.ejb.Catalog</remote>
  <ejb-link>TheCatalog</ejb-link>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>cart</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.sun.estore.cart.ejb.ShoppingCartHome</home>
    <remote>com.sun.estore.cart.ejb.ShoppingCart</remote>
    <ejb-link>TheCart</ejb-link>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>inventory</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.sun.estore.inventory.ejb.InventoryHome</home>
    <remote>com.sun.estore.inventory.ejb.Inventory</remote>
    <ejb-link>TheInventory</ejb-link>
  </ejb-ref>
</session>
<entity>
  <description>no description</description>
  <display-name>TheOrder</display-name>
  <ejb-name>TheOrder</ejb-name>
  <home>com.sun.estore.order.ejb.OrderHome</home>
  <remote>com.sun.estore.order.ejb.Order</remote>
  <ejb-class>com.sun.estore.order.ejb.OrderEJB</ejb-class>
  <persistence-type>Bean</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>False</reentrant>
  <resource-ref>
    <description>description</description>
    <res-ref-name>EstoreDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Application</res-auth>
```

```
          </resource-ref>
        </entity>
        <entity>
          <description>no description</description>
          <display-name>TheAccount</display-name>
          <ejb-name>TheAccount</ejb-name>
          <home>com.sun.estore.account.ejb.AccountHome</home>
          <remote>com.sun.estore.account.ejb.Account</remote>

<ejb-class>com.sun.estore.account.ejb.AccountEJB</ejb-class>
          <persistence-type>Bean</persistence-type>
          <prim-key-class>java.lang.String</prim-key-class>
          <reentrant>False</reentrant>
          <resource-ref>
            <description>description</description>
            <res-ref-name>EstoreDataSource</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
          </resource-ref>
        </entity>
        <session>
          <description>no description</description>
          <display-name>TheCart</display-name>
          <ejb-name>TheCart</ejb-name>
          <home>com.sun.estore.cart.ejb.ShoppingCartHome</home>
          <remote>com.sun.estore.cart.ejb.ShoppingCart</remote>

<ejb-class>com.sun.estore.cart.ejb.ShoppingCartEJB</ejb-class>
          <session-type>Stateful</session-type>
          <transaction-type>Container</transaction-type>
        </session>
        <session>
          <description>no description</description>
          <display-name>TheInventory</display-name>
          <ejb-name>TheInventory</ejb-name>
          <home>com.sun.estore.inventory.ejb.InventoryHome</home>
          <remote>com.sun.estore.inventory.ejb.Inventory</remote>
          <ejb-class>com.sun.estore.inventory.ejb.InventoryEJB
          </ejb-class>
          <session-type>Stateless</session-type>
          <transaction-type>Container</transaction-type>
          <resource-ref>
            <description>description</description>
            <res-ref-name>InventoryDataSource</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
          </resource-ref>
        </session>
        <session>
          <description>no description</description>
          <display-name>TheCatalog</display-name>
          <ejb-name>TheCatalog</ejb-name>
          <home>com.sun.estore.catalog.ejb.CatalogHome</home>
          <remote>com.sun.estore.catalog.ejb.Catalog</remote>

<ejb-class>com.sun.estore.catalog.ejb.CatalogEJB</ejb-class>
          <session-type>Stateless</session-type>
          <transaction-type>Container</transaction-type>
          <resource-ref>
            <description>description</description>
            <res-ref-name>InventoryDataSource</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
          </resource-ref>
        </session>
      </enterprise-beans>
      <assembly-descriptor>
        <container-transaction>
          <method>
            <ejb-name>TheMailer</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>sendOrderConfirmationMail</method-name>
            <method-param>int</method-param>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheMailer</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getPrimaryKey</method-name>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheMailer</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getEJBHome</method-name>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheMailer</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getHandle</method-name>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheMailer</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>isIdentical</method-name>
            <method-param>javax.ejb.EJBObject</method-param>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheEstorekeeper</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getPrimaryKey</method-name>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheEstorekeeper</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>handleEvent</method-name>
            <method-param>com.sun.estore.control.event.EStoreEvent
               </method-param>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheEstorekeeper</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getShoppingCart</method-name>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheEstorekeeper</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getAccount</method-name>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheEstorekeeper</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getOrder</method-name>
            <method-param>int</method-param>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheEstorekeeper</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getEJBHome</method-name>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheEstorekeeper</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getHandle</method-name>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheEstorekeeper</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getOrders</method-name>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheEstorekeeper</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>isIdentical</method-name>
            <method-param>javax.ejb.EJBObject</method-param>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
```

```
  <method>
    <ejb-name>TheEstorekeeper</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getCatalog</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheOrder</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getPrimaryKey</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheOrder</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getOrderDetails</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheOrder</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getEJBHome</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
```

```
<container-transaction>
  <method>
    <ejb-name>TheOrder</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getHandle</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheOrder</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>remove</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheOrder</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>isIdentical</method-name>
    <method-param>javax.ejb.EJBObject</method-param>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheAccount</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getPrimaryKey</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheAccount</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>changeContactInformation</method-name>
    <method-param>com.sun.estore.util.ContactInformation
    </method-param>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheAccount</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getEJBHome</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheAccount</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getHandle</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheAccount</ejb-name>
```

```
        <method-intf>Remote</method-intf>
        <method-name>remove</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>TheAccount</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getAccountDetails</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>TheAccount</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>isIdentical</method-name>
        <method-param>javax.ejb.EJBObject</method-param>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>TheCart</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>updateItemQty</method-name>
        <method-param>java.lang.String</method-param>
        <method-param>int</method-param>
      </method>
      <trans-attribute>Required</trans-attribute>
```

```
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCart</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>deleteItem</method-name>
    <method-param>java.lang.String</method-param>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCart</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getPrimaryKey</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCart</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>empty</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCart</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getEJBHome</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCart</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getHandle</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCart</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>addItem</method-name>
    <method-param>java.lang.String</method-param>
    <method-param>int</method-param>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCart</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getItems</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
```

```
    <ejb-name>TheCart</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>addItem</method-name>
    <method-param>java.lang.String</method-param>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCart</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>isIdentical</method-name>
    <method-param>javax.ejb.EJBObject</method-param>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheInventory</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getPrimaryKey</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheInventory</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getEJBHome</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
```

```
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheInventory</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getHandle</method-name>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheInventory</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>updateInventory</method-name>

<method-param>com.sun.estore.inventory.ejb.InventoryDetails
                </method-param>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
          <method>
            <ejb-name>TheInventory</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>updateQuantity</method-name>
            <method-param>java.lang.String</method-param>
            <method-param>int</method-param>
          </method>
          <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
```

```
  <method>
    <ejb-name>TheInventory</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>isIdentical</method-name>
    <method-param>javax.ejb.EJBObject</method-param>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheInventory</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getInventory</method-name>
    <method-param>java.lang.String</method-param>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheInventory</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getQuantity</method-name>
    <method-param>java.lang.String</method-param>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCatalog</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getPrimaryKey</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCatalog</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getEJBHome</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCatalog</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getHandle</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCatalog</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>searchProducts</method-name>
    <method-param>java.util.Vector</method-param>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>TheCatalog</ejb-name>
```

```
        <method-intf>Remote</method-intf>
        <method-name>findProducts</method-name>
        <method-param>com.sun.estore.catalog.ejb.Category
             </method-param>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>TheCatalog</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>isIdentical</method-name>
        <method-param>javax.ejb.EJBObject</method-param>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>TheCatalog</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getAllCategories</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Sample iAS EJB-jar DD XML File

This section provides an example of an iAS ejb-jar deployment descriptor (DD) XML file. The ejb-jar DD that follows, has a file name of `ias-ejb-jar.xml`.

```
<ias-ejb-jar>
  <enterprise-beans>
     <session>
        <ejb-name>TheMailer</ejb-name>
           <guid>{Deadbabe-AB3F-11D2-98C5-0060B0EF0618}</guid>
           <pass-timeout>100</pass-timeout>
           <session-timeout>180000</session-timeout>
           <is-thread-safe>false</is-thread-safe>
           <pass-by-value>false</pass-by-value>
           <ejb-ref>
              <ejb-ref-name>account</ejb-ref-name>
              <jndi-name>ejb/estoreEjb/TheAccount</jndi-name>
           </ejb-ref>
           <ejb-ref>
              <ejb-ref-name>order</ejb-ref-name>
              <jndi-name>ejb/estoreEjb/TheOrder</jndi-name>
           </ejb-ref>
      </session>
      <session>
         <ejb-name>TheEstorekeeper</ejb-name>
         <guid>{Deadbabe-AB3F-11D2-98C5-000011112222}</guid>
         <pass-timeout>100</pass-timeout>
         <session-timeout>180000</session-timeout>
         <is-thread-safe>false</is-thread-safe>
         <pass-by-value>false</pass-by-value>
         <ejb-ref>
```

```
                <ejb-ref-name>account</ejb-ref-name>
                <jndi-name>ejb/estoreEjb/TheAccount</jndi-name>
            </ejb-ref>
            <ejb-ref>
                <ejb-ref-name>order</ejb-ref-name>
                <jndi-name>ejb/estoreEjb/TheOrder</jndi-name>
            </ejb-ref>
            <ejb-ref>
                <ejb-ref-name>mailer</ejb-ref-name>
                <jndi-name>ejb/estoreEjb/TheMailer</jndi-name>
            </ejb-ref>
            <ejb-ref>
                <ejb-ref-name>catalog</ejb-ref-name>
                <jndi-name>ejb/estoreEjb/TheCatalog</jndi-name>
            </ejb-ref>
            <ejb-ref>
                <ejb-ref-name>cart</ejb-ref-name>
                <jndi-name>ejb/estoreEjb/TheCart</jndi-name>
            </ejb-ref>
            <ejb-ref>
                <ejb-ref-name>inventory</ejb-ref-name>
                <jndi-name>ejb/estoreEjb/TheInventory</jndi-name>
            </ejb-ref>
       </session>
       <session>
         <ejb-name>TheInventory</ejb-name>
         <guid>{deadbabe-ab3f-11d2-98c5-999999990002}</guid>
         <pass-timeout>100</pass-timeout>
         <is-thread-safe>false</is-thread-safe>
         <pass-by-value>false</pass-by-value>
         <session-timeout>180000</session-timeout>
```

```
</session>
<session>
  <ejb-name>TheCatalog</ejb-name>
  <guid>{deadbabe-ab3f-11d2-98c5-999999990003}</guid>
  <pass-timeout>100</pass-timeout>
  <is-thread-safe>false</is-thread-safe>
  <pass-by-value>false</pass-by-value>
  <session-timeout>180000</session-timeout>
</session>
<session>
  <ejb-name>TheCart</ejb-name>
  <guid>{deadbabe-ab3f-11d2-98c5-999999990001}</guid>
  <pass-timeout>100</pass-timeout>
  <is-thread-safe>false</is-thread-safe>
  <pass-by-value>false</pass-by-value>
  <session-timeout>180000</session-timeout>
</session>
<entity>
  <ejb-name>TheAccount</ejb-name>
  <guid>{deadbabe-ab3f-11d2-98c5-999999990000}</guid>
  <pass-timeout>100</pass-timeout>
  <is-thread-safe>false</is-thread-safe>
  <pass-by-value>false</pass-by-value>
  <pool-manager>
    <factory-class-name></factory-class-name>
    <commit-option>NO_CACHE_READY_INSTANCE</commit-option>
    <Ready-pool-timeout>0</Ready-pool-timeout>
    <Ready-pool-maxsize>0</Ready-pool-maxsize>
  </pool-manager>
</entity>
<entity>
```

```
      <ejb-name>TheOrder</ejb-name>
      <guid>{deadbabe-ab3f-11d2-98c5-333344445555}</guid>
      <pass-timeout>100</pass-timeout>
      <is-thread-safe>false</is-thread-safe>
      <pass-by-value>false</pass-by-value>
      <persistence-manager>
            <persistence-manager-factory-class-name>
             com.netscape.server.ejb.PersistenceManagerFactory
             </persistence-manager-factory-class-name>
             <properties-file-location>
                EmployeeRecord_pm1.xml
             </properties-file-location>
             <external-xml-location>
             </external-xml-location>
      </persistence-manager>
      <pool-manager>
        <factory-class-name></factory-class-name>
        <commit-option>NO_CACHE_READY_INSTANCE</commit-option>
        <Ready-pool-timeout>0</Ready-pool-timeout>
        <Ready-pool-maxsize>0</Ready-pool-maxsize>
      </pool-manager>
    </entity>
        </enterprise-beans>
</ias-ejb-jar>
```

Rich Client DD XML Files

To be supplied.

Resource DD XML Files

The following is a sample resource XML descriptor file.

```
<ias-resource>
       <resource>
         <jndi-name>jdbc/SampleSybaseDS1</jndi-name>
           <jdbc>
             <database>nasqadev</database>
             <datasource>SYBFRED</datasource>
             <username>aparna</username>
             <password>aparnak</password>
             <driver-type>SYBASE_CTLIB</driver-type>
           </jdbc>
       </resource>
     </ias-resource>
```

Glossary

This glossary provides definitions for common terms used to describe the iAS deployment and development environment. For a glossary of standard J2EE terms, please see the glossary at *http://java.sun.com/j2ee/glossary.html*.

ACL Access Control List, a list of users or groups and their specified permissions. See *component ACL, general ACL*.

administration server A process in iPlanet Application Server that handles administrative tasks.

API Application Programmer Interface, a set of instructions that a computer program can use to communicate with other software or hardware that is designed to interpret that API.

applet A small application written in Java that runs in a web browser. Typically, applets are called by or embedded in web pages to provide special functionality. By contrast, a *servlet* is a small application that runs on a server.

application A computer program that performs a task or service for a user. Also see *web application*.

application event A named action that you register with the iAS registry. The event occurs either when a timer expires or when the event is called (triggered) from application code at run time. Typical uses for events include periodic backups, reconciling accounts at the end of the business day, or sending alert messages.

application server A program that runs an application in a client/server environment, executing the logic that makes up the application and acting as middleware between a web browser and a datasource.

application tier A conceptual division of an application:

client tier: The user interface (UI). End users interact with client software (web browser) to use the application.

server tier: The business logic and presentation logic that make up your application, defined in the application's components.

data tier: The data access logic that enables your application to interact with a datasource.

AppLogic A iAS-specific class responsible for completing a well-defined, modular task within a iPlanet Application Server application. In NAS 2.1, applications used AppLogics to perform actions such as handling form input, accessing data, or generating data used to populate HTML templates. This functionality is replaced with servlets and JSPs in iAS.

AppPath An iAS registry entry that contains the name of the directory where application files reside. This entry defines the top of a logical directory tree for the application, similarly to the document path in a web server. By default, AppPath contains the value *`BasePath/`*`APPS`, where *`BasePath`* is the base iAS directory. (BasePath is also a iAS variable.)

attribute Attributes are name-value pairs in a request object that can be set by servlets. Contrast with *parameter*. More generally, an attribute is a unit of metadata.

authentication The process of verifying a user-provided username and password.

BasePath A iAS registry entry that contains the directory where iAS is installed, including the `iAS` subdirectory (other iPlanet products can also be installed in BasePath). For instance, if you install into `/usr/local/iPlanet` on a UNIX machine, BasePath is `/usr/local/iPlanet/ias`. BasePath is a building block for AppPath.

bean property file A text file containing EJB deployment information. The type of information is defined in `javax.ejb.DeploymentDescriptor`.

bean-managed transaction See *declarative transaction*.

business logic The implementation rules determined by an application's requirements.

business method Method that performs a single business task, such as querying a database or authenticating a user, in the course of business logic.

C++ server A process in iPlanet Application Server that runs and manages C++ objects.

cached rowset A `CachedRowSet` object permits you to retrieve data from a datasource, then detach from the datasource while you examine and modify the data. A cached rowset keeps track both of the original data retrieved, and any changes made to the data by your application. If the application attempts to update the original datasource, the rowset is reconnected to the datasource, and only those rows that have changed are merged back into the database.

callable statement A class that encapsulates a database procedure or function call for databases that support returning result sets from stored procedures.

class A named set of methods and member variables that define the characteristics of a particular type of object. The class defines what types of data and behavior are possible for this type of object. Contrast with *interface.*

class file A file that contains a compiled class, usually with a `.class` extension. See also *class name, classpath.* Normally referred to in terms of its location in the filesystem, as in

```
.../com/myDomain/myPackage/myClass.
```

class loader A Java component responsible for loading Java classes, according to specific rules.

class name The name of a class in the Java Virtual Machine. See also *class file, classpath.*

classpath The path that identifies a Java class or package, in terms of its derivation from other classes or packages. See also *class file, class name.* For example,

com.myDomain.myPackage.myClass.

client An entity that invokes a resource.

client contract A contract that determines the communication rules between a client and the EJB container, establishes a uniform development model for apps that use EJBs, and guarantees greater reuse of beans by standardizing the relationship with the client. See *Enterprise JavaBean (EJB).*

cluster A set of hosts running the same server software in tandem with each other.

co-locate Positioning a component in the same memory space as a related component in order avoid remote procedure calls and improve performance.

column A field in a database table.

commit Complete a transaction by sending the required commands to the database. See *transaction*.

component A servlet, Enterprise JavaBean (EJB), or JavaServer Page (JSP).

component ACL A property in a servlet or EJB configuration file that defines that defines the users or groups that may execute.

component contract A contract that establishes the relationship between an Enterprise JavaBean (EJB) and its container. See *Enterprise JavaBean (EJB).*

configuration The process of providing metadata for a component. Normally, the configuration for a specific component is kept in a file that is uploaded into the registry when the component executes.

container A process that executes and provides services for an EJB.

context, server A programmatic view of the state of the server, represented by an object.

control descriptor A set of Enterprise JavaBean (EJB) configuration entries that enable you to specify optional individual property overrides for bean methods, plus EJB transaction and security properties.

cookie A small collection of information that can be transmitted to a calling web browser, then retrieved on each subsequent call from that browser so the server can recognize calls from the same client. Cookies are domain-specific and can take advantage of the same web server security features as other data interchange between your application and the server.

CORBA Common Object Request Broker Architecture, a standard architecture definition for object-oriented distributed computing.

data access logic Business logic that involves interacting with a datasource.

database A generic term for Relational Database Management System (RDBMS). A software package that enables the creation and manipulation of large amounts of related, organized data.

database connection A database connection is a communication link with a database or other datasource. Components can create and manipulate several database connections simultaneously to access data.

datasource A handle to a source of data, such as a database. Datasources are registered with the iAS and then retrieved programmatically in order to establish connections and interact with the datasource. A datasource definition specifies how to connect to the source of data.

declarative security Declaring security properties in the component's configuration file and allowing the component's container (i.e., a bean's container or a servlet engine) to manage security implicitly. This type of security requires no programmatic control. Opposite of *programmatic security*.

declarative transaction Declaring the transaction's properties in the bean property file and allowing the bean's container to manage the transaction implicitly. This type of transaction requires no programmatic control. Opposite of *programmatic transaction.*

deploy To create a copy of all the files in a project on one or more servers, in such a way that one or more iPlanet Application Servers and optionally one or more web servers can run the application.

deployment descriptor An attribute that determines how and where an Enterprise JavaBean (EJB) is deployed. See *Enterprise JavaBean (EJB).*

Directory Server An LDAP server that is bundled with iPlanet Application Server. Every instance of iPlanet Application Server uses Directory Server to store shared server information, including information about users and groups.

distributable session A user session that is distributable among all servers in a cluster.

distributed transaction A single transaction that can apply to multiple heterogeneous databases that may reside on separate servers.

dynamic reloading Updating and reloading a component without restarting the server. By default, servlet and JavaServer Page (JSP) components can be dynamically reloaded.

e-commerce Industry buzzword, a term meaning electronic commerce, indicating business done over the Internet.

Enterprise JavaBean (EJB) A business logic component for applications in a multitiered, distributed architecture. EJBs conform to the Java EJB standard specifications, which defines beans in terms of their expected roles. An EJB encapsulates one or more application tasks or application objects, including data structures and the methods that operate on them. Typically they also take parameters and send back return values. EJBs always work within the context of a container, which serves as a link between the EJBs and the server that hosts them. See also *container*, *session EJB*, and *entity EJB*.

entity EJB An entity Enterprise JavaBean (EJB) relates to physical data, such as a row in a database. Entity beans are long-lived, because they are tied to persistent data. Entity beans are always transactional and multiuser aware. Also see *session EJB*.

executive server Process in iPlanet Application Server that handles executive functions such as load balancing and process management.

failover recovery A process whereby a bean can transparently survive a server crash.

finder method Method which enables clients to look up a bean or a collection of beans in a globally available directory. See *Enterprise JavaBean (EJB)*.

form action handler A specially defined method in a servlet or AppLogic that performs an action based on a named button on a form.

general ACL A named list in the Directory Server that relates a user or group with one or more permissions. This list can be defined and accessed arbitrarily to record any set of permissions.

generic application A collection of globally available components, loosely organized into an application structure for configuration purposes.

generic servlet A servlet that extends `javax.servlet.GenericServlet`. Generic servlets are protocol independent, meaning that they contain no inherent support for HTTP or any other transport protocol. Contrast with *HTTP servlet*.

global database connection A database connection available to multiple component. Requires a resource manager.

global transaction A transaction that is managed and coordinated by a transaction manager and can span multiple databases and processes. The transaction manager typically uses the XA protocol to interact with the database backends. Also see *local transaction*.

group A group of users that are related in some way, maintained by a local system administrator. See also *user, role.*

GUID 128-bit hexadecimal number, guaranteed to be globally unique, used to identify components in an iAS application.

home interface A mechanism that defines the methods that enable a client to create and remove an Enterprise JavaBean (EJB). See *Enterprise JavaBean (EJB).*

HTML Hypertext Markup Language. A coding markup language used to create documents that can be displayed by web browsers. Each block of text is surrounded by codes that indicate the nature of the text.

HTML page A page coded in HTML and intended for display in a web browser.

HTTP A protocol for communicating hypertext documents across the Internet.

HTTP servlet A servlet that extends `javax.servlet.HttpServlet`. These servlets have built-in support for the HTTP protocol. Contrast with *generic servlet*.

iAS registry A collection of application metadata, organized in a tree, that is continually available in active memory or on a readily-accessible Directory Server.

iASRowSet A RowSet object that incorporates iAS extensions. The `iASRowSet` class is a subclass of `ResultSet`.

IIOP Internet Inter-Orb Protocol. Transport protocol for RMI clients and servers, based on CORBA.

inheritance A mechanism in which a subclass automatically includes the method and variable definitions of its superclass. A programmer can change or add to the inherited characteristics of a subclass without affecting the superclass.

instance An object that is based on a particular class. Each instance of the class is a distinct object, with its own variable values and state. However, all instances of a class share the variable and method definitions specified in that class.

instantiation The process of allocating memory for an object at run time. See *instance.*

interface Description of the services provided by an object. An interface defines a set of functions, called methods, and includes no implementation code. An interface, like a class, defines the characteristics of a particular type of object. However, unlike a class, an interface is always abstract. A class is instantiated to form an object, but an interface is implemented by an object to provide it with a set of services. Contrast with *class.*

isolation level (JDBC) Sets the level at which the datasource connection makes transactional changes visible to calling objects such as ResultSets.

jar file contract A contract that specifies what information must be in the Enterprise JavaBean (EJB)'s package (`jar` file). See *Enterprise JavaBean (EJB).*

JavaBean A discrete, reusable Java object.

Java server Process in iPlanet Application Server that runs and manages Java objects.

JavaServer Page (JSP) A text page written using a combination of HTML or XML tags, JSP tags, and Java code. JSPs combine the layout capabilities of a standard browser page with the power of a programming language.

JDBC Java Database Connectivity APIs. A standards-based set of classes and interfaces that enable developers to create data-aware components. JDBC implements methods for connecting to and interacting with datasources in a platform- and vendor-independent way.

JNDI Java Naming and Directory Interface. JNDI provides a uniform, platform-independent way for applications to find and access remote services over a network. iAS supports JNDI lookups for datasources and Enterprise JavaBean (EJB) components.

JTA Java Transaction API. This is an API that allows applications and J2EE servers to access transactions.

J2EE Java 2 Enterprise Edition. This is an environment for developing and deploying multi-tiered, Web-based enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing these applications.

kas See *administration server*.

kcs See *C++ server*.

kjs See *Java server*.

kxs See *executive server*.

LDAP Lightweight Directory Access Protocol. LDAP is an open directory access protocol that runs over TCP/IP. It is scalable to a global size and millions of entries. Using Directory Server, a provided LDAP server, you can store all of your enterprise's information in a single, centralized repository of directory information that any application server can access via the network.

load balancing A technique for distributing the user load evenly among multiple servers in a cluster. Also see *sticky load balancing*.

local database connection The transaction context in a local connection is not distributed across processes or across datasources; it is local to the current process and to the current datasource.

local session A user session that is only visible to one server.

local transaction A transaction that is native to one database and is restricted within a single process. Local transactions can work against only a single backend. Local transactions are typically demarcated using JDBC APIs. Also see *global transaction*.

memory cache An iAS feature that enables a servlet to cache its results for a specific duration in order to improve performance. Subsequent calls to that servlet within the duration are given the cached results so that the servlet does not have to execute again.

metadata Information about a component, such as its name, and specifications for its behavior.

package A collection of related classes that are literally packaged together in a Java archive (`.jar`) file.

parameter Parameters are name-value pairs sent from the client, including form field data, HTTP header information, etc., and encapsulated in a request object. Contrast with attribute. More generally, an argument to a Java method or database prepared command.

passivation A method of releasing an EJB's resources without destroying the bean. In this way, a bean is made to be persistent, and can be recalled without the overhead of instantiation. See *Enterprise JavaBean (EJB)*.

permission A set of privileges granted or denied to a user or group. See also *ACL*.

persistent Refers to the creation and maintenance of a bean throughout the lifetime of the application. In iAS, beans are responsible for their own persistence, called *bean-managed persistence*. Opposite of *transient*.

pooling Providing a number of preconfigured resources to improve performance. If a resource is pooled, a component can use an existing instance from the pool rather than instantiating a new one. In iAS, database connections, servlet instances, and Enterprise JavaBean (EJB) instances can all be pooled.

prepared command A database command (in SQL) that is precompiled to make repeated execution more efficient. Prepared commands can contain parameters. A prepared statement contains one or more prepared commands.

prepared statement A class that encapsulates a query, update, or insert statement that is used repeatedly to fetch data. A prepared statement contains one or more prepared command.

presentation layout Creating and formatting page content.

presentation logic Activities that create a page in an application, including processing a request, generating content in response, and formatting the page for the client.

primary key class name A variable that specifies the fully qualified class name of a bean's primary key. Used for JNDI lookups.

principal This is the identity assigned to an entity as a result of authentication.

process A sequence of execution in an active program. A process is made up of one or more threads.

programmatic security Controlling security explicitly in code rather than allowing the component's container (i.e., a bean's container or a servlet engine) to handle it. Opposite of declarative security.

programmatic transaction Controlling a transaction explicitly in code rather than allowing an Enterprise JavaBean (EJB)'s container to handle it. Opposite of declarative transaction.

property A single attribute that defines the behavior of an application component.

registration The process by which iAS gains access to a servlet, Enterprise JavaBean (EJB), and other application resource, so named because it involves placing entries in the iAS registry for each item.

remote interface Describes how clients can call a Enterprise JavaBean (EJB)'s methods. See *Enterprise JavaBean (EJB)*.

remote procedure call (RPC) A mechanism for accessing a remote object or service.

request object An object that contains page and session data produced by a client, passed as an input parameter to a servlet or JavaServer Page (JSP).

resource manager Object that controls globally-available datasources.

response object An object that references the calling client and provides methods for generating output for the client.

ResultSet An object that implements the `java.sql.ResultSet` interface. ResultSets are used to encapsulate a set of rows retrieved from a database or other source of tabular data.

reusable component A component created so that it can be used in more than one capacity, i.e., by more than one resource or application.

RMI Remote Method Invocation (RMI), a Java standard set of APIs that enable developers to write remote interfaces that can pass objects to remote processes.

role A functional grouping of subjects in an application, represented by one or more groups in a deployed environment. See also *user*, *group*.

rollback Cancel a transaction. See *transaction*.

row One single data record that contains values for each column in a table.

RowSet An object that encapsulates a set of rows retrieved from a database or other source of tabular data. RowSet extends the `java.sql.ResultSet` interface, enabling a ResultSet to act as a JavaBeans component.

security A condition whereby application resources are only used by authorized clients.

serializable An object is serializable if it can be deconstructed and reconstructed, which enables it to be stored or distributed among multiple servers.

server A computer or software package that provides a specific kind of service to client software running on other computers. A server is designed to communicate with a specific type of client software.

servlet An instance of the `Servlet` class. A servlet is a reusable application that runs on a server. In iAS, a servlet acts as the central dispatcher for each interaction in your application by performing presentation logic, invoking business logic, and invoking or performing presentation layout.

servlet engine An internal object that handles all servlet metafunctions. Collectively, a set of processes that provide services for a servlet, including instantiation and execution.

servlet runner Part of the servlet engine that invokes a servlet with a request object and a response object. See *servlet engine*.

session cookie A cookie that is returned to the client containing a user session identifier.

session EJB A session Enterprise JavaBean (EJB) relates to a unit of work, such as a request for data. Session beans are short lived—the lifespan of the client request is the same as the lifespan of the session bean. Session beans can be stateless or stateful, and they can be transaction aware. See *stateful session EJB* and *stateless session EJB*. Also see *entity EJB*.

session timeout A specified duration after which iAS can invalidate a user session. See *user session*.

SQL Structured Query Language (SQL) is a language commonly used in relational database applications. SQL2 and SQL3 designate versions of the language.

state **1.** The circumstances or condition of an entity at any given time. 2. A distributed data storage mechanism which you can use to store the state of an application using the iAS feature interface IState2.

stateful session EJB An Enterprise JavaBean (EJB) that represents a session with a particular client and which automatically maintains state across multiple client-invoked methods.

stateless session EJB An Enterprise JavaBean (EJB) that represents a stateless service. A stateless session bean is completely transient and encapsulates a temporary piece of business logic needed by a specific client for a limited time span.

sticky cookie A cookie that is returned to the client to force it to always connect to the same executive server process.

sticky load balancing A method of load balancing where an initial client request is load balanced, but subsequent requests are directed to the same process as the initial request. Also see *load balancing*.

stored procedure A block of statements written in SQL and stored in a database. You can use stored procedures to perform any type of database operation, such as modifying, inserting, or deleting records. The use of stored procedures improves database performance by reducing the amount of information that is sent over a network.

streaming A technique for managing how data is communicated via HTTP. When results are streamed, the first portion of the data is available for use immediately. When results are not streamed, the whole result must be received before any part of it can be used. Streaming provides a way to allow large amounts of data to be returned in a more efficient way, increasing the perceived performance of the application.

system administrator The person who is responsible for installing and maintaining iPlanet Application Server software and for deploying production iAS applications.

table A named group of related data in rows and columns in a database.

thread A sequence of execution inside a process. A process may allow many simultaneous threads, in which case it is multithreaded. If a process executes each thread sequentially, it is single-threaded.

transaction context A transaction's scope, either local or global. See *local transaction*, *global transaction*.

transaction manager Object that controls a global transaction, normally using the XA protocol. See *global transactions.*

transaction A set of database commands that succeed or fail as a group. All the commands involved must succeed for the entire transaction to succeed.

transient A resource that is released when it is not being used. Opposite of *persistent*.

URI Universal Resource Identifier, describes specific resource at a domain. Locally described as a subset of a base directory, so that `/ham/burger` is the base directory and a URI specifies `toppings/cheese.html`. A corresponding URL would be `http://`*`domain`*`:`*`port`*`/toppings/cheese.html`.

URL Uniform Resource Locator. An address that uniquely identifies an HTML page or other resource. A web browser uses URLs to specify which pages to display. A URL describes a transport protocol (e.g. HTTP, FTP), a domain (e.g. `www.my-domain.com`), and optionally a URI.

user A person who uses your application. Programmatically, a user name, password, and set of attributes that enables an application to recognize a client. See also *group, role*.

user interface (UI) The pages that define what a user sees and with which a user interacts in a web application.

user session A series of user-application interactions that are tracked by the server. Sessions maintain user state, persistent objects, and identity authentication.

versioning See *dynamic reloading.*

web application A computer program that uses the World Wide Web for connectivity and user interface (UI). A user connects to and runs a web application by using a web browser on any platform. The user interface of the application is the HTML pages displayed by the browser. The application itself runs on a web server and/or application server.

web browser Software that is used to view resources on the World Wide Web, such as web pages coded in HTML or XML.

web connector plug-in An extension to a web server that enables it to communicate with a iPlanet Application Server.

web server A host that stores and manages HTML pages and web applications. The web server responds to user requests from web browsers.

XA protocol A database industry standard protocol for distributed transactions.

XML XML, the Extensible Markup Language, uses HTML-style tags to identify the kinds of information used in documents as well as to format documents.

Index