Developer's Guide (Java™)

iPlanet™ Application Server

Version 6.0

816-1678-01
June 2001

Contents

Preface

The *iPlanet Application Server Developer's Guide (Java™)* describes how to create and run Java 2 Platform, Enterprise Edition (J2EE) applications that follow the new open Java standards model for Servlets, Enterprise JavaBeans (EJBs), JavaServer Pages (JSPs), and Java Database Connectivity (JDBC) on the iPlanet Application Server.

This guide is intended for information technology developers in a 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, reference material, and a glossary.

This preface contains information about the following topics:

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

Using the Documentation

Table 1 lists the tasks and concepts that are described in the iPlanet Application Server printed manuals and online *Release Notes*. If you are trying to accomplish a specific task or learn more about a specific concept, refer to the appropriate guide.

Note that the printed guides are also available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML) formats, at `http://docs.iplanet.com/docs/manuals/ias.html`.

Table 1 iPlanet™ Application Server Documentation Roadmap

For information about	See the following	Shipped with
Late-breaking information about the software and the documentation	*Release Notes*	iPlanet Application Server 6.0
Installing iPlanet Application Server and its various components (Web Connector plug-in, iPlanet Application Server Administrator), and configuring the sample applications	*Installation Guide*	iPlanet Application Server 6.0
Creating iPlanet Application Server 6.0 applications that follow the 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	*Developer's Guide (Java)*	iPlanet Application Server 6.0

Table 1 iPlanet™ Application Server Documentation Roadmap

For information about	See the following	Shipped with
Administering one or more application servers using the iPlanet Application Server Administrator Tool to perform the following tasks: • Monitoring and logging server activity • Implementing security for Netscape Application Server • Enabling high availability of server resources • Configuring web-connector plugin • Administering database connectivity • Administering transactions • Configuring multiple servers • Administering multiple-server applications • Load balancing servers • Managing distributed data synchronization • Setting up Netscape Application Server for development	*Administrator's Guide*	iPlanet Application Server 6.0
Migrating your applications to the new iPlanet Application Server 6.0 programming model from the Netscape Application Server version 2.1, including a sample migration of an Online Bank application provided with iPlanet Application Server	*Migration Guide*	iPlanet Application Server 6.0
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)*	iPlanet Application Server 6.0
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

What You Should Already Know

This guide assumes you are familiar with the following topics:

- J2EE specification
- HTML
- Java programming
- Java APIs as defined in specifications for EJBs, JSPs, 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

The first part of this guide provides an iPlanet Application Server environment overview for designing programs. This part includes the following topic:

- Chapter 1, “Developing 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 Entity EJBs”
- Chapter 7, “Handling Transactions with EJBs”
- Chapter 8, “Using JDBC for Database Access”
- Chapter 9, “Developing and Deploying RMI/IIOP-Based Clients”

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

- Chapter 10, “Deployment Packaging”
- Chapter 11, “Creating and Managing User Sessions”
- Chapter 12, “Writing Secure Applications”
- Chapter 13, “Taking Advantage of the iPlanet Application Server 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 Microsoft Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except that forward slashes are used to separate directories.

This guide uses URLs of the form:

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

In these URLs, *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server’s directory structure; 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 code variables.
- *Italic* type is also used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
- **Bold** type is used as either a paragraph lead-in or to emphasis words used in the literal sense.

Related Information

You can find a directory of URLs for the official specifications at *install_dir*`/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 is `http://www.servletcentral.com`.

Programming with EJBs:

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

The web site is `http://www.oreilly.com/catalog/entjbeans2/`.

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

Developing Applications

This chapter summarizes the iPlanet ™ Application Server application design process and offers effective development guidelines.

This chapter contains the following sections:

- Application Requirements
- About the Application Programming Model
- Effective iPlanet Application Guidelines

Application Requirements

When developing an iPlanet Application Server application, start by identifying the application requirements. Typically, this means developing a distributed application as a widely deployable application that is fast and secure, and that can reliably handle additional requests as new users are added.

The iPlanet Application Server meets these needs because it supports the J2EE APIs as well as a set of pre-existing high performance features. For example, for an online banking application, you can deliver:

- High performance
- Scalability
- Rapid deployment
- Security
- Rapid deployment of specific features; for example, account transfers, account reporting, online trades, special offers to qualified customers

- Management and administration of different types of end users; for example, individuals, corporations, or internal users
- Internal reporting
- Enterprise Information System (EIS) connectivity; that provides access to information stored in legacy databases

About the Application Programming Model

A distributed application model allows different individual application areas to focus on different functional elements, thereby improving performance. For instance, designing security requirements may affect one or more application model layers.

In the presentation layer, you may need to check a user's identity so your application could present one set of pages for anonymous users and another set for registered users. Additionally, the application may present a page explaining why the attempt to use a restricted feature 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 end user category.

The Presentation Layer

The presentation layer is where the user interface is dynamically generated. An application may require the following application elements:

- Servlets
- JSPs
- HTML pages
- Client side JavaScript elements

Servlets

Servlets handle the application's presentation logic. Servlets are the page-to-page navigation dispatchers, and they also provide session management and simple input validation. Servlets tie business logic elements together.

A servlet developer must understand programming issues related to HTTP requests, security, internationalization, and web statelessness (such as sessions, cookies, and time-outs). For an iPlanet ™ Application Server application, servlets must be written in Java. Servlets are likely to call JSPs, EJBs, and JDBC objects. Therefore, a servlet developer works closely with the application element developers.

JSPs

JSPs handle most application display tasks, and they work in conjunction with servlets to define the application's presentation screens and page navigation. JSPs are likely to call EJBs and JDBC objects. The EJBs typically encapsulate business logic functionality. As such, they carry out calculations and other repetitively requested tasks. JDBC objects are used to connect to databases, make queries, and return query results.

HTML Pages

Properly designed HTML pages provide:

- Uniform appearance across different browsers.
- Efficient HTML loading across slow modem connections.
- Dynamically generated page appearances that are JSP dispatched.

Client-Side JavaScript

Client-side JavaScript can also be used to handle such things as simple input validation before passing data to the server, or to make the user interface more exciting. Client-side JavaScript developers work closely with servlet and JSP developers.

The Business Logic Layer

The business logic layer typically contains deployed entities that encapsulate business rules and other business functions in:

- Session beans

- Entity beans

Session Beans

Session beans encapsulate the business processes and rules logic. For example, a session bean could calculate taxes for a billing invoice. When there are complex business rules that change frequently (for example, due to new business practices or new government regulations), an application typically uses more session beans than entity beans, and session beans may need continual revision.

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 taxes are calculated in a stateful session bean. The application must access a specific server where the bean's state information resides. If the server happens to be down the application processing is delayed.

Entity Beans

Entity beans represent persistent objects, such as a database row. Entity beans are likely to call a full range of JDBC interfaces. However, entity beans typically do not call other EJBs. The entity bean developer's role 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 session bean and servlet developers to ensure that the application provides fast, scalable access to persistent business data.

The Data Access Layer

In the Data Access layer, custom connectors work with the iPlanet ™ Application Server Unified Integration Framework (UIF) to enable communication with legacy EISs, such as IBM's CICS.

Connector developers are most likely to use C++ and typically need to understand issues related to wrapping C++ in Java, such as Java Native Interfaces (JNI), as well as UIF.

UIF is an API framework, that enables the application server to pass information to an EIS database. These developers are likely to integrate access to the following systems:

- CORBA applications

- Mainframe systems
- Third-party security systems

For more information about UIF, see the *iPlanet Unified Integration Framework Developer's Guide* and the release notes at the following URL:

`http://docs.iplanet.com/docs/manuals/ias.html#uifsp1`

Effective iPlanet Application Guidelines

This section lists guidelines to consider when designing and developing an iPlanet ™ Application Server application, and is merely a summary. For more details, refer to later chapters in this guide.

The guidelines are grouped into the following goals:

- Presenting Data with Servlets and JSPs
- Creating Reusable Application Code
- Improving Performance
- Scalability Planning

Presenting Data with Servlets and JSPs

Servlets are often used for presentation logic and serve as central dispatchers of user input and data presentation. JSPs are used to dynamically generate the presentation layout. Both servlets and JSPs can be used to conditionally generate different pages.

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

For example, after an Online Bookstore application authenticates a user, it provides a boilerplate *portal* front page for the user to choose one of several tasks, including a book search, purchase selected items, and so on. Since this portal conducts little or no processing, it can 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.

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

Creating Reusable Application Code

Aside from using good object-oriented design principles, there are several things to consider when developing an application to maximize reusability, including the following tips:

- Use relative paths and URLs so links remain valid if the code tree moves.
- Minimize Java in 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 hard-coded strings such as the datasource names, 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 UIF-based connectors to access legacy data.

Improving Performance

Here are several tips to improve your application’s performance when it is deployed on an iPlanet ™ Application Server:

- In most cases, deploy servlets and JSPs to the iPlanet ™ Application Server rather than to the iPlanet Web Server. iPlanet ™ Application Server is best if an application is highly transactional, requires failover support to preserve session data, or accesses legacy data. The iPlanet Web Server is useful if an application is mostly stateless, read-only, and non-transactional.

- Use entity beans and stateless session beans; design for co-location to avoid time intensive 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 information rows, 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 Java performance.

Scalability Planning

To plan an application to easily scale as customer demand increases:

- Develop your application so that it stores scaling or serializing information in `HttpSession` objects that are configured for distribution.
- Avoid using global variables.
- Design an application to run in a multi-machine server farm environment.

Chapter 2

Controlling Applications with Servlets

This chapter describes how to create effective servlets to control application interactions running on an iPlanet Application Server, including standard servlets. In addition, this chapter describes the iPlanet Application Server features to use to augment the standards.

This chapter contains the following sections:

- About Servlets
- About the Server Engine
- Designing Servlets
- Creating Servlets
- Invoking Servlets

About Servlets

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

Servlets supported by the iPlanet Application Server are based on the Java Servlet Specification v2.2. All relevant specifications are accessible from *install_dir*`/ias/docs/index.htm`, where *install_dir* is the directory where the iPlanet Application Server is installed.

Servlets are used for an application's presentation logic. A servlet acts as an application's central dispatcher by processing form input, invoking business logic components encapsulated in EJBs, and formatting web page output using JSPs. Servlets control the application flow from one user interaction to the next by generating content in response to user requests.

The fundamental characteristics are:

- Servlets are created and managed at runtime by the iPlanet Application Server servlet engine.
- Servlets operate on input data that is encapsulated in a `request` object.
- Servlets respond to a query with data encapsulated in a `response` object.
- Servlets call EJBs to perform business logic functions.
- Servlets call JSPs to perform page layout functions.
- Servlets are extensible; use the APIs provided with the iPlanet Application Server to add functionality.
- Servlets provide user session information persistence between interactions.
- Servlets can be part of an application or they can reside discretely on the application server so they are available to multiple applications.
- Servlets can be dynamically reloaded while the server is running.
- Servlets are addressable with URLs; buttons on an application's pages often point to servlets.
- Servlets can call other servlets.

Several iPlanet Application Server API features enable an application to take programmatic advantage of specific iPlanet features. For more information, see "iPlanet Application Server Optional Features," on page 47.

Servlet Data Flow

When a user clicks a Submit button, information entered in a display page is sent to a servlet. The servlet processes the incoming data and orchestrates a response by generating content, often through business logic components, which are EJBs. Once the content is generated, the servlet creates a response page, usually by forwarding the content to a JSP. The response is sent back to the client, which sets up the next user interaction.

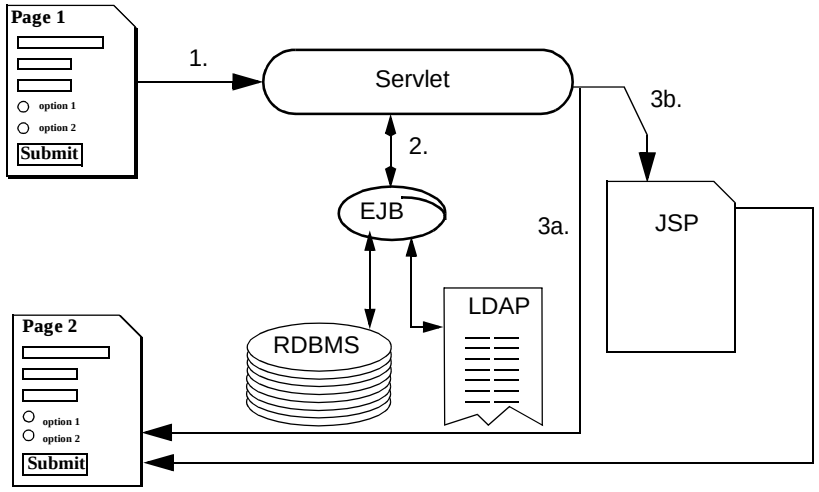
The following illustration shows the information flow to and from the servlet, as:

1. Servlet processes the client request
2. Servlet generates content
3. Servlet creates response and either:

a. Sends it back directly to the client

or

b. Dispatches the task to a JSP



The servlet remains in memory, available to process another request.

Servlet Types

There are two main servlet types:

- Generic servlets
  - Extend `javax.servlet.GenericServlet`.
  - Are protocol independent; they contain no inherent HTTP support or any other transport protocol.
- HTTP servlets
  - Extend `javax.servlet.HttpServlet`.
  - Have built-in HTTP protocol support and are more useful in an iPlanet Application Server environment.

For both servlet types, implement the constructor method `init()` and the destructor method `destroy()` to initialize or deallocate resources, respectively.

All servlets must implement a `service()` method, which is responsible for handling servlet requests. For generic servlets, 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, override `doPost()` to process `POST` requests, `doGet()` to process `GET` requests, and so on.

About the Server Engine

Servlets exist in a Java server process on an iPlanet Application Server and are managed by the servlet engine. The servlet engine is an internal object that handles all servlet meta functions. These functions include instantiation, initialization, destruction, access from other components, and configuration management.

Instantiating and Removing Servlets

After the servlet engine instantiates the servlet, the servlet engine runs its `init()` method to perform any necessary initialization. Override this method to perform an initialize a function for the servlet's life, such as initializing a counter.

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

Request Handling

When a request is made, the iPlanet Application Server hands the incoming data to the servlet engine. The servlet engine processes the request's input data, such as form data, cookies, session information, and URL name-value pairs, into an `HttpServletRequest` request object type.

The servlet engine also captures client metadata by encapsulating it in an `HttpServletResponse` response object type. The engine then passes both as parameters to the servlet's `service()` method.

In an HTTP servlet, the default `service()` method routes requests to another method based on an HTTP transfer method, such as `POST`, `GET`, and so on. For example, HTTP `POST` requests are sent to the `doPost()` method, HTTP `GET` requests are sent to the `doGet()` method, and so on. This enables the servlet to process request data differently, depending on which transfer method is used. Since the routing takes place in the service method, you generally do not override `service()` in an HTTP servlet. Instead, override `doGet()`, `doPost()`, and so on, depending on the request type you expect.

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

To perform the tasks to answer a request, override the `service()` method for generic servlets, and the `doGet()` or `doPost()` methods for HTTP servlets. Very often, this means accessing EJBs to perform business transactions, collating the 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 user.

Allocating Servlet Engine Resources

By default, the servlet engine creates a thread for each new request. This is less resource intensive than instantiating a new servlet copy in memory for each request. Avoid threading issues, since each thread operates in the same memory space where variables can overwrite each other.

If a servlet is specifically written as a single thread, the servlet engine creates a pool of ten servlet instances 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 pool instances is configurable in the Deployment Descriptor (DD), which is an iPlanet Application Server specific XML file. For more information about deployment descriptors, see Chapter 10, “Deployment Packaging.”

For more information on threading issues, see “Handling Threading Issues,” on page 43.

Dynamically Reloading Servlets at Runtime

If no configuration file changes are needed, servlet reloading in an iPlanet Application Server is done without restarting the server by simply redeploying the servlet. The iPlanet Application Server *notices* the new component and reloads it within 10 seconds. For more information, see Appendix B, "Dynamic Reloading."

NOTE This feature is turned off by default for a production environment. Turn it on when needed.

Configuring Servlets for Deployment

When you configure a servlet for deployment, you actually provide the metadata, which the application server uses to create the servlet object and use it in the application framework. For more information about servlet configuration, see Chapter 10, "Deployment Packaging."

Locating Servlet Files

Servlet files and other application files reside in a directory structure location known to the iPlanet Application Server as `AppPath`. This variable defines the top of a logical directory tree for the application. The `AppPath` variable is similar to the document path in a web browser. By default, `AppPath` contains the value *BasePath*`/APPS`, where *BasePath* is the base iPlanet Application Server directory.

`AppPath` and `BasePath` are variables held in the iPlanet Application Server registry, which is a repository for server and application metadata. For more information, see "The iPlanet Application Server Registry," on page 233 and the *Deployment Tool Online Help*.

In addition to `AppPath` and `BasePath`, the registry has a third variable called `ModulesDirName`. This variable corresponds to a directory under `AppPath` that is the home for web modules that do not exist as a part of any J2EE application. They are registered as standalone modules.

Table 2-1 describes important files and servlet locations:

Table 2-1 Important Files and Servlet Locations

Location Variable	Description
`BasePath`	Top of the iPlanet Application Server tree. All files in this directory are part of the iPlanet Application Server. Defined by the registry variable `BasePath`.
`AppPath`	Top of the application tree. Applications reside in subdirectories of this location. Defined by the registry variable `AppPath`.
`ModulesDirName`	A special directory that contains all J2EE web and EJB modules that are registered as stand-alone entities (in the `Default` application). This directory exists under *AppPath*. Defined by the registry variable `ModulesDirName`. The default value of this variable in the registry is `modules`.
AppPath`/`*appName*`/*`	Top of the subtree for the application *appName*. The *appName* directory in turn contains subdirectories for different modules within the application. For more information, see "Invoking Servlets," on page 48.

Deploying Servlets

You normally deploy servlets with the rest of an application using the iPlanet Application Server Deployment Tool. You can also deploy servlets manually for testing or to update servlets while the server is running. For more information, see the *Deployment Tool Online Help*.

Designing Servlets

This section describes basic design decisions to make when planning the servlets that help make up an application.

Web applications generally follow a request-response paradigm so that 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 determines how to design each servlet by defining the required input and output parameters for each interaction.

Choosing a Servlet Type

Servlets that extend `HttpServlet` are much more useful in an HTTP environment, since that is what they were designed for. We recommend that all iPlanet Application Server servlets extend `HttpServlet` rather than `GenericServlet` to take advantage of the built-in HTTP support. For more information, see “Servlet Types,” on page 31.

Create Standard or Non-Standard Servlets

One important decision 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 to utilize the features provided in the iPlanet Application Server APIs. These APIs can greatly increase the usefulness of servlets in an iPlanet Application Server framework.

You can also create portable servlets that only take advantage of iPlanet Application Server features if the servlet runs in an iPlanet Application Server environment.

For more information on iPlanet Application Server specific APIs, see “iPlanet Application Server Optional Features,” on page 47.

Planning for Servlet Reuse

Servlets by definition are discrete, reusable applications that run on a server. A servlet does not necessarily have to be tied to one application. You can create a servlet library to be used across multiple applications by placing it in the application named `Default`.

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 a specific application.

Creating Servlets

To create a servlet, perform the following tasks:

- Design the servlet into your application, or, if accessed in a generic way, design it to access no application data.

- Create a class that extends either `GenericServlet` or `HttpServlet`, overriding the appropriate methods so it handles requests.
- Use the iPlanet Application Server Administration Tool to create a web application Deployment Descriptor (DD) for the servlet.

Servlet Files for an iPlanet Application

The files that make up a servlet include:

- The Servlet's Class File
- The Servlet's Deployment Descriptor
- iPlanet Application Server Optional Features

The Servlet's Class File

This section describes how to write a servlet, including the decisions to make about an application and the 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 iPlanet Application Server servlets exist in an HTTP environment, the latter class is recommended. If the servlet is part of a package, you must also declare the package name so the class loader can properly locate it.

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

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

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

Overriding Methods

Next, override one or more methods to provide servlet instructions to perform its intended task. All processing by a servlet is done on a request-by-request basis and happens in the service methods, either `service()` for generic servlets or one of the do*Operation*`()` methods for HTTP servlets. This method accepts incoming requests, processing them 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 database access to perform a transaction or passing the request to an EJB.

Overriding Initialize

Override the class initializer `init()` to initialize or allocate resources for the servlet instance's life, 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)` to set their scope. This makes the servlet's configuration object available to other servlet methods.

The following example of the `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 servlet methods can access the variable.

Overriding Destroy

Override the class destructor `destroy()` to write log messages or to release resources that are not 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 the following, based on the example for "Overriding Initialize" above:

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

Overriding Service, Get, and Post

When a request is made, the iPlanet Application Server hands the incoming data to the servlet engine to process the request. The request includes form data, cookies, session information, and URL name-value pairs, all in a type `HttpServletRequest` object called the request object. Client metadata is encapsulated as a type `HttpServletResponse` object called the response object. The servlet engine passes both objects as the servlet's `service()` method parameters.

The default `service()` method in an HTTP servlet routes the request to another method based on the HTTP transfer method (POST, GET, and so on). 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 request data processing depending on the transfer method. Since the routing takes place in `service()`, there is no need to generally override `service()` in an HTTP servlet. Instead, override `doGet()`, `doPost()`, and so on, depending on the expected request type.

The automatic routing in an HTTP servlet is based simply on a call to `request.getMethod()`, which provides the HTTP transfer method. In an iPlanet Application Server, request data is already preprocessed into a name-value list by the time the servlet sees the data, so simply overriding the `service()` method in an HTTP servlet does not lose any functionality. However, this does make the servlet less portable, since it is now dependent on preprocessed request data.

Override the `service()` method (for generic servlets) or the `doGet()` and/or `doPost()` methods (for HTTP servlets) to perform 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 result set 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 implementing both methods to provide for both input types or simply pass the request object to a central processing method, as shown in the following example:

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

All 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 iPlanet Application Server features.

If a servlet intends to call the `RequestDispatcher` method `include()` or `forward()`, be aware the request information is no longer sent as HTTP POST, GET, and so on. In other words, if a servlet overrides `doPost()`, it may not process anything if another servlet calls it, if the calling servlet happens to receive its data through HTTP GET. For this reason, be sure to implement routines for all possible input types, as explained above. `RequestDispatcher` methods always call `service()`.

For more information, see "Calling a Servlet Programmatically," on page 50.

NOTE Arbitrary binary data, such as 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 these kinds of data and package them correctly in the request object.

Accessing Parameters and Storing Data

Incoming data is encapsulated in a request object. For HTTP servlets, the request object type is `HttpServletRequest`. For generic servlets, the request object type is `ServletRequest`. The request object contains all request parameters, including your own request values called *attributes.*

To access all incoming request parameters, use the `getParameter()` method. For example:

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

Set and retrieve values in a request object using `setAttribute()` and `getAttribute()`, respectively. For example:

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

This shows 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 JavaBeans," on page 77.

Handling Sessions and Security

From a web or application server's perspective, a web application is a series of unrelated server hits. There is no automatic recognition if a user has visited the site before, even if their last interaction were seconds before. A session provides a context between multiple user interactions by remembering the application state. Clients identify themselves during each interaction by 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."

After a successful login, you should 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 additional information to retain. Application components can then query the session object to obtain user authentication.

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

Accessing Business Logic Components

In the iPlanet Application Server programming model, you implement business logic, including database or directory transactions and complex calculations, in EJBs. A `request` object reference can be passed as an EJB parameter to perform the specified task.

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

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

This example shows a servlet accessing an EJB called `ShoppingCart`. The servlet creates a cart handle 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) {
```

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

Access EJBs from servlets by using the Java Naming Directory Interface (JNDI) to establish a handle, or proxy, to the EJB. Next, refer to the EJB as a regular object; overhead is managed by the bean's container.

This example shows JNDI looking 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 servlet instance are usually executed numerous times simultaneously (up to the available memory limit). Each execution occurs in a different thread though only one servlet copy exists in the servlet engine.

This is efficient system resource usage, but is dangerous because of how Java manages memory. 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 write access to all instance variables, as in `public synchronized void` *`method`*`()` (whole method) or `synchronized(this) {...}` (block only). Because synchronizing slows response time considerably, synchronize only blocks, or make sure that the blocks in the servlet do not need synchronization.

  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
   }
}
```

- Use `SingleThreadModel` to create a single-threaded servlet. In this case, when a single-threaded servlet is registered with the iPlanet Application Server, the servlet engine creates a 10 servlet instance pool used for incoming requests (10 copies of the same servlet in memory). The number of servlet instances in the pool is changed by setting the `number-of-singles` element in the iPlanet Application Server specific web application DD to a different number. The iPlanet Application Server Deployment Tool is used to modify this number in the iPlanet Application Server specific web application DD. For more information on the iPlanet Application Server web application DD, see Chapter 10, "Deployment Packaging," the iPlanet Application Server Deployment Tool, and the *Administrator's Guide*. A single-threaded servlet is slower under load because new requests must wait for a free instance in order to proceed, but this is not 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 Client Results

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

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

Creating a Servlet Response Page

Generate the output page within a servlet by writing to the output stream. The recommended way to do this depends on the output type.

Always specify the output MIME type using `setContentType()` before any 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, write to the output stream directly by creating a `ServletOutputStream` object and then write to it using `print()`. For example:

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

NOTE A servlet cannot call a JSP from a `PrintWriter` or `ServletOutputStream` object.

NOTE If you use the iPlanet Application Server with the iPlanet Web Server, do not set the date header in the output stream using `setDateHeader()`. This results in a duplicate date field in the response page's HTTP header the server returns to the client. This is because the iPlanet Web Server automatically provides a header field. Conversely, Microsoft Internet Information Server (IIS) does *not* add a date header, so one must be provided.

Creating a JSP Response Page

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 using `include()`:

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

- The `forward()` method in the `RequestDispatcher` interface hands the JSP interaction control. The servlet is no longer involved with the current interaction's output after invoking `forward()`, thus only one call to the `forward()` method can be made in a particular servlet.

NOTE You cannot use the `forward()` method if you have already defined a `PrintWriter` or `ServletOutputStream` object.

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

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

NOTE Identify which JSP to call by specifying a Universal Resource Identifier (URI). The path is a `String` describing a path within the `ServletContext` scope and must begin with a forward slash (“`/`”). There is also a `getRequestDispatcher()` method in the request object that takes a `String` argument indicating a complete path. For more information about this method, see the Java Servlet Specification, v2.2, section 8.

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

The Servlet’s Deployment Descriptor

Servlet DDs are created by the iPlanet Application Server Deployment Tool (you can also create them by hand). These descriptor files are packaged within Web Application aRchive (`.war`) files. They contain metadata, plus information that identifies the servlet and establishes its application role.

The sample applications for iPlanet Application Server contain instructions for creating servlet DDs. These sample applications are located in the *install_dir*`/ias/ias-samples` directory.

Elements

The DD for a servlet contains standard J2EE specified elements as well as iPlanet Application Server specific elements. The servlet DDs convey the elements and configuration information of a web application between developers, assemblers, and deployers. For more information about these elements, see Chapter 10, “Deployment Packaging.”

Changing Configuration Files

To modify deployment descriptor settings, you can use the Deployment Tool or a combination of an editor and command line utilities such as Ant to reassemble and deploy the updated deployment descriptor information.

Using the Deployment Tool

1. Open the EAR, WAR or EJB JAR file.
2. Modify the deployment descriptors.
3. Redeploy the EAR, WAR or EJB JAR module.
4. Restart the application server to pick up the modified deployment descriptor settings.

Using the Command Line

Refer to the sample applications (in the *install_dir*`/ias/ias-samples` directory) for extensive examples of using Ant-based `build.xml` files to reassemble applications and modules from the command line.

1. Edit the appropriate deployment descriptor file (`web.xml` or `ias-web.xml`) by hand.
2. Execute an Ant build command (such as `build war`) to reassemble the appropriate EAR, WAR or EJB JAR module.
3. Use the `iasdeploy` command to deploy the EAR or WAR file or module.
4. Restart the application server to pick up the modified deployment descriptor settings.

iPlanet Application Server Optional Features

Many additional iPlanet features augment servlets for use in an iPlanet Application Server environment. These features are not a part of the official specifications, though some are based on emerging Sun standards and conform to future standards.

For more information on the iPlanet Application Server features, see Chapter 13, “Taking Advantage of the iPlanet Application Server Features.”

The iPlanet Application Server provides support for more robust sessions, based on a previous version model of the iPlanet Application Server. 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

Invoke a servlet by either directly addressing it from an application page with a URL or calling it programmatically from an already running servlet. You can also Verify servlet parameters. See the following sections:

- Calling a Servlet With a URL
- Calling a Servlet Programmatically
- Verifying Servlet Parameters

Calling a Servlet With a URL

Most times, you call servlets by using URLs embedded as links in the 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 responding to a request has several sections. Each section has to locate the appropriate servlet. The request object exposes the following elements when obtaining the request's URI path:

- Context Path
- Servlet Path
- PathInfo

For more information on these elements, see the Java Servlet Specification, v2.2, section 5.4.

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

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

Table 2-2 describes each URL section.

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

URL element	Description
server`:`*port*	The address and optional web server port number handling the request.
`NASApp`	Indicates to the web server that this URL is for an iPlanet Application Server application. The request routes to the iPlanet Application Server executive server.

Table 2-2 URL Fields for Servlets within a Specific Application *(Continued)*

URL element	Description
moduleName	The servlet module name (these names are unique across the server). The *moduleName* corresponds to a directory under *AppPath/applicationName* for a module that is registered as part of an application. It reflects the `.war` module name that contains the servlets and JSPs, and its contents are the same as those of the `.war` module.
servletName	The servlet name as configured in the XML file.

For example:

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

Invoking Generic Application Servlets

Address servlets that are part of the generic `Default` application as follows:

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

Table 2-3 describes each URL section.

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

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

For example:

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

NOTE All servlets deployed to use the `/servlet` path must be deployed with the application name `Default`. Additionally, the servlet engine of the web server instance must be deactivated in order to pass the requests started with `/servlet` to the iPlanet Application Server.

Calling a Servlet Programmatically

First, identify which servlet to call by specifying a URI. This is normally a path relative to the current application. For instance, if your servlet is part of an application with a context root called `Office`, the URL to a servlet called `ShowSupplies` from a browser is as follows:

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

You can call this servlet programmatically from another servlet in one of two ways, as described below.

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

  For example:

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

- To hand interaction control to another servlet, use the `RequestDispatcher` interface's `forward()` method with the servlet's URI as a parameter.

NOTE Forwarding a request means the original servlet is no longer involved with the current interaction output after `forward()` is invoked. Therefore, only one `forward()` call can be made in a particular servlet.

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

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

NOTE Both servlet invoking mechanisms, either programmatic (using `include()` or `forward()`) or from the URL, can use URL patterns for the servlet specified in the DD XML file or the `<servlet-name>` entry. For example, if the XML entry in the `web.xml` file is:

```
<servlet-name>Fortune</ servlet-name>
<servlet-mapping>
<servlet-name>Fortune</ servlet-name>
<url-pattern>Business</ url-pattern>
</servlet-mapping>
```

You can access the servlet in either of the following ways:

- `http://`*`server`*`:`*`port`*`/NASApp/`*`context_root`*`/Fortune`
- `http://`*`server`*`:`*`port`*`/NASApp/`*`context_root`*`/Business`

Verifying Servlet Parameters

You can verify the parameters passed to a servlet. This feature can increase iPlanet Application Server response time and save development time.

iPlanet Application Server can call a specified class for parameter verification. Based on the results of the verification, the server can either call the servlet method or abort the call to the servlet, redirecting the user to an error page. You must provide the parameter verification class and specify it in the Deployment Tool during servlet deployment. You can specify which parameters are validated.

The parameter verification code need not be present within the servlet. If more than one servlet accepts the same parameter, they must both use the same parameter verification function for that parameter.

In the IAS Params tab of the servlet descriptor in the Deployment Tool, you can specify the following for each parameter:

- The name of the parameter
- Whether verification is required
- The class and method to call for verification
- The format of the parameter
- The parameter's scope
- The error page to display in case of an error

Chapter 3

Presenting Application Pages with JavaServer Pages

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

This chapter contains the following sections:

- Introducing JSPs
- How JSPs Work
- Designing JSPs
- Creating JSPs
- Programming Advanced JSPs
- Deploying JSPs
- Invoking JSPs
- JSP 1.1 Tag Summary
- Modifying Custom Tags for JSP 1.1
- Compiling JSPs: The Command-Line Compiler
- Value-added Features

Introducing JSPs

JSPs are browser pages in HTML or XML. They also contain Java code, which enables them to perform complex processing, conditionalize output, and communicate with other application objects. JSPs in iPlanet Application Server are based on the JSP 1.1 specification. This specification is accessible from *install_dir*`/ias/docs/index.htm`; *install_dir* is where the iPlanet Application Server is installed.

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

How JSPs Work

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

The iPlanet Application Server compiles JSPs into HTTP servlets the first time they are called. This makes them available to the application environment as standard objects and 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 an output page and is returned to the caller.

The response object contains a calling client reference; this is where a JSP presents the page it creates. If a JSP is called from a servlet using the `RequestDispatcher` interface's `forward()` method, `forward()` provides the response object as a JSP parameter. If a JSP is invoked directly from a client, the server managing the relationship with the caller provides the response object.

In either case, the page is automatically returned to the client through the response object reference without any additional programming.

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

JSPs and other application components can be updated at runtime without restarting the server, making it easy to change an application's look and feel without stopping service. For more information, see Appendix B, "Dynamic Reloading."

Designing JSPs

This section describes decisions to consider when writing JSPs. Since JSPs are compiled into servlets, servlet design considerations are also relevant to JSPs. For more information about design considerations for servlets, see Chapter 2, "Controlling Applications with Servlets."

A page's information can loosely be categorized into page layout elements, which consist of tags and information pertaining to the page structure, and page content elements, which consist of the actual page information sent to the user.

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

Since page layout is more or less a straightforward task, the design decisions must relate to the way the JSP interacts with the application and how it is optimized.

This section contains the following subsections:

- Choosing a Component
- Designing for Ease of Maintenance
- Designing for Portability
- Handling Exceptions

Choosing a Component

The first task is to decide on a JSP or a servlet. If the main feature is the page layout with little processing involved for page generation, 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. Servlets are strong in processing and adaptability, and since they are Java files, you can take advantage of integrated development environments while 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 is awkward. Choose the right component for the job at hand.

For example, the following component is presented as both a JSP and a servlet for comparison. This component performs no complex content generation activities, and 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>Feedback</title></head>"
                + "<body>\n"
                + "<h1>The name you typed is:"
                + req.getParameter("name") + ".</h1>"
                + "</body></html>";
   }
}
```

For more information about servlets, see Chapter 2, “Controlling Applications with Servlets.”

Designing for Ease of Maintenance

Each JSP can call or include any other JSP. For example, you can create a generic corporate banner, a standard navigation bar, and a left-side column table of contents, where each element is in a separate JSP and is included for each page built. The page can be constructed with a JSP functioning as a frameset, dynamically determining the pages to load into each subframe. A JSP can also be included when the JSP is compiled into a servlet or when a request arrives.

Designing for Portability

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

One possible use for generic JSPs is for portable page elements, such as navigation bars or corporate headers and footers, which are meant to be included in other JSPs. You can create a library of reusable generic page elements to use throughout an 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 that makes a change to the page's structure based on a standard value set in the request object.

Handling Exceptions

If an uncaught exception occurs in a JSP file, iPlanet Application Server generates an exception, usually a `404` or `500` error. To avoid this problem, set the `errorPage` attribute of the `<%@ page%>` tag.

Creating JSPs

JSPs are created in basically the same way as static HTML files are. You can use an HTML editor to create pages and edit the page layout. You make a page a JSP by inserting JSP-specific tags into the raw source code where needed.

The following sections describe how to use JSP-specific tags in HTML files to create JSPs, including JSP elements, directive elements, scripting elements, and action elements.

This section contains the following subsections:

- General Syntax
- Directives
- Scripting Elements
- 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 needed, in the same way as standard HTML tags. For example, if the request contains a parameter name that contains the user name, a welcome sentence could look like this:

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

JSP Tags

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

NOTE These shortcuts are not valid for XML files.

Empty elements or tag constructs that have nothing between the start and end tags can be shortened to one tag ending with />. Some examples:

An include tag with no body:

```
<jsp:include page="/corporate/banner.jsp"></jsp:include>
```

A shorter form of an include tag with no body:

```
<jsp:include page="/corporate/banner.jsp" />
```

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

Escape Characters

Attributes in which difficulty with nested single and double quotes exist use the following escape characters:

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

Comments

There are two JSP comment types:

- JSP page comments that document what the page is doing
- Document generated comments 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 Client Output

Use the HTML and XML comment syntax to generate comments to the requesting client in the response output stream, as shown in the following example:

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

The JSP engine treats comments as uninterpreted template text. If the generated comment has dynamic data, obtain it through expression syntax, as shown in the following example:

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

Directives

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

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

The valid directives are:

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

<%@ page%>

The page directive sets the JSP page level preferences.

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}"
%>
```

Attributes

Table 3-1 shows the valid attributes.

Table 3-1 JSP page Directives

Attribute	Valid Values	Description
`language`	`java`	Default: `java`. Scripting language for this JSP. Currently, iPlanet Application Server 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.

Table 3-1 JSP page Directives

Attribute	Valid Values	Description
`import`	comma-separated list of valid Java class names	Types and classes available to other methods in this JSP. This is identical to the `import` statement in a Java class.
`session`	`true` or `false`	Default: `true`. Indicates 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 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 the output buffer is full. If `true`, output is flushed to the output stream when the buffer is full. If `false`, an exception is raised when the buffer is full.
`isThreadSafe`	`true` or `false`	Default: `false`. Indicates the thread safety level in the page. The value determines the JSP engine behavior: if `true`, multiple requests are made to the JSP instance simultaneously, otherwise multiple requests are handled serially. For the most part, ensure 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 inside the translated page which is obtained from the page's `Servlet.getServletInfo()` method.
`errorPage`	valid URL for a JSP error page	Error page for this JSP; must be a JSP. Any `Throwable` object thrown but not caught by the original page is 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` and the initial `JspWriter` contents have been flushed to the `ServletResponse` output stream (for example, if part of the page has already been sent to the client), any subsequent attempt to invoke an error page may fail.

Table 3-1 JSP `page` Directives

Attribute	Valid Values	Description
`isErrorPage`	`true` or `false`	Default: `false`. Indicates whether the current JSP page is the possible target of another JSP page's `errorPage`. If `true`, 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%>

The include directive enables other JSP inclusions (or static pages) when the JSP is compiled into a servlet. The resource is treated as a part of the JSP.

Another way to include other resources is to use the `<jsp:include>` action, which includes resources at request time. For more information on file inclusion, see "Including Other Resources," on page 75.

Syntax

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

Attributes

Table 3-2 shows the valid attribute.

Table 3-2 JSP `include` Directive

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

The file attribute is either 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 slash ('/'). For absolute file attributes, the file name should begin with a slash ('/').

Example

If `who.jsp` is in the application `MyApp` (typically located in *install_dir*`/ias/APPS/MyApp`) and `who.jsp` contains the following tag:

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

then the system tries to include the file `baz.jsp` from *install_dir*`/ias/APPS/MyApp/add/baz.jsp`.

If `baz.jsp` contains the following tag:

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

then the system also includes the file *install_dir*`/ias/APPS/MyApps/add/who.jsp`.

<%@ taglib... %>

The tag library directive enables custom tag creation. For more information on creating custom tags, see "Value-added Features," on page 89.

Syntax

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

Attributes

Table 3-3 shows the valid attributes.

Table 3-3 JSP `<taglib>` Directive

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

Example

Consider the following JSP file, `who.jsp`, in the application `MyApp`, and a corresponding XML deployment descriptor file with a web application section as follows:

```
<taglib>
   <taglib-uri> http://www.mytaglib.com/spTags </taglib-uri>
   <taglib-location> /who/add/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 tag library location for `http://www.mytaglib.com/spTags`. The engine locates `/who/add/baz.tld`, and therefore looks for an XML file *install_dir*`/ias/APPS/MyApp/who/add/baz.tld`. This is the tag library descriptor file that describes the tags used in the file.

The URI or tag library location (if the URI is aliased) can also be relative. In this case, the `.tld` file is searched for relative to the current directory. For more details, see JSP v1.1 specification, section 5.2.

Scripting Elements

Scripting elements are made up of the following tags:

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

There are several implicit objects available to scripts, including the request and response objects. For more information about implicit objects, see “Implicit Objects,” on page 74.

Declarations <%! ... %>

The declarations element defines valid variables used throughout the JSP. 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.

Syntax

```
<%! declaration %>
```

Example

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

Expressions <%= ... %>

The expressions element evaluates variables. The expression value is substituted where the expression occurs. The result appears on the output stream.

Syntax

```
<%= expression %>
```

Example

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

Scriptlets <%...%>

The scriptlets element defines code blocks for execution and any legal code can appear here.

Syntax

```
<% script %>
```

Example

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

Actions

Actions perform activities, such as including other JSPs or specifying required plug-ins, creating or loading a Java bean, or setting or retrieving bean properties.

Some actions allow request time expressions as parameters, allowing you to set values for these attributes dynamically for the request. 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>

The `<jsp:useBean>` action tries to find a Java bean with the given name (`id`) and `scope`. If the bean exists, it is made available, otherwise this action creates it using the provided name, `scope`, and type and class information. A variable called *name*, specified with the attribute `id="`*name*`"`, is made available to the JSP so to 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, and so on are treated normally. Note that the `<jsp:useBean>` tag body 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.

Syntax

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

Attributes

Table 3-4 shows the valid attributes.

Table 3-4 `<jsp:useBean>` Attributes

Attribute	Description
`id`	Unique identifying object name.
`scope`	The object lifecycle is 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 (retrieved with `getAttribute(name)` where `name` is the object's `id`), and is available for the life of the request. • `session`: object is bound to the session object (retrieved with `getValue(name)` where `name` is the object's `id`) and is available wherever the session is available for the session life. A session must be active for this JSP in order to use this `scope`. • `application`: object is bound to the `ServletContext` (retrieved with `getAttribute(name)` where `name` is the object's `id`) and is available for the application existence, unless it is specifically destroyed.
`class`	Valid bean classname, used to instantiate the bean if it does not exist. If `type` is specified, `class` must be assignable to `type`. Both `beanName` and `class` cannot be specified for the same bean.
`beanName`	Valid bean name in the form of, `a.b.c` (classname) or `a/b/c` (resource name). Both `beanName` and `class` cannot be specified for the same bean. The `beanName` attribute can be an expression, evaluated at request time.
`type`	Defines the bean variable `type`. This attribute enables the variable `type` to be distinct from the implementation class specified. The type is required to be either the class itself, a class superclass, or an interface implemented by the class specified. If unspecified, the value is the same as the `class` attribute value.

Examples

This example shows a bean creation or a bean access 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 is present in the session. If so, it is given the local name `wombat` with `WombatType`. A `ClassCastException` is raised if the object is the wrong class and an `InstantiationException` is raised if the object is not defined.

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

For more information, see “Examples,” on page 69.

<jsp:setProperty>

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

Syntax

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

Attributes

Table 3-5 shows the valid attributes.

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

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

This example sets the property `name` value to the 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>

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

Syntax

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

Attributes

Table 3-6 shows the valid attributes.

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

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

Examples

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

<jsp:include>

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

An additional method for including other resources is the `<%@ include%>` directive, which includes the resource at compile time. For more information on file inclusion, see "Including Other Resources," on page 75.

Syntax

```
<jsp:include page="URI" flush="true|false"/>
```

Attributes

Table 3-7 shows the valid attributes.

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

Attribute	Description
`page`	Includes either an absolute or relative page reference. For absolute references, this field begins with a slash ("`/`"), 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 to be evaluated at request time.
`flush`	Determines whether to flush the included page to the output stream.

Examples

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

<jsp:forward>

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

Syntax

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

Attributes

Table 3-8 shows the valid attributes.

Table 3-8 <jsp:forward> Attributes

Attribute	Description
page	Valid URL pointing to the page to include. This attribute may contain, at request time, an expression to evaluate. The evaluation must be a valid URL.

NOTE If the page output is unbuffered (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="/who/handleAlternativeInput.jsp" />
```

The following element shows how to forward a static page based on a dynamic condition.

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

<jsp:plugin>

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

The <jsp:plugin> tag is replaced by either the appropriate <object> or <embed> tag for the requesting user agent and is sent to the response output stream.

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

- <jsp:params> sends a parameter block to the Applet or JavaBean 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 browser content if the plugin cannot be started (either because `object` or `embed` is not supported, or due to some problem). The tag body is presented to the browser when a failure of the surrounding `<jsp:plugin>` occurs. For example:

```
<jsp:plugin ...>
   <jsp:fallback><b>Plugin could not be
   started!</b></jsp:fallback>
</jsp:plugin>
```

If the plug-in starts, but the Applet or JavaBean cannot be found or started, a plug-in specific message is sent to the user, often as a popup window reporting a `ClassNotFoundException`.

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>
```

Attributes

The `<jsp:plugin>` tag takes most of its attributes from the HTML `<applet>` and `<object>` tags (`<applet>` is defined in HTML 3.2 and is deprecated, `<object>` is defined in HTML 4.01). Refer to the official HTML 4.01 specification where these tags are described:

`http://www.w3.org/TR/REC-html40/`

Table 3-9 shows the valid attributes.

Table 3-9 `<jsp:plugin>` Attributes

Attribute	Description
`type`	Identifies the component type, `bean` or `applet`.
`code`	As defined by the HTML specification.
`codebase`	As defined by the HTML specification.
`align`	As defined by the HTML specification.
`archive`	As defined by the HTML specification.
`height`	As defined by the HTML specification.
`hspace`	As defined by the HTML specification.
`jreversion`	Identifies the JRE specification version number the component requires to operate. Default: `1.1`
`name`	As defined by the HTML specification.
`vspace`	As defined by the HTML specification.
`title`	As defined by the HTML specification.
`width`	As defined by the HTML specification.
`nspluginurl`	URL where the JRE plug-in can be downloaded for Netscape Navigator, default is implementation-defined.
`iepluginurl`	URL where the JRE plug-in 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. You can refer to them from anywhere in a JSP without previously defining them (for example, with `<jsp:useBean>`).

Table 3-10 shows the objects 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`, for example, `javax.servlet.HttpServletRequest`
`response`	The request response (for example, the page and its path returned to the caller).	page	protocol dependent subtype of `javax.servlet.ServletResponse`, for example, `javax.servlet.HttpServletResponse`
`pageContext`	The JSP page context.	page	`javax.servlet.jsp.PageContext`
`session`	The session object (if any) created for or associated with the caller.	session	`javax.servlet.http.HttpSession`
`application`	This JSP's servlet context, from the servlet's configuration object through `getServletConfig()`, `getContext()`.	application	`javax.servlet.ServletContext`
`out`	An object that writes to the output stream.	page	`javax.servlet.jsp.JspWriter`
`config`	This JSP's servlet configuration object (`ServletConfig`).	page	`javax.servlet.ServletConfig`
`page`	This page's class instance that is processing the current request.	page	`java.lang.Object`
`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 with one of the request parameters as `<%= request.getParameter("param"); %>`.

Programming Advanced JSPs

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

- Including Other Resources
- Using JavaBeans
- Accessing Business Objects

Including Other Resources

An important JSP feature is the ability to dynamically include other page generating resources or their results at runtime. You can include static HTML page content or process a separate JSP and include its results in the output page.

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

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

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

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

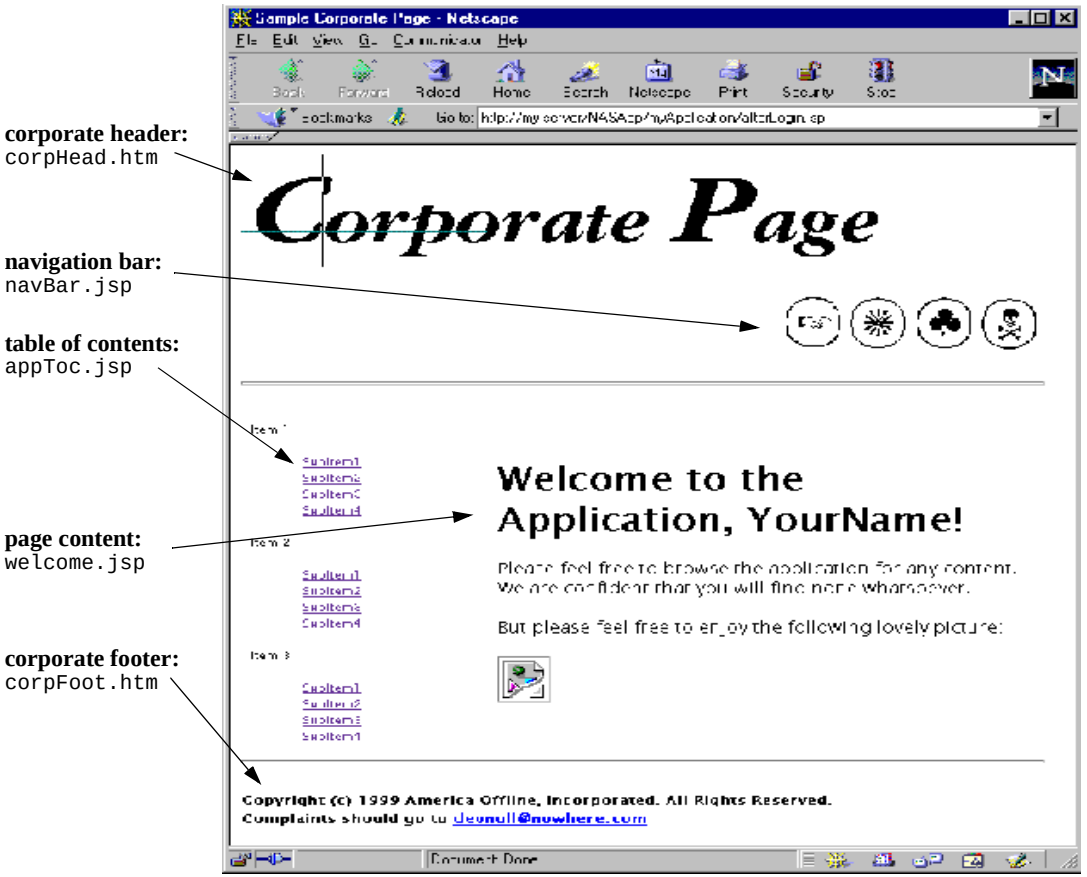
- the `<jsp:include>` action:

  `<jsp:include page="URI" flush="true|false" />`

If you include a resource with the `<%@ include%>` directive, the resource is included when the JSP is compiled into a servlet, and is treated as 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 “Directives,” on page 60.

If you include a resource with the `<jsp:include>` action, the resource is included when the JSP is called. For more information, see “Actions,” on page 65.

The following example shows how each page portion is a separate resource, while access is from a single JSP. The source code for this example page shows both methods for including resources: static resources are included with the `<jsp:include>` action, and dynamic resources are included with the `<%@ include%>` directive.



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 JavaBeans

JSPs support several tags to instantiate and access JavaBeans. Beans perform computations to obtain a result set and 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 JavaBean specification. For information about JavaBeans, see `http://java.sun.com/beans`.

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

This supports standard JavaBeans, not EJBs. To access EJBs from a JSP, see “Accessing Business Objects,” on page 77. In the JSP 0.92 specification, the request and response objects were accessed through “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. For more information, see “Implicit Objects,” on page 74.

Accessing Business Objects

Because JSPs are compiled into servlets at runtime, they have access to all server processes, including EJBs. You access beans or servlets in the same way you would access them from a servlet, as long as the Java code is embedded inside an escape tag.

The method described here for contacting EJBs is identical to the method used from servlets. For more information about contacting EJBs, see “Accessing Business Logic Components,” on page 41.

This example shows a JSP accessing an EJB called `ShoppingCart` by importing the cart’s remote interface and creating a cart handle with the user’s session ID:

```
<%@ 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 JNDI looking 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 provide an EJB method to convert raw data to a format acceptable to the page, such as `getDataAsHTML()`, as shown above.

Deploying JSPs

There are two ways the iPlanet Application Server deploys JSPs, as either unregistered or registered JSPs.

Unregistered JSPs

Unregistered JSPs are deployed by copying them to the corresponding directory structure (`applicationName/moduleName`), in the `AppPath`. These JSPs are invoked using the URL access as follows:

`http://`*server*`:`*port*`/`*AppPrefix*`/`*ModuleName*`/`*JSPFileName*

For more information, see “Invoking JSPs,” on page 80.

Registered JSPs

The iPlanet Application Server allows JSPs to be registered with `GUID`s, using XML. This allows JSPs to use iPlanet Application Server value-added features such as load balancing. This is done using XML files with the `<jsp-file>` entry as detailed in the Servlet 2.2 specification.

The following XML files are a deployment descriptor example for a registered JSP. This is the `web.xml` file:

```
<?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>
```

This is the `ias-web.xml` file:

```
<?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 this 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 the `.jsp` extension is required. It is possible for the servlet name to be `JSPExample.jsp` instead.

This JSP is accessed from a URL by using one of the following examples:

- `http://`*server*`:`*port*`/`*AppPrefix*`/`*ModuleName*`/JSPExample`
- `http://`*server*`:`*port*`/`*AppPrefix*`/`*ModuleName*`/JSPExample.jsp` (use if the `servlet-name` entry in the XML file is `JSPExample.jsp`)

Invoking JSPs

A JSP is invoked programmatically from a servlet or by addressing it directly from a client using a URL. You can also include JSPs. For more information, see “Including Other Resources,” on page 75.

Calling a JSP With a URL

JSPs can be called using URLs embedded as links in the application pages. This section describes how to invoke JSPs 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*`/`*ModuleName*`/`*jspName*`?`*name*`=`*value*

Table 3-11 shows each URL section.

Table 3-11 URL Sections

URL element	Description
server:*port*	Address and optional web server port number handling the request.
AppPrefix	Indicates to the web server that the URL is for an iPlanet Application Server application. The request is routed to the iPlanet Application Server executive server. Configure this using the registry entry `SSPL_APP_PREFIX`.
moduleName	The name of the web module (these names are unique across the server).
jspName	The JSP's file 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.mycompany.com/BookApp/OnlineBookings/directedLogin.jsp
```

Using a generic application for a JSP has the same requirements and restrictions as using a generic application for a servlet. There must be an application called `Default` with a registered XML file. Any URL request to access a servlet or JSP with the `/servlet/` entry is sent to the generic application `Default`. For more information about this requirement, see "Invoking Generic Application Servlets," on page 49.

Invoking JSPs in a Generic Application

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

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

Table 3-12 shows each URL section.

Table 3-12 URL Sections

URL element	Description
server:*port*	Address and optional web server port number handling the request.
`servlet`	Indicates to the web server that the URL is for a generic servlet object.
jspName	The JSP's name, including the `.jsp` extension.

Table 3-12 URL Sections

URL element	Description
?*name*=*value*...	Optional *name*=*value* parameters to the JSP. These are accessible from the `request` object.

For example:

```
http://www.Who.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.
- The `forward()` method in the `RequestDispatcher` interface hands JSP interaction control.

For more information about these methods, see “Delivering Client Results,” on page 44.

For example:

```
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

The following sections summarize the JSP 1.1 tags.

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"
```

For more information, see “Directives,” on page 60.

Expressions

```
<%= expression %>
```

For more information, see “Scripting Elements,” on page 64.

Scriptlets

```
<% scriptlet %>
```

For more information, see “Scripting Elements,” on page 64.

Comments

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

For more information, see “Comments,” on page 59.

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">
```

For more information, see “Actions,” on page 65.

Other Actions

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

```
<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>
```

For more information, see “Actions,” on page 65.

Modifying Custom Tags for JSP 1.1

iPlanet Application Server custom tags may need to be modified for JSP 1.1 for the following reasons:

- The `.tld` files do not conform to the DTD at `http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd`.

  For example, every reference to the `prefix` attribute must be changed to `shortname`.

- The following `DOCTYPE` element is missing:

  ```
  <!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
  ```

These modifications are necessary if you want to use the JSP command-line compiler. For more information about this compiler, see “Compiling JSPs: The Command-Line Compiler,” on page 85.

Compiling JSPs: The Command-Line Compiler

As of Service Pack 3, iPlanet Application Server uses the Jasper JSP compiler from Apache Tomcat 3.2 to compile JSP 1.1 compliant source files into servlets. All of the features available in this version of Jasper are available in the iPlanet Application Server environment.

NOTE Jasper has been modified to meet the requirements of iPlanet Application Server, so you should use only the Jasper version provided with iPlanet Application Server. Other versions may not work with iPlanet Application Server.

Developers can use the JSP compiler to perform syntax checks of JSP files prior to deployment. Deployers can also benefit from the JSP compiler by precompiling JSP files into WAR files before the WAR files are deployed to the application server.

The `jspc` command line tool is located under *install_dir*`/ias/bin` (make sure this directory is in your path). The format of the `jspc` command is as follows:

`jspc [`*options*`]` *jsp_files*

The *jsp_files* can be one of the following:

files	One or more JSP files to be compiled.
`-webapp` *dir*	A directory containing a web application. All JSPs in the directory and its subdirectories are compiled. You cannot specify a WAR, JAR, or ZIP file; you must first deploy it to an open directory structure using `iasdeploy`.

The basic *options* for the `jspc` command are:

`-q`	Enables quiet mode (same as `-v0`). Only fatal error messages are displayed.
`-d` *dir*	Specifies the output directory for the compiled JSPs. Package directories are automatically generated based on the directories containing the uncompiled JSPs. The default top-level directory is the directory from which `jspc` is invoked.
`-p` *name*	Specifies the name of the target package for all specified JSPs, overriding the default package generation performed by the `-d` option.
`-c` *name*	Specifies the target class name of the first JSP compiled. Subsequent JSPs are unaffected.
`-uribase` *dir*	Specifies the URI directory to which compilations are relative. Applies only to JSP files listed in the command, and not to JSP files specified with `-webapp`. This is the location of each JSP file relative to the `uriroot`. If this cannot be determined, the default is `/`.

`-uriroot` *dir*	Specifies the root directory against which URI files are resolved. Applies only to JSP files listed in the command, and not to JSP files specified with `-webapp`. If this option is not specified, all parent directories of the first JSP page are searched for a `WEB-INF` subdirectory. The closest directory to the JSP page that has one is used. If none of the JSP's parent directories have a `WEB-INF` subdirectory, the directory from which `jspc` is invoked is used.
`-webinc` *file*	Creates partial servlet mappings for the `-webapp` option, which can be pasted into a `web.xml` file.
`-webxml` *file*	Creates an entire `web.xml` file for the `-webapp` option.
`-ieplugin` *class_id*	Specifies the Java plugin COM class ID for Internet Explorer. Used by the `<jsp:plugin>` tags.

The advanced *options* for the `jspc` command are:

`-v[`*level*`]`	Enables verbose mode. The *level* is optional; the default is 2. Possible *level* values are: • `0` - fatal error messages only • `1` - error messages only • `2` - error and warning messages only • `3` - error, warning, and informational messages • `4` - error, warning, informational, and debugging messages
`-dd` *dir*	Specifies the literal output directory for the compiled JSPs. Package directories are not made. The default is the directory from which `jspc` is invoked.
`-mapped`	Generates separate `write` calls for each HTML line and comments that describe the location of each line in the JSP file. By default, all adjacent `write` calls are combined and no location comments are generated.
`-die[`*code*`]`	Causes the JVM to exit and generates an error return *code* if a fatal error occurs. If the *code* is absent or unparsable it defaults to `1`.

`-webinc` *file*	Creates partial servlet mappings for the `-webapp` option, which can be pasted into a `web.xml` file.
`-webxml` *file*	Creates an entire `web.xml` file for the `-webapp` option.
`-ieplugin` *class_id*	Specifies the Java plugin COM class ID for Internet Explorer. Used by the `<jsp:plugin>` tags.

When a JSP is compiled, a package is created for it. The package is located in *install_dir*`/ias/APPS/`*appName*`/`*moduleName*`/WEB-INF/compiled_jsp/`. (If the code is deployed as an individual module, the *moduleName* is `modules`.) The package name should start with `jsp.APPS`, which is the default package prefix name in iPlanet Application Server.

Use the basic options of `jspc` when compiling the JSP for iPlanet Application Server. iPlanet Application Server does not use the standard Jasper naming conventions, so you must specify the generated file name, class name (`-c`), package (`-p`) and directory (`-d`).

For example, to precompile `fortune.jsp` to `fotune.java`, use these commands:

```
cd install_dir/ias/APPS/fortune/fortune
jspc -d WEB-INF/compiled_jsp -p jsp.APPS.fortune -c fortune fortune.jsp
```

The `fortune.java` file and its respective class are generated in the following directory:

install_dir`/ias/APPS/fortune/fortune/WEB-INF/compiled_jsp/jsp/APPS/fortune`

The package name of the `fortune.class` file is `jsp.APPS.fortune`, because `fortune` is the package name of the JSP and iPlanet Application Server uses `jsp.APPS` as a prefix.

NOTE iPlanet Application Server 6.0 SP3 supports debugging of the compiled servlet code using Forte For Java Internet Edition 2.0. However, you cannot debug the uncompiled JSP pages.

Additional documentation for the JSP compiler is on the Jakarta site:

`http://jakarta.apache.org/tomcat-4.0/jakarta-tomcat-4.0/jasper/doc/jspc.html`

Value-added Features

The following sections summarize the iPlanet value-added features:

- Custom Tag Extensions
- JSP Load Balancing
- JSP Page Caching

Custom Tag Extensions

The JSP 1.1 specification has support for a user defined custom tags protocol. Although the specification does not mandate any tags, as a value-added feature the iPlanet Application Server contains custom tags that follow the JSP 1.1 defined tag extension protocol. For more information, see the JSP 1.1 specification, Chapter 5.

Some tags provide LDAP and database query support, while others provide support for conditionals inside JSPs because the specification provides no primitive support.

To support JSP page caching, the iPlanet Application Server ships with a `Cache` tag library. For details, see “JSP Page Caching,” on page 106.

The other tags included with the iPlanet Application Server are *for internal use only*, to support converting `GX` tags. These tags are used in generating JSP 1.1 pages from JSP 0.92 pages (which supported `GX` tags), and should not be used externally.

The following tag libraries are introduced by the iPlanet Application Server:

- `Query`
- `LDAP`
- `Conditional`
- `Attribute`

For examples of the custom tag extensions, see the samples in the *install_dir*`/ias/ias-samples/iastags/` directory.

Database Query Tag Library

The query tag library supports row set declarative declarations in JSP pages, along with loops to loop through a result set and a display tag for displaying column values. The following sections describe the query tag library.

useQuery Tag

The useQuery tag declares a result set to use. The useQuery tag defines what query is being made and what available fields are for use. If the result set the useQuery wants to save to is already in the scope, the tag body is skipped, and although the row set is created it is ignored.

If the result set does not exist, the created row set is exported using the useQuery tag's id attribute of the specified scope, which defaults to request. Either the specified command is used, or a query located in the queryFile is loaded. The loaded query file name is either the name located in the queryName attribute, or, if none is specified, the tag's id attribute value is used. Once the row set is initialized, it may be executed if the execute tag is specified. Note that you must execute a query 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 attribute value is a relative path, it is looked up in the RDBMS.path.query path. If this variable is not set, the iPlanet Application Server specific GX.path.query property is used. If the file is *not* located relative to the JSP, the query file should look something like the following examples:

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

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

A blank line (without spaces, tabs, and so on) separates queries which are named using the query ... is construct.

Syntax

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

param Tag

The param tag sets a parameter on a row set. The parameter name can be either an index or the actual parameter name as saved in the dictionary. Note that bindOnLoad parameters must exist in the useQuery tag body before any non-bindOnLoad parameters. The parameter value is either the value stored in the value attribute or the param tag body contents. Because JSP 1.1 tags don't generally

nest (<%= . . . %> being the notable exception), the only way to bind a parameter to a value from another query is to place a `field` tag within the `param` tag body and have the `param` tag use its body as the value, which is why you can place the value in the tag's body.

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

Syntax

```
<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|Timestamp"
format="java-format-string for dates">value</rdbm:param>
```

loop Tag

The `loop` tag loops through the result set contents. The `query` attribute is used to locate the result set or an enclosing `useQuery` tag. The `start` attribute is used to indicate the loops's starting position. `start` may either refer to a parameter or to an attribute, which is looked up using `PageContext.findAttribute()`, or to a constant integer value. The value indicates which record number to start on, or `last`, which causes the row set 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 row set is executed before looping begins.

Syntax

```
<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>
```

field Tag

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

Syntax

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

close Tag

The `close` tag releases system resources. The `resource` attribute locates the exported query resource (result set) and calls `close()`.

Syntax

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

execute Tag

The `execute` tag executes the identified query.

Syntax

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

goRecord Tag

The `goRecord` tag executes the specified query and moves the result set to the record indicated by the `start` attribute. `start` may either refer to a parameter, or to an attribute, or to a constant. If the `start` attribute is `last`, the result set is moved to the last record.

Syntax

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

Example

The following tags produce the output display at the end of the example.

```
<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>
```

Here are the results:

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 of LDAP connections is that they are likely to be request-specific, that is, the current user may be the only user authenticated to read the LDAP attributes of that user's data. Because of this, an additional `LDAPAuthenticate/Authorize` tag is required so mappings between *current user* and *connection to perform LDAP searches* are programmable. When the LDAP server is remote and a general authorization-capable login is not available, the `LDAPAuthenticate` tag is used. The following sections describe the LDAP tag library.

authenticate Tag (also called connection)

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

Syntax

```
<ldap:[authenticate|connection] query="name of ldap exported query"
url="ldap://..." password="..."> </ldap:[authenticate|connection]>
```

authorize Tag

The `authorize` tag works in the `LDAPTagSearch` context. The `LDAPTagSearch` is either retrieved from the `PageContext` using `findAttribute` and the `query` attribute name, or by finding a parent `useQuery` tag and getting 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 `:Who` values in it), then the `authorize` tag body needs to contain `param` tags for each parameter. At the end of the tag, the tag attempts to authorize the `LDAPTagSearch`.

Syntax

```
<ldap:authorize query="name of ldap exported query"
dn="distinguished name for the user to authorize against">
</ldap:authorize>
```

param Tag

The `param` tag sets the LDAP URL parameters. LDAP URLs are specified in the `url` and `dn` attributes of the `authorize` tag and in the `url` attribute of the `field` and `useQuery` tags.

A URL `param` is any Java level identifier with a prepended ":", similar to the query parameters in a `.gxq` file. For example:

```
ldap://nsdir.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 that `389` is not a tag because it's before the LDAP URL DN section and isn't a Java level identifier.

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

Syntax

```
<ldap:param name="parameter name in authenticate userDN or query
url" query="name of ldap exported query" value="...">default
value</ldap:param>
```

password Tag

The `password` tag sets the `authenticate` tag password. Like the `param` tag, the `password` tag body becomes the password value, assuming that no value is specified as a `password` tag attribute. The `password` tag is legal only inside the `authenticate` tag.

Syntax

```
<ldap:password query="name of ldap exported query"
value="...">default value</ldap:password>
```

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 a `loop` tag loops through or that a `field` tag displays. This is because the `loop` tag cannot specify parameter mappings except in the body – which is too late for the loop to determine if there are any results. The `field` tag can already specify a URL and 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 attribute value is a relative file specification, the file is searched for in the `LDAP.path.query` path. If this variable is not set, then the iPlanet Application Server specific `GX.path.query` property is used instead. The file is *not* located relative to the JSP. The query file should look something like the following example:

```
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 the queries, which are named using the `query ... is` construct.

Syntax

```
<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>
```

loopEntry 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` (for more information, see “useQuery Tag,” on page 95). The `start` and `end` tags work as specified in the query’s loop tag. If the `useVL` attribute is true, then an `{id}_contentCount` value is exported, which corresponds to the `VirtualListResponse contentCount`. On each pass through the loop, the current `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` and if the `useVL` attribute is set, then a `VirtualListControl` is used to position the returned entries window. The actual public draft URL for `VirtualList` is located here.

Syntax

```
<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]>
```

loopValue Tag

The `loopValue` tag loops through a multi-value `LDAPEntry` attribute or the first `LDAPEntry` in an `LDAPSearchResults`. The `query` attribute points to an exported `LDAPTagSearch` (for more information, see "useQuery Tag," on page 95). 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 specified in the query's loop tag. On each pass through the loop, the current `LDAPAttribute` value is exported at the specified `scope` using the specified `id`.

Syntax

```
<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>
```

field Tag

The `field` tag prints out the value of a single value attribute as specified in the `query`, `url`, or `entry` attributes, and the `attribute` attribute. If no value exists, the `field` tag body is passed. The `field` tag body is only evaluated if the `url` has parameters (and hence, there are 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 `entry` name is returned as the field value.

Syntax

```
<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>
```

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 `useQuery` 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 `LDAPSortKey` constructor's `keyDescription` parameter. The `useQuery` tag supports multiple sorts. Sorts are prioritized in the order specified.

Syntax

```
<ldap:sort query="name of ldap exported query" order="..."/>
```

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 call abandons any executing `SearchResults` and releases the connection to the connection pool (or calls `disconnect()` on the connection, if the connection doesn't come from a pool; the connection can come from the `authenticate` tag).

Syntax

```
<ldap:close resource="name of ldap exported query"/>
```

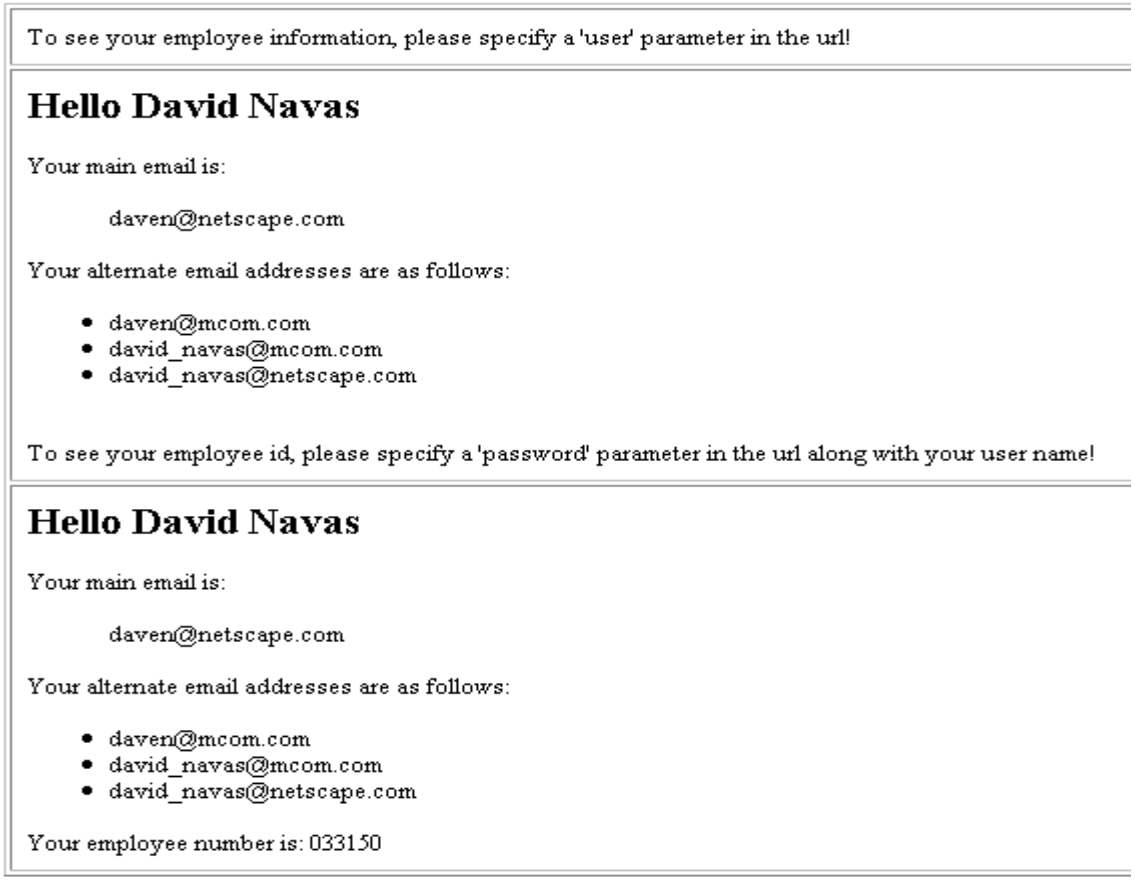
Example

The following example uses both `LDAP` and `switch` tags. 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="Who" 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 -- not good!

</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!):
```



Conditional Tag Library

The cond tag family supports `switch` and `case` tags, allowing a special case when a row set is at the end, when a user is given management-only information types, when a user has requested high bandwidth content, and so on.

However, for ease of use and better readability, the following equivalents can be used:

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 ways may be more expressive than you really require. For example:

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

is the same as:

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

Additionally, one might assume that:

```
<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>
```

Always consider, if the increased expressiveness is worth the trade-off in possible user confusion.

The root tags are described next.

switch Tag

The `switch` tag defaults to a straight value comparison. However, it is more likely to be used as a `RowSet` type switch to replace some callbacks that `DBRowSet` contains. The `switch` tag keeps track of whether a particular case statement has fulfilled the switch statement and only exports its body to the content page.

Syntax

```
<cond:switch type="[value|role|rowset|ldap|attribute|parameter]"
value="constant value, role name, rowset name, etc."> ...
</cond:switch>
```

case Tag

The `case` tag contains an operation and (possibly) a second operand, and is used to determine if the case statement fulfills the `switch` tag. Note that if a case and switch combination are used where 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, which allows more efficient JSP building. The `case` tag body is not 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, to fulfill the switch regardless. If no operation is specified and the switch type is `role`, the operation assumes it is `equals`.

Note that certain case operations make sense only in certain switch types. For example:

- The `isLast` and `notEmpty` tags are useful for both `ldap` (query or entry) and `RowSet` switch types.
- The `executeNotEmpty` operation makes sense only for `RowSet` switch types.
- The `connected` and `authenticated` operations make sense only for `ldap` switch types.
- The “=, <, >” etc. operations make sense only when comparing numerical values. The switch and case values are converted to doubles (if necessary), and their values are compared.
- The `equals` and `equalsIgnoreCase` operations make sense only when comparing strings, although `equals` is called against the switch value `equals` method – which might be implemented by an object to compare itself to a string (the case value, always). `notEmpty` also makes sense for strings as a check when a parameter has specified a non-zero length setting.

Syntax

```
<cond:case
operation="[=|<|>|<=|>=|!=|<>|><|=>|=<|~=|equals|equalsIgnoreCase|e
lse|exists|notEmpty|executeNotEmpty|isLast|connected|authenticated|
{method-name}]"
value="..."></cond:case>
```

value Tag

The value tag body is evaluated and passed to the value tag's parent. The parent implements the IValueContainingTag, which both switch and dynamicValue do. You can also specify the value in a value tag attribute. But then, if you do that, it is better to put the value in the switch or case directly.

Syntax

```
<cond:value value="blah">default value</cond:value>
```

Dynamic Value Tag

The dynamicValue tag body should have at least two elements. One is a value tag, which builds the dynamic interest value. The second is a case tag, whose value attribute is extracted from the enclosing dynamicValue instance. The dynamicValue tag does have a value tag, as follows:

```
<cond:attribute name="Who">
       <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>
```

There are no machine equivalents to comparison bits in the status register, therefore the operation performs three (3) times.

Syntax

```
<cond:dynamicValue value="blah"> ... <cond:value/> ...
<cond:*case*/> ... </cond:dynamicValue>
```

Example

The following example shows how a switch might be used. The three links at the end produce the three different output types:

```
<%@ 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 are as follows:

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

The following sections provide information on the attribute tag library.

getAttribute Tag

The `getAttribute` tag prints the named attribute’s value, which is retrieved from the specified `scope`. If no scope is specified, `findAttribute()` is used to find the attribute. If no value is found, the tag body is printed instead. The `format` is used as the `query:field` tag.

Syntax

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

setAttribute Tag

The `setAttribute` tag sets the named attribute’s value in 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 specified, the tag body is used.

Syntax

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

getParameter Tag

The `getParameter` tag prints the named parameter's value. If no parameter value exists, the tag body is printed instead. The format attribute is used as the `query:field` tag.

Syntax

```
<attr:getParameter name="parameterName" format="urlEncode">default
value</attr:getParameter>
```

Get Remote User Tag

The `getRemoteUser` tag prints the servlet's remote user name.

Syntax

```
<attr:getRemoteUser>default value</attr:getRemoteUser>
```

Example

For more information, see the examples in "LDAP Tag Library," on page 94 and "Conditional Tag Library," on page 100.

JSP Load Balancing

Servlets can be load balanced because each servlet has a GUID associated with it. You simply distribute the servlet across all the iPlanet Application Server instances. However, JSPs are converted by iPlanet Application Server into servlets at runtime and initially have no individual GUIDs associated with them. This makes load balancing and failover of JSPs impossible when they are called directly from the browser (as opposed to being called through a servlet).

iPlanet Application Server 6.0 supports load balancing of JSPs individually. To obtain load balancing and failover capabilities for JSPs called directly from the browser, follow these steps:

1. In the XML descriptor, assign a GUID to each JSP you want to load balance. For details about assigning GUIDs to JSPs, see "Registered JSPs," on page 79.

2. In iPlanet Application Server, JSPs are run using the system servlets `JSPRunner` and `JSPRunnerSticky`. These servlets are registered at the time of installation. Use the Administration Tool to make these system servlets distributed across the servers you want to include in the load balancing.
3. Check the servlet component properties of `System_JSPRunner` and `System_JSPRunnerSticky`. Make sure that all the servers across which JSPs are to be load balanced are listed correctly.
4. Load balance the JSP just you would a servlet, through the administrative tool. Distribute the JSPs across the servers you want to include in the load balancing.
5. Restart the web server.

For details about distributing application components and changing component properties, see the *Administrator's Guide*.

JSP Page Caching

A new feature called JSP Caching, aids in compositional JSP development. This provides functionality to cache JSPs within the Java engine, thereby allowing a master JSP to include multiple JSPs (for example, a portal page). Each can be cached using different cache criteria. Think of a portal page containing a window to view stock quotes, another to view weather information, and so on. The stock quote window can be cached for 10 minutes, the weather report window for 30 minutes, and so on.

Note that JSP caching is in addition to results caching. A JSP can be composed of several JSPs, each having separate cache criteria. The composed JSP can be cached in the KXS using the results caching with a `GUID`. For more information, see "Registered JSPs," on page 79.

JSP caching uses the custom tag library support provided by JSP 1.1. A typical cacheable JSP page looks like this:

```
<%@ 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> tags delimit the cache constraints. The <ias:criteria> tag specifies the timeout value and encloses different cache criteria. Cache criteria are expressed using any or both tags, <ias:check> and <ias:param>. The tag syntax is as follows:

- <ias:criteria timeout="val" > – specifies the cached element timeout, in seconds. The cache criteria are specified here before the closing </ias:criteria>.
- <ias:check class="classname" /> – is one mechanism of specifying cache criteria. The classname refers to a class that has a method called check, which has the following signature:

  ```
  public Boolean check(ServletRequest, Servlet)
  ```

  This returns a boolean value indicating if the element is to be cached or not.
- <ias:param name="paramName" value="paramValue" scope="request" /> – is the other mechanism to specify cache criteria.

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

Table 3-13 shows the paramValue parameter values, which determine if caching is performed or not.

Table 3-13 paramValue Parameter Values

Constraint	Description
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 attribute sources to be checked and can be page, request (default), session, or application.

Example

The following example represents 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, Servlet _serv)
   {

      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 it is possible to have multiple sets of `<ias:param>` and `<ias:check>` inside an `<ias:criteria>` block. Also, it is possible to have multiple `<ias:criteria>` blocks inside a JSP.

Chapter 4

Introducing Enterprise JavaBeans

This chapter describes how Enterprise JavaBeans (EJBs) work in the iPlanet Application Server application programming model. This chapter begins by defining an EJB's role and delivery mechanisms. Next it describes the two EJB types—entity and session beans—and gives details on when to use them. Finally, the chapter provides a design overview of an object-oriented iPlanet Application Server application using EJBs to encapsulate business logic.

This chapter contains the following sections:

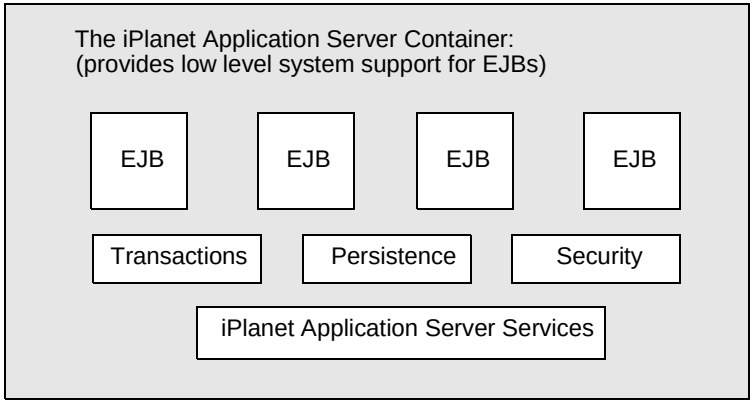
- What Enterprise JavaBeans Do
- What is an Enterprise JavaBean?
- Session Beans and Entity Beans
- EJB Role in an iPlanet Application Server Application
- Designing an Object-Oriented Application

NOTE If you know about EJBs and how they are used in an iPlanet Application Server, jump ahead for specific instructions and guidelines for developing EJBs for use with an iPlanet Application Server. See Chapter 6, "Building Entity EJBs," and Chapter 11, "Creating and Managing User Sessions."

What Enterprise JavaBeans Do

In an iPlanet Application Server, EJBs are the application workhorses. Servlets act as the application's central dispatchers and handle the presentation logic. EJBs do the bulk of the application's actual data and rules processing, but provide no presentation or visible user interface services. EJBs enable partitioning of business logic, rules, and objects into discrete, modular, and scalable units. Each EJB encapsulates one or more application tasks or objects, including data structures and operation methods. EJBs 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. The iPlanet Application Server software environment provides the EJB container. This container provides all standard container services denoted by the Sun EJB specification and also provides additional services specific to an iPlanet Application Server.



The container handles remote access, security, concurrency, transaction control, and database accesses. 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 bean developer can create generic, task focused EJBs to be used with any vendor's products that support the EJB standard.

What is an Enterprise JavaBean?

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

The fundamental EJB characteristics are as follows:

- Bean creation and management is handled at runtime by the iPlanet Application Server provided container.
- Client access mediation is handled by the container and the server where the bean is deployed, freeing the bean developer 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 deployable 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 bean recompiling.
- A client's bean definition view 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.
- EJBs can be dynamically reloaded while the iPlanet Application Server is running.

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 application development model that uses EJBs, and guarantees greater bean reuse. The client contract stipulates how an EJB object is identified, how its methods are invoked, and how it is created and destroyed.

The EJB container enables distributed application building using your own components and components from other suppliers. The iPlanet Application Server provides high level transaction, state management, multithreading, and resource pooling wrappers, thereby shielding you from having to know the low-level API details.

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 tools at design and deployment. At runtime, a client's bean access is container-controlled by the server where the EJB is deployed.

The EJB container is also responsible for ensuring that a client can invoke the specialized business methods the EJB defines. While a bean developer implements methods inside the bean, the developer must 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* for 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 to create a bean. In addition, some EJB types, known as *entity beans*, must also define finder methods for each way used 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, and is completely transparent to a client. There are several parts to the component contract for any given bean, as follows:

- **Life cycle:** For EJB session beans, this includes the `javax.ejb.SessionBean` and `javax.ejb.SessionSynchronization` interface implementations. For EJB entity beans, this includes the `javax.ejb.EntityBean` interface implementation.
- **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 of its EJBs.

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

Understanding JAR File Contracts

The standard format used to package an enterprise bean 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 the iPlanet Application Server you can create a `.jar` file containing EJBs using the iPlanet Application Server Deployment Tool. For more information, see the *Deployment Tool Online Help*.

The EJB-JAR file must contain the Deployment Descriptor (DD) as well as all class files for the following:

- The enterprise bean class.
- The enterprise bean home and remote interface.
- The primary key class for an entity bean.

In addition, the EJB-JAR file must contain the class files for all classes and interfaces for the enterprise bean class, and the remote and home interfaces to use. For more information on the EJB-JAR file contents, see Chapter 10, “Deployment Packaging.”

Session Beans and Entity Beans

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

The following sections describe the two bean types.

Understanding Session Beans

Session EJBs 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 the iPlanet Application Server failover support for stateful session beans.

A session bean implements business rules or logic. All functionality for remote access, security, concurrency, and transactions are provided by the EJB container. A session EJB is a private resource used only by the client that creates it. For example, you might create an EJB to simulate an electronic shopping cart. Each time a user logs in to an 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:

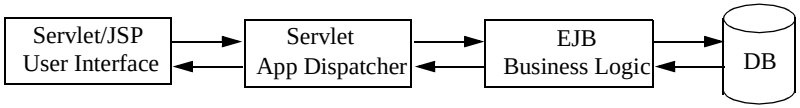
- Data representation in the Enterprise Information System (EIS) resource, usually a database.
- Bean managed transaction demarcation.
- Container managed transaction demarcation.
- Shared access for all users.
- Exists as long as its data is in a database.
- Transparently survives EJB server crashes.

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

EJB Role in an iPlanet Application Server Application

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

When a user invokes an iPlanet Application Server application servlet from a browser, the servlet invokes one or more EJBs to do the bulk of the application's business logic and data processing. For example, the servlet may load a JavaServer Page (JSP) to the user's browser to request a user name and password, then pass the user 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 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, 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 shopping cart data 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 the 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 challenge when creating an application that uses EJBs is determining how to break up an application into servlets, JSPs, session beans, and/or entity beans.

Designing an Object-Oriented Application

Partitioning an iPlanet Application Server application's business logic and data processing into the most effective set of EJBs is the bulk of your job as a developer. There are no hard and fast rules for object-oriented design with EJBs, other than that entity bean instances tend to be long lived, persistent, and shared among clients, while session bean instances tend to be short lived and used only by a single client. Therefore, the following sections are mostly high level iPlanet Application Server specific information to improve application speed, making EJBs modular, shareable, and maintainable.

With all object-oriented development, you must determine what granularity level you need for your business logic and data processing. Granularity level refers to how many pieces you break an application into. A high level of granularity—where you divide an application into many, smaller, more narrowly defined EJBs—creates an application that may promote greater EJB sharing and reuse 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 application performance degradation and more overhead. EJBs, like JavaBeans, are not simply Java objects. EJBs are higher level entities than Java objects. They are components with remote call interface semantics, security semantics, transaction semantics, and properties.

Planning Guidelines

In general, create an iPlanet Application Server application to balance the need for execution speed with the need for sharing EJBs among applications and clients, and deploying applications across servers:

- Ask the server 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 the application runs.

- Create stateless session beans instead of stateful session beans as much as possible. If you must create stateful session beans, have the server 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 is used in many applications.

In addition to these general considerations, decide which parts of an application are candidates for entity and session beans.

Using Session Beans

Session beans are intended to represent transient objects and processes, such as a single database record, a document copy 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 the conversational state. Session beans that maintain the conversational state are called stateful session beans; beans that do not are called stateless session beans.

When a client is done with the session objects, the objects are released. When designing an application, designate each temporary, single client object as a potential session bean. For example, in an online shopping application each shopping cart is a temporary object. The cart lasts only as long as the customer selects 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 can also provide transaction settings. These transaction settings and JDBC calls are referenced by the session bean's container, which is transparent. The container provided with the iPlanet Application Server 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 iPlanet Application Server application, see Chapter 5, "Using Session EJBs to Manage Business Rules."

Using Entity Beans

Entity beans commonly represent persistent data. This data is maintained directly in a database or accessed through an EIS application as an object. A simple example of an entity bean is one defined to represent a single row in a database table and where each bean instance represents a specific row. A more complex example is an entity bean designed to represent complicated views of joined tables in a database where each bean instance represents the contents of a single shopping cart.

Unlike session beans, entity bean instances are accessed simultaneously by multiple clients. The container is responsible for synchronizing the instance state by transactions in use. This responsibility delegation to the container means that the bean developer does not need to consider concurrent access methods from multiple transactions.

An entity bean's persistence can either be managed by the bean or the container. When an entity bean manages its own persistence, it's called Bean Managed Persistence. When the bean delegates this to the container, it's called Container Managed Persistence (CMP).

- **Bean Managed Persistence:** the bean developer implements persistence code (such as JDBC calls) directly in the EJB class methods for bean managed persistence. The possible downside is portability loss, if a proprietary interface is used, and the risk of tying the bean to a specific database.
- **Container Managed Persistence:** the container provider uses the Deployment Tool to implement the container persistence. The container transparently manages the persistence state. Therefore, you do not need to implement any data access code in the bean methods. Not only is this method simpler 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 iPlanet Application Server application, see Chapter 6, "Building Entity EJBs."

Planning for Failover Recovery

Failover recovery is a process in which a bean can reinstantiate itself after a server crash. Both stateless and stateful session beans support failover recovery. The Deployment Tool is used to set the failover properties for session beans; for a description of these settings see the *Deployment Tool Online Help*. 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. For more information, see “Using Finder Methods,” on page 140.

Working with Databases

In an iPlanet Application Server, the preferred method for working with databases is through the JDBC API in conjunction with transaction attributes. Use the Java Naming and Directory Interface (JNDI) to obtain a database connection. JNDI provides a standard way for applications 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 iPlanet Application Server 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

Deploy EJBs with the rest of an application using the Deployment Tool. For more information on how to deploy EJBs, see the *Deployment Tool Online Help*. For information on property settings made by the Deployment Tool and how they effect an application, see Chapter 10, “Deployment Packaging.”

Dynamically Reloading EJBs

EJB reloading in an iPlanet Application Server is done without restarting the server by simply redeploying the EJB. The iPlanet Application Server notices the new component and reloads it within 10 seconds. For more information, see Appendix B, “Dynamic Reloading.”

NOTE This feature is turned off by default for a production environment. Turn it on when needed.

Chapter 5

Using Session EJBs to Manage Business Rules

This chapter describes how to create session EJBs that encapsulate an application's business rules and logic. 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 code units that perform repetitive, time-bound, and user-dependent tasks. These tasks can be simple or complex, and are often needed in different applications. For example, banking applications must verify a user's account ID and balances before performing any transaction. These tasks define the business rules and logic that you use to run your business. Such discrete tasks, transient by nature, are candidates for session EJBs.

Session EJBs are self-contained code units that represent client-specific generic object instances. These objects are transient in nature, created and freed throughout an application's life on an as-needed basis. For example, the *shopping cart* employed by many web-based, online shopping applications is a typical session bean. It is created by the online shopping application only when an item is chosen.

When an item selection is completed, the item prices in the cart are calculated, the order is placed, and the shopping cart object is freed. A user can continue browsing merchandise in the online catalogue, and if the user decides to place another order, a new shopping cart is created.

Often, a session bean 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 available items 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, but uses a conversational state to preserve information about its contents and values between client calls. The conversational state enables the bean's container to maintain information about the session bean state and to recreate the 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 application that runs 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 data snapshot. However, a bean can 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. 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 session bean instances 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 `java.ejb.EJBObject` interface methods. This includes methods to obtain the session bean's home interface, 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. In addition, most EJB objects also support specific business logic methods. These methods are at the heart of an application.

All specifications are accessible from *install_dir*`/ias/docs/index.htm`, where *install_dir* is the location where the iPlanet Application Server is installed.

Session Bean Components

When programming 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 metadata (Deployment Descriptors (DDs) 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 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 bean's home interface, to retrieve the bean's handle (a 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 used, see the EJB specification. All specifications are accessible from *install_dir*`/ias/docs/index.htm`, where *install_dir* is the location where the iPlanet Application Server is installed.

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 (through `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 must be one method for each way a client invokes 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 must be legal Java RMI types. The `throws` clause may define application specific exceptions and 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 a session bean manages user logins, it might include a unique function called `ValidateLogin()`.

Business method names can be anything, but must not conflict with the method names 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 *install_dir*`/ias/docs/index.htm`, where *install_dir* is the location where the iPlanet Application Server is installed.

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 deployment descriptor. For more information, see "EJB XML DTD," on page 247.

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," on page 247.

For more information about passivation, see the EJB specification. All specifications are accessible from *install_dir*`/ias/docs/index.htm`, where *install_dir* is the location where the iPlanet Application Server is installed.

Creating the Home Interface

The *home* interface defines the methods that enable a client using the 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 method must be named `create`, and must correspond in number and argument types to an `ejbCreate` method defined in the session bean class. The return type for each create method, however, does not match its corresponding `ejbCreate` method's return type. 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.

NOTE Do not override these methods.

Additional Session Bean Guidelines

Before deciding which parts of an 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 the iPlanet Application Server 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 use stateful session beans, co-locate the stateful beans with their clients. Also, use *sticky load balancing* to reduce the number of RPCs, 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 iPlanet Application Server Functionality

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

The iPlanet Application Server offers several features through the iPlanet Application Server container, and the iPlanet Application Server APIs enable applications to take programmatic advantage of specific iPlanet Application Server environment features. Embed API calls in session beans if you plan on using those beans only in an iPlanet Application Server environment.

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

1. First obtain a `com.kivasoft.IContext` instance 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.

In the iPlanet Application Server, direct references are also serializable. If you take advantage of this extension, be aware that not all vendors support it.

Managing Transactions

Many session beans interact with databases. You control bean transactions by using settings in the bean's property file. This permits specifying transaction attributes at bean deployment time. By having a bean handle transaction management there is no need to explicitly *start*, *rollback*, 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 the bean's activities—even those not directly tied to the database access—under the same transaction control as your database calls. This guarantees that all application parts controlled by a session bean run as part of the same transaction, and either everything the bean undertakes is committed, or it is rolled back in a failure case. In effect, a bean managed transactional state permits synchronizing the application without programming 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 the application processes, commit time also affects the session bean state. The iPlanet Application Server container implements *commit option C* as described in the EJB specification.

When a commit occurs, it signals the container that the session bean has completed its useful work and tells the container to synchronize its state with the underlying datasource. 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 datasource.

Note that transactions from the container are implicitly committed. Also, any participant can rollback a transaction. For details about transactions, see Chapter 7, “Handling Transactions with EJBs.”

Accessing Databases

Many session beans access and update data. Because session beans are transient, be careful about how accesses occur. In general, use the JDBC API to make calls, and always use the transaction and security management methods described in Chapter 7, “ Handling Transactions with EJBs” to manage the transaction isolation level and transaction requirements at the bean level.

For details about database accesses, see Chapter 8, “Using JDBC for Database Access.”

Session Bean Failover

The session bean failover feature allows conversational state recovery for stateful session beans when an iPlanet Application Server becomes unavailable due to a service loss. Supporting failover for stateful session beans is an iPlanet Application Server value-added feature. J2EE programs do not need any modification to support the iPlanet Application Server failover feature. Failover is handled by the container and is defined by the deployer in the deployment descriptor.

Imagine a corporate buyer performing online 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 iPlanet Application Server instance becomes unavailable. Without failover capability, the failure would result in the buyer’s shopping cart becoming empty; the stateful session bean’s state 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 iPlanet Application Server instance that has the bean’s state before the failure. The buyer’s shopping cart contains the same selected items as it did before the failover took place.

Notable failover feature support for stateful session beans includes:

- Failover is a value-added feature that supports J2EE programs.
- Failover is transparent to the client; no special APIs are required.
- Failover is handled by the container and configured by the deployer.
- Distributed Store (DSync) is the enabling mechanism for restoring the 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 is a combination of configuring the bean with failover and DSync.

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

To take advantage of the failover feature, the bean must be configured with both failover and DSync. The DSync mechanism saves the session bean's conversational state during runtime. The failover mechanism allows the container to detect a system failure and connects to another running iPlanet Application Server instance that has the saved session bean state.

For more information, see the *Administrator's Guide* for details on how to configure a stateful session bean with failover during deployment and how to configure DSync during runtime. For more information on configuring DSync during installation, see the *Installation Guide.*

How the Failover Process Works

Stateful bean failover 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 method invocation, the smart stubs detect failures and transparently relocate a bean to a new home potentially in a different engine. The stubs determine if the bean's reference has become stale by getting a connection exception from the dead bean. The stubs then do a home look up and obtain the remote interface. Once the bean is relocated, the stubs retry 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 bean state. The bean state is saved at regular intervals and is automatically reinstated as part of the recovery process.

For more information on the deployment descriptors used by stateful session beans for failover, see Chapter 10, “Deployment Packaging.”

Failover Guidelines

Keep in mind the following 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 its original home may be preserved after failover.
- Use judgement by carefully weighing the advantages of bean failover against the failover process performance cost.

NOTE Do not configure every stateful bean with failover.

- Remember, session bean state is conversational. Use entity beans for transactional data.
- The time interval for saving a stateful session bean’s state is configurable using the Administration Tool (under the EJB tab); the default is 10 seconds.
- If the bean is transactional, timer-based state saving is automatically disabled during transactions. This ensures transactional data integrity in case of a server engine failure during the transaction. Transactional database updates are rolled back by the database if a failure occurs. The state of the recovered bean is whatever it was at the begining of the failed transaction. However, if the transaction proceeds smoothly, the bean state is saved when the transaction completes, and timer-based saving resumes until the next transaction begins.
- If the bean implements the iPlanet Application Server provided `com.netscape.server.ejb.IEBFoStateModification` interface, the state saver can check if the state of the bean is modified or not before it performs the expensive save operation. This interface defines two methods:

```
package com.netscape.server.ejb;

public interface IEBFoStateModification {
```

```
    /**
     ** This method is called by the container to check if a bean
     ** instance is dirty.
     **/

    boolean isDirty();

    /**
     ** Sometimes the container performs immediate saves. Then it
     ** calls to reset the dirty state of the modified bean
     **/

    void setDirty(boolean dirty);

}
```

The user-supplied bean implementation has a boolean variable that tracks the modified state of the bean. This variable is consulted prior to any state saving.

How Often Is the State Saved?

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

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

The regular time interval is configured in the Administration Tool.

How the State Is Saved

The process for state saving is as follows:

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

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

Chapter 6

Building Entity EJBs

This chapter describes what an entity EJB is and what entity beans must contain. This chapter also provides additional guidelines for creating entity beans and for determining what the entity bean's needs are in an application.

This chapter contains the following sections:

- Introducing Entity EJBs
- Entity Bean Components
- Additional Entity Bean Guidelines
- Container Managed Persistence

All specifications are accessible from *install_dir*`/ias/docs/index.htm`, where *install_dir* is the location where the iPlanet Application Server is installed.

Introducing Entity EJBs

The heart of a distributed, multi-user application involves interactions with datasources which are often transactional, such as a database or an existing legacy application. In most cases, the external datasource or business object is transparent to the user, or is shielded or buffered from direct user interactions. These protected, transactional, and persistent interactions with databases, documents, and other business objects are candidates for entity EJB encapsulation.

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

There are two persistence management types, and the iPlanet Application Server supports both types as listed below.

- **Container managed persistence** – this is when the container is responsible for the bean persistence.
- **Bean managed persistence** – this is when the bean is responsible for their own persistence.

A developer codes a bean managed entity bean by providing database access calls—through JDBC or SQL—directly in the bean class methods. Database access calls must be in the `ejbCreate()`, `ejbRemove()`, `ejbFind`*`XXX`*`()`, `ejbLoad()`, and `ejbStore()` methods. The bean managed persistence advantage is that these beans can be in 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 accesses through transactions.

As an application developer, you cannot access the container's entity bean services directly, nor do you ever need to. Instead, the container is there to take care of low level implementation details so you can focus on the larger role the entity bean plays in an 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 entry point to a data resource or business object, such as a database. Regardless of which client accesses an entity object at a given time, each client's object view 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 looks up an entity bean's home interface through the Java Naming and Directory Interface (JNDI).

An entity bean includes the following attributes:

- 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 an entity bean instance 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 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. In addition, the remote interface for most EJB objects also supports specific business logic methods. These are the methods at the heart of a specific application.

All specifications are accessible from *install_dir*`/ias/docs/index.htm`, where *install_dir* is the location where the iPlanet Application Server is installed.

Entity Bean Components

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

- Enterprise bean class
- Enterprise bean home interface, implementing `javax.ejb.EJBHome`
- Enterprise bean remote interface, implementing `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 RemoteException {
}
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() throws 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 define one `ejbPostCreate()` method for each `ejbCreate()` method. It may provide additional, developer defined finder methods that take the form `ejbFind`*XXX*, where *XXX* represents a unique method name continuation (for example, `ejbFindApplesAndOranges`) that does not duplicate any other method 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, but must not conflict with the method names used in the EJB architecture. Business methods must be declared as `public`. Method arguments and return value types must be Java RMI legal. The `throws` clause may define application specific exceptions and may include `java.rmi.RemoteException`.

There are two business method types to implement in an entity bean:

- internal ones, which are used by other business methods in the bean, but are never accessed outside the bean itself
- external ones, which are referenced by the entity bean's remote interface

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

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

If, specific application tasks need to be performed when a bean is first made ready for an application or needs to be performed when a bean is no longer needed, program 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.

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," on page 247.

For more information about `ejbCreate()` and `ejbRemove()`, see "Using ejbCreate Methods," on page 139.

For more information about `ejbActivate()` and `ejbPassivate()`, see the EJB specification. All specifications are accessible from *install_dir*`/ias/docs/index.htm`, where *install_dir* is the location where the iPlanet Application Server is installed.

Using ejbLoad and ejbStore

An entity bean should permit its container to store the bean 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()` method definitions that 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();
}
```

For more information about bean isolation levels that access transactions concurrently with other beans, see “Handling Concurrent Access,” on page 145.

Using setEntityContext and unsetEntityContext

A container calls `setEntityContext()` after it creates an entity bean instance to provide the bean’s interface to the container. Implement this method, to store the container reference 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 eventually calls `finalize()` on the instance to clean it up and dispose it.

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

Using ejbCreate Methods

The entity bean must also implement one or more `ejbCreate(...)` methods. There must be one 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 converts to a number for key purposes. Any arguments must be legal Java RMI types. The `throws` clause, may define application specific exceptions, and may include `java.rmi.RemoteException` and/or `java.ejb.CreateException`.

For each `ejbCreate()` method, the entity bean class may 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 argument type of its corresponding `ejbCreate` method. The `throws` clause, may define application specific exceptions, and may include `java.rmi.RemoteException` and/or `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 one instance instantiated at the same time, an entity bean must implement at least one method, `ejbFindByPrimaryKey()`, 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. Implement the `ejbFindByPrimaryKey()` 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 find a specific entity bean instance based on what the bean does, based on certain values the instance is working with, or based on other criteria. These implementation specific finder method names take the form `ejbFindXXX`, where `XXX` represents a unique continuation of a method name (for example, `ejbFindAppleSAndOranges`) that does not duplicate any other method names.

Finder methods must be declared as `public`, and their arguments and return values must be legal Java RMI types. Each finder method return type 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 is an application specific exception, and 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 permits the bean class to implement the remote interface's methods, but recommends against this practice to avoid inadvertently passing a direct reference (through `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 enables a client accessing an 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;
}
```

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 method must be named create, and must correspond in number and argument types to an `ejbCreate` method defined in the entity bean class. The return type for each create method, however, does not match the corresponding `ejbCreate` method's return type. 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 method must be named `find`*XXX* (for example, `findApplesAndOranges`), where *XXX* is a unique method name continuation. Each finder method must correspond to one of the finder methods defined in the entity bean class definition. The number and argument types must also correspond to the finder method definitions in the bean class. The return type, however, may be different. The finder method's return type 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.

NOTE 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. 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 enables you to retrieve the bean's home interface, 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 used, see the EJB specification. All specifications are accessible from *install_dir*`/ias/docs/index.htm`, where *install_dir* is the location where the iPlanet Application Server is installed.

Additional Entity Bean Guidelines

Before you decide what application parts you can represent as entity beans, you should consider a few more guidelines. A couple of these are related to the EJB specification for entity beans, and a couple are specific to the iPlanet Application Server and its support for entity beans.

Accessing iPlanet Application Server Functionality

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

The iPlanet Application Server offers several features through the iPlanet Application Server container and the iPlanet Application Server APIs that enables your applications to take programmatic advantage of specific iPlanet Application Server environment features. You can embed API calls in your entity beans if you plan on using those beans only in an iPlanet Application Server 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.

The iPlanet Application Server direct references are also serializable. You may wish to take advantage of this extension, but be aware 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 to explicitly start, rollback, 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 bean activities—even those not directly tied to database access—under the same transaction control as your database calls. This guarantees that all application parts controlled by an entity bean run as part of the same transaction, and either everything the bean undertakes is committed, or is rolled back because of a failure. In effect, bean managed transactional state permits you to synchronize an application without having to code any synchronization routines.

Committing a Transaction

When a commit occurs, it signals the container that the entity bean has completed its useful work and should synchronize its state with the underlying datasource. 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 datasource.

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

Commit Option C

Commit option C is supported by the iPlanet Application Server. Commit option C gets a bean instance 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.

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 gets the ready instance and subsequent concurrent clients get new instances from the pool.

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 an iPlanet Application Server, the container activates one entity bean instance for each simultaneously occurring transaction that uses the bean. Transaction synchronization is performed automatically by the underlying database during database access calls.

The iPlanet Application Server EJB container implementation does not provide its own synchronization mechanism when multiple transactions try to access an entity bean. It creates a new entity bean instance for every new transaction. The iPlanet Application Server container delegates the responsibility of the application synchronization.

You typically perform this synchronization in conjunction with the underlying database or resource. One 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 being used. For more information, see the EJB specification as it relates to concurrent access.

The following example `ejbLoad()` snippet illustrates the `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, and always use the transaction and security management methods in `javax.ejb.deployment.ControlDescriptor` to manage transaction requirements. For details 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 details about using JDBC to work with data, see Chapter 8, “Using JDBC for Database Access.”

Container Managed Persistence

An entity bean using container-managed persistence (CMP) defers the management of its state (or persistence) to the iPlanet Application Server. Normally, CMP beans persist to a relational database.

Developers use CMP to simplify the work of creating an entity bean. Rather than write all the JDBC code that is necessary to implement a BMP entity bean, a developer using CMP simply uses tools to create the bean’s deployment descriptors. The deployment descriptors contain information that the container uses to map fields to the bean to columns in a relational database.

For more information on CMP, see Chapter 9.4 of the EJB 1.1 specification.

The iPlanet Application Server provides the following support for CMP entity beans:

- Full support for the J2EE v 1.2 specification’s CMP model (i.e., EJB 1.1).

- Support for third party O/R mapping tools.
- An “out-of-the-box” lightweight implementation of CMP. Lightweight CMP provides:
  - A basic object-to-relational (O/R) mapping tool within the iPlanet Application Server Deployment Tool that creates XML Deployment Descriptors for each CMP bean.
  - Support for compound (multi-column) primary keys.
  - Support for sophisticated custom finder methods.
  - Standards-based query language (SQL92).

Full J2EE Support

The iPlanet Application Server fully supports the Entity Bean Component Contract defined in the EJB 1.1 specification. Here are a few items of interest:

- The iPlanet Application Server implements commit option C as defined in the EJB 1.1 specification.
- The primary key class must be a subclass of `java.lang.Object`. This is in accordance with the specification, and ensures portability, but is noted because a few vendors allow primitive types (such as `int`) to be listed as the primary key class.

Third Party O/R Mapping Tools

iPlanet Application Server certifies third party enterprise tool vendors for use. In general, third-party CMP solutions that fully support the EJB1.1 specification work with iPlanet Web Server.

For example, Thought, Inc., provides CocoBase Enterprise as a sophisticated O/R mapping solution for mapping EJBs to relational databases. To use Cocobase, you build your EJBs using Cocobase’s O/R Mapping Tool, then deploy the beans using the iPlanet Application Server’s Deployment Tool or the iPlanet Application Server’s Command Line Interface (CLI).

Several other vendors are currently completing certification. Check the website (`developer.iplanet.com`) for current information on certified third party O/R mapping tools.

Full Example of a CMP Entity Bean

For a full example of a CMP entity bean, look at the Product sample application from the *J2EE Developer's Guide*, which is available here:

install_dir`/ias/ias-samples/j2eeguide/product`

Using the Lightweight CMP Implementation

The iPlanet Application Server provides an "out-of-the-box" lightweight CMP implementation. The implementation includes a mapping tool, found in the iPlanet Application Server Deployment Tool, and a CMP runtime environment. The CMP runtime environment creates persistence managers for each CMP bean. The persistence managers then use information specified in XML deployment descriptors. The three deployment descriptors used in a CMP bean are:

- `ejb-jar.xml` - There is one `ejb-jar.xml` file in each EJB module. This deployment descriptor is covered in detail in the EJB 1.1 specification.
- `ias-ejb-jar.xml` - Like the `ejb-jar.xml` file, there is just one `ias-ejb-jar.xml` file per EJB module. To use lightweight CMP, you must set some properties in this file. For an overview of the DTD, see Chapter 10, "Deployment Packaging."
- *property-file-name*`.xml` - Additionally, each CMP Bean has its own deployment descriptor. The name of the file is specified in the `ias-ejb-jar.xml` file. The contents of this file determine how the reference implementation's persistence managers load and store each bean's state in a relational database.

There are two ways to generate these files. The following sections cover each method in detail:

- Creating the Deployment Descriptors by Hand
- Using the Deployment Tool

Creating the Deployment Descriptors by Hand

It's easiest to understand what's happening in the iPlanet Application Server's Deployment Tool if you know what's happening behind the scenes, so the manual steps are explained first.

ejb-jar Deployment Descriptor

The `ejb-jar.xml` file is described in detail in the EJB 1.1 specification. The ejb-jar deployment descriptor is where important information like the transactional attributes of the beans and the fields of a bean that are going to be container-managed are specified. Any J2EE-compliant ejb-jar file is deployable on the iPlanet Application Server if you provide a corresponding `ias-ejb-jar.xml` file.

ias-ejb-jar Deployment Descriptor

The J2EE vendor-specific information for Enterprise JavaBeans is stored in another deployment descriptor, `ias-ejb-jar.xml`. The Document Type Definition (DTD) for this XML-based deployment descriptor is covered in detail in Chapter 10, "Deployment Packaging."

There is some information that is unique to CMP beans that goes in this file, however, within the `<persistence-manager>` element:

- The fully qualified class name for the factory class that creates persistence managers is specified in the `<factory-class-name>` element. The factory class name for the reference implementation is `com.netscape.server.ejb.SQLPersistenceManagerFactory`.
- The relative path of the CMP bean's specific property file within the `ejb-jar.xml` file is specified in the `<properties-file-location>` element.

Here's a code snippet of what the relevant elements of the deployment descriptor look like:

```
...
   <persistence-manager>
      <factory-class-name>
         com.netscape.server.ejb.SQLPersistenceManagerFactory
      </factory-class-name>
      <properties-file-location>
         META-INF/MyProduct-ias-cmp.xml
      </properties-file-location>
   </persistence-manager>
...
```

CMP Bean Deployment Descriptor

The file name of the CMP bean-specific deployment descriptor is specified in the `ias-ejb-jar.xml` file. In the previous example, the properties file would be named `MyProduct-ias-cmp.xml`. The file's root element is the `<ias-persistence-manager>` node, but the rest is a simple bean property file. The file uses a simple XML format to describe various properties. The tags in the xml file follow this basic format:

```
<bean-property>
   <property>
      <name></name>
      <type></type>
      <value></value>
      <delimiter></delimiter>
   </property>
</bean-property>
```

Here are descriptions of the subelements of `<property>`:

`name`	is one of these valid names: `dataSource`, `allFields`, `findByPrimaryKeySQL`, `findByPrimaryKeyParms`, `insertSQL`, `insertParms`, `deleteSQL`, `deleteParms`, `loadSQL`, `loadParms`, `loadResults`, `storeSQL`, `storeParms`, or the name of a custom finder. Each of these properties is described later in this section.
`type`	is either `java.lang.String` or `java.util.Vector`. If `Vector` is used as the type, the value is treated as a comma-delimited list.
`value`	is any string.
`delimiter`	is always `,` (a comma).

The following properties are defined in the lightweight CMP bean's deployment descriptor:

- Data Source (`dataSource`)
- CMP field to RDB column mapping (`allFields`)
- Persistence operations:
  - findByPrimaryKey (`findByPrimaryKeySQL` and `findByPrimaryKeyParms`)
  - insert (`insertSQL` and `insertParms`)

- delete (deleteSQL and deleteParms)
- load (loadSQL, loadParms, loadResults)
- store (storeSQL and storeParms)
- custom finders (optional)

Data Source

The first property used in the XML file is the dataSource property. The value of the dataSource property is the JNDI name of the JDBC data source used as a persistent store. For example:

```
...
   <bean-property>
      <property>
         <name>dataSource</name>
         <type>java.lang.String</type>
         <value>j2eeguide/ProductDB</value>
         <delimiter>,</delimiter>
      </property>
   </bean-property>
...
```

CMP Field to RDB Column Mapping

The allFields property is where the O/R mapping is specified. In the value element, a bracket-enclosed String maps the CMP fields to database columns. CMP fields go to the left side of the =, while database columns go to the right of the expression. A semicolon, ;, must separate the expressions. For example:

```
...
   <bean-property>
      <property>
         <name>allFields</name>
         <type>java.lang.String</type>
         <value>
      {description=DESCRIPTION;price=PRICE;productId=PRODUCTID;}
         </value>
         <delimiter>,</delimiter>
      </property>
   </bean-property>
...
```

Persistence Operations

Persistence operations consist of three types of properties. These properties follow the following naming patterns:

- **xxxxSQL** is an SQL statement for a particular persistence operation (such as insert). The SQL statement in the `xxxxSQL` property is used to create a `java.sql.PreparedStatement`; therefore, the SQL statement should conform to the rules specified for parameterized queries (for example, use ? to signify a parameter). To understand how to map your CMP fields to SQL datatypes, see "Mapping Rules," on page 159.
- **xxxxParms** is a list of parameters that are sent to the persistence operation. The first field maps to the first parameter in the SQL statement (denoted by ?), the second maps to the second, and so on.
- **xxxxResults** is a list of the fields in the `ResultSet` that are returned from the execution of the `PreparedStatement`.

The xxxx part of the name can be one of the following:

- findByPrimaryKey (`findByPrimaryKeySQL` and `findByPrimaryKeyResults`)
- insert (`insertSQL` and `insertParms`)
- delete (`deleteSQL` and `deleteParms`)
- load (`loadSQL`, `loadParms`, `loadResults`)
- store (`storeSQL` and `storeParms`)
- The name of a custom finder

Persistence operation properties vary based on whether their CMP bean has a single-field primary key or a multi-field primary key. Where there is a difference, it is noted in the following examples.

findByPrimaryKey

The findByPrimaryKey properties are `findByPrimaryKeySQL` and `findByPrimaryKeyParms`. It is not necessary to provide the `findByPrimaryKeyResults` property for the `findByPrimaryKey` property because it is already defined in the primary key class. This operation corresponds to the `findByPrimaryKey()` method in the EJB's home interface.

Here is a single-field primary key example:

```
...
   <bean-property>
      <property>
         <name>findByPrimaryKeySQL</name>
         <type>java.lang.String</type>
         <value>
            SELECT PRODUCTID FROM PRODUCT WHERE PRODUCTID = ?
         </value>
```

```
            <delimiter>,</delimiter>
         </property>
      </bean-property>
      <bean-property>
         <property>
            <name>findByPrimaryKeyParms</name>
            <type>java.util.Vector</type>
            <value>productId</value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
...
```

Here is a multi-field primary key example:

```
...
      <bean-property>
         <property>
            <name>findByPrimaryKeySQL</name>
            <type>java.lang.String</type>
            <value>
SELECT PRODUCTID, DESCRIPTION FROM PRODUCT WHERE PRODUCTID = ? AND DESCRIPTION = ?
            </value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
      <bean-property>
         <property>
            <name>findByPrimaryKeyParms</name>
            <type>java.util.Vector</type>
            <value>productId,description</value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
...
```

insert

The insert properties are `insertSQL` and `insertParms`. Inserts are exactly the same for single- and multi-field primary keys. This property corresponds to the bean's `create()` method in its home interface.

```
...
      <bean-property>
         <property>
            <name>insertSQL</name>
            <type>java.lang.String</type>
            <value>
INSERT INTO PRODUCT ( DESCRIPTION,PRICE,PRODUCTID ) VALUES(?,?,?)
```

```
            </value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
      <bean-property>
         <property>
            <name>insertParms</name>
            <type>java.util.Vector</type>
            <value>description,price,productId</value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
...
```

delete

The delete properties are `deleteSQL` and `deleteParms`. The delete operation provides the functionality for the `remove()` function in the bean's home interface.

Here is a single-field primary key example:

```
...
   <bean-property>
      <property>
         <name>deleteSQL</name>
         <type>java.lang.String</type>
         <value>DELETE FROM PRODUCT WHERE PRODUCTID = ?</value>
         <delimiter>,</delimiter>
      </property>
   </bean-property>
   <bean-property>
      <property>
         <name>deleteParms</name>
         <type>java.util.Vector</type>
         <value>productId</value>
         <delimiter>,</delimiter>
      </property>
   </bean-property>
...
```

Here is a multi-field primary key example:

```
...
   <bean-property>
      <property>
         <name>deleteSQL</name>
         <type>java.lang.String</type>
         <value>
      DELETE FROM PRODUCT WHERE PRODUCTID = ? AND DESCRIPTION = ?
```

```
            </value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
      <bean-property>
         <property>
            <name>deleteParms</name>
            <type>java.util.Vector</type>
            <value>productId,description</value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
...
```

load

The load properties are `loadSQL`, `loadParms`, and `loadResults`. Load operations are almost identical for single- and multi-field primary keys. There is a minor difference in the `loadSQL` property and, consequently, the `loadParms` property. Load operations correspond to the EJB's `ejbLoad()` method.

Here is a single-field primary key example:

```
...
      <bean-property>
         <property>
            <name>loadSQL</name>
            <type>java.lang.String</type>
            <value>
SELECT DESCRIPTION,PRICE,PRODUCTID FROM PRODUCT WHERE PRODUCTID = ?
            </value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
      <bean-property>
         <property>
            <name>loadParms</name>
            <type>java.util.Vector</type>
            <value>productId</value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
      <bean-property>
         <property>
            <name>loadResults</name>
            <type>java.util.Vector</type>
            <value>description,price,productId</value>
```

```
            <delimiter>,</delimiter>
        </property>
    </bean-property>
...
```

Here is a multi-field primary key example:

```
...
    <bean-property>
        <property>
            <name>loadSQL</name>
            <type>java.lang.String</type>
            <value>
SELECT DESCRIPTION,PRICE,PRODUCTID FROM PRODUCT WHERE PRODUCTID = ? AND DESCRIPTION = ?
            </value>
            <delimiter>,</delimiter>
        </property>
    </bean-property>
    <bean-property>
        <property>
            <name>loadParms</name>
            <type>java.util.Vector</type>
            <value>productId,description</value>
            <delimiter>,</delimiter>
        </property>
    </bean-property>
    <bean-property>
        <property>
            <name>loadResults</name>
            <type>java.util.Vector</type>
            <value>description,price,productId</value>
            <delimiter>,</delimiter>
        </property>
    </bean-property>
...
```

store

The store properties are `storeSQL` and `storeParms`. As with load properties, there are small differences in the `storeSQL` and `storeParms` properties. Make sure you get the order right in the multi-field primary key. The store operation is performed when the EJB container calls the `ejbStore()` method on the bean implementation.

Here is a single-field primary key example:

```
...
    <bean-property>
        <property>
            <name>storeSQL</name>
```

```
            <type>java.lang.String</type>
            <value>
        UPDATE PRODUCT SET DESCRIPTION=?,PRICE=? WHERE PRODUCTID = ?
            </value>
            <delimiter>,</delimiter>
        </property>
    </bean-property>
    <bean-property>
        <property>
            <name>storeParms</name>
            <type>java.util.Vector</type>
            <value>description,price,productId</value>
            <delimiter>,</delimiter>
        </property>
    </bean-property>
...
```

Here is a multi-field primary key example:

```
...
    <bean-property>
        <property>
            <name>storeSQL</name>
            <type>java.lang.String</type>
            <value>
UPDATE PRODUCT SET PRICE=? WHERE PRODUCTID = ? AND DESCRIPTION = ?
            </value>
            <delimiter>,</delimiter>
        </property>
    </bean-property>
    <bean-property>
        <property>
            <name>storeParms</name>
            <type>java.util.Vector</type>
            <value>price,productId,description</value>
            <delimiter>,</delimiter>
        </property>
    </bean-property>
...
```

Custom Finders

Optionally, you can add custom finders to the deployment descriptor. Custom finder operations follow slightly different rules than other operations:

- For the `xxxxSQL` property of custom finders, the first argument to the finder method defined in the home interface maps to the first parameter in the SQL statement, the second to the second, and so on.

- The `xxxxResults` property for custom finders maps the columns of the `ResultSet` of the SQL statement to the primary key's fields (for multi-field primary keys) or to the primary key itself (for single-field primary keys).

For example, suppose the following method is defined in an entity bean's home interface:

```
public Collection findInRange(double low, double high)
    throws FinderException, RemoteException;
```

The name of the property is the name found in the bean's home interface. In this example, this operation would have up to three properties in the deployment descriptor: `findInRangeSQL`, `findInRangeParms`, and `findInRangeResults` (needed only for a multi-field primary key).

Here are the properties that implement this operation for a single-field primary key:

```
...
    <bean-property>
        <property>
            <name>findInRangeSQL</name>
            <type>java.lang.String</type>
            <value>
            SELECT PRODUCTID FROM PRODUCT WHERE PRICE BETWEEN ? AND ?
            </value>
            <delimiter>,</delimiter>
        </property>
    </bean-property>
    <bean-property>
        <property>
            <name>findInRangeParms</name>
            <type>java.lang.Vector</type>
            <value>low,high</value>
            <delimiter>,</delimiter>
        </property>
    </bean-property>
...
```

Here are the properties that implement this operation for a multi-field primary key:

```
...
    <bean-property>
        <property>
            <name>findInRangeSQL</name>
            <type>java.lang.String</type>
            <value>
    SELECT PRODUCTID, DESCRIPTION FROM PRODUCT WHERE PRICE BETWEEN ? AND ?
            </value>
```

```
            <delimiter>,</delimiter>
         </property>
      </bean-property>
      <bean-property>
         <property>
            <name>findInRangeParms</name>
            <type>java.lang.Vector</type>
            <value>low,high</value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
      <bean-property>
         <property>
            <name>findInRangeResults</name>
            <type>java.util.Collection</type>
            <value>productid,description</value>
            <delimiter>,</delimiter>
         </property>
      </bean-property>
...
```

Mapping Rules

Lightweight CMP uses JDBC (specifically, the setter methods of the `PreparedStatement` interface) to map CMP fields to columns in a relational database table. Therefore, standard JDBC mapping rules apply to CMP fields.

For example, to map a `java.lang.String` to an SQL column, Lightweight CMP uses the `setString` method in the `PreparedStatement` interface. The documentation for the `PreparedStatement` interface specifies that `setString` maps to a `VARCHAR`.

Lightweight CMP supports all native Java field types, all the classes that represent native types (such as Integer), java.lang.String, java.sql.Date, java.sql.Time, java.sql.Timestamp, and arbitrary serializable objects. Table 6-1 describes the mappings between bean attributes and table columns.

Table 6-1 EJB/JDBC mapping

Java Type	JDBC Type	JDBC Driver Access Methods
`boolean`	`BIT`	`getBoolean()`, `setBoolean()`
`byte`	`TINYINT`	`getByte()`, `setByte()`
`short`	`SMALLINT`	`getShort()`, `setShort()`
`int`	`INTEGER`	`getInt()`, `setInt()`

Table 6-1 EJB/JDBC mapping

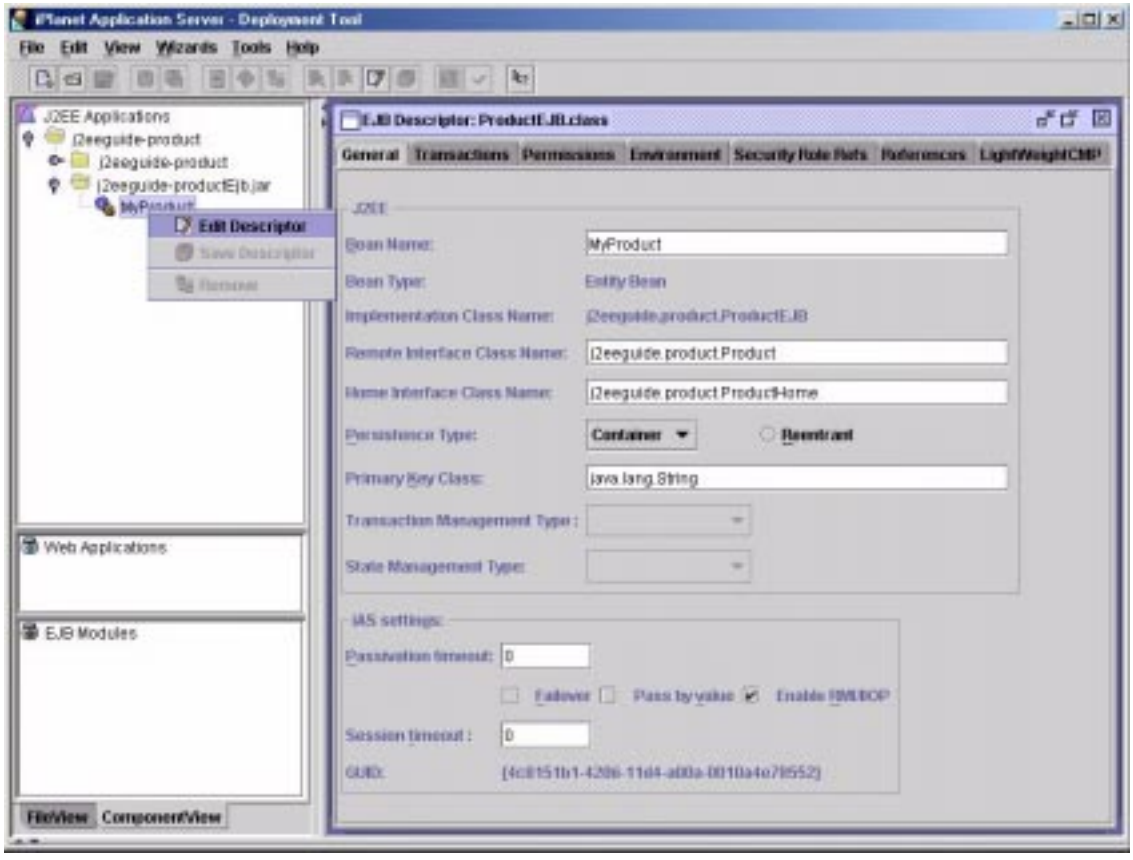
Java Type	JDBC Type	JDBC Driver Access Methods
long	BIGINT	getLong(), setLong()
float	FLOAT	getFloat(), setFloat()
double	DOUBLE	getDouble(), setDouble()
byte[]	VARBINARY or LONGVARBINARY(1)	getBytes(), setBytes()
java.lang.String	VARCHAR or LONGVARCHAR(1)	getString(), setString()
java.lang.Boolean	BIT	getObject(), setObject()
java.lang.Integer	INTEGER	getObject(), setObject()
java.lang.Long	BIGINT	getObject(), setObject()
java.lang.Float	REAL	getObject(), setObject()
java.lang.Double	DOUBLE	getObject(), setObject()
java.math.BigDecimal	NUMERIC	getObject(), setObject()
java.sql.Date	DATE	getDate(), setDate()
java.sql.Time	TIME	getTime(), setTime()
java.sql.Timestamp	TIMESTAMP	getTimestamp(), setTimestamp()
any serializable class	VARBINARY or LONGVARBINARY(1)	getBytes(), setBytes()

Using the Deployment Tool

A simpler way to create the standard ejb-jar deployment descriptors for a CMP bean is by using the iPlanet Application Server Deployment Tool. This tool's extensive built-in help goes into great detail about how to create this deployment descriptor.

Start by either opening an existing EJB Module or creating a new one. For more information on how to use this tool to create a CMP bean, refer to the help within the tool. Once the EJB's class files have been added the EJB Module, you can right-click on the bean to edit its descriptor, as in Figure 6-1.

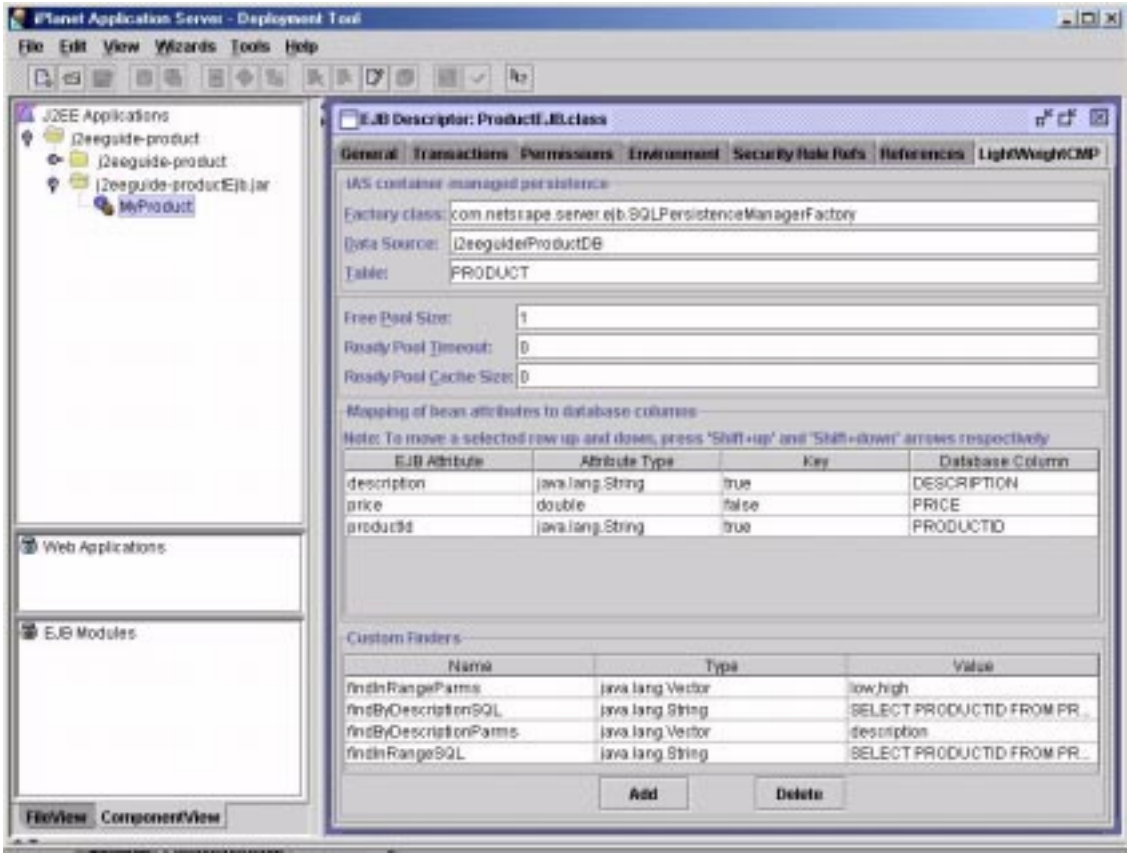
Figure 6-1 Selecting a bean in the iPlanet Application Server Deployment Tool



Once you open the deployment descriptor, the changes you make in the user interface are reflected in the CMP EJB's section of the `ejb-jar.xml` deployment descriptor, its section of the `ias-ejb-jar.xml` deployment descriptor, and the CMP bean-specific deployment descriptor. The CMP bean's mapping information is saved in a file named *ejbname*`-ias-cmp.xml`. For an in-depth description of the elements of the user interface, refer to the tool's help.

Figure 6-2 shows the Lightweight CMP tab.

Figure 6-2 The Lightweight CMP tab



After you have studied the previous section on creating a deployment descriptor by hand, everything should be familiar to you on the Lightweight CMP tab, with these exceptions:

- The TABLE text box is an input field for the relational database table that you'll be accessing via the specified data source.

- You can toggle the Key field for each EJB Attribute. To create a multi-field primary key, merely set more than one attribute to `true`; the change is reflected in the bean's corresponding deployment descriptor. (multi-field primary keys require some other modifications, namely the inclusion of a primary key class as defined in the EJB 1.1 specification).
- Use the Name, Type, and Value fields for the custom finders. Use these in the same manner as described in the previous section.

NOTE You can start creating your EJB's deployment descriptors in the iPlanet Application Server's Deployment Tool, save the application in the tool, edit the files by hand, and then go back into the tool. However, if you do this, make sure you re-open the EJB Module or J2EE Application in the tool before you edit the deployment descriptors, then re-save the application in the tool after you make the changes. If you fail to do this, your changes in the user interface are not reflected in the deployment descriptors.

Chapter 7

Handling Transactions with EJBs

This chapter describes the transaction support built-in to the EJB programming model. This chapter begins by introducing the EJB transaction model, and explains container and bean managed transactions. This chapter provides 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 contains the following sections:

- Understanding the Transaction Model
- Specifying Transaction Attributes in an EJB
- Using Bean Managed Transactions

Understanding the Transaction Model

One primary EJB advantage is the support they provide for declarative transactions. In the declarative transaction model, attributes are associated with beans at deployment time. Its 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 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()`.

Its necessary to understand the distinction between global and local transactions in order to understand the iPlanet Application Server 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 Enterprise Information System (EIS) database. Local transactions are native to a database and are restricted within a single process. Local transactions work against a single EIS only. Local transactions are typically demarcated using JDBC APIs. In an iPlanet Application Server, the container or `javax.transaction.UserTransaction()` starts all transactions.

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, configure and enable the corresponding iPlanet Application Server Resource Managers. For more information, see the *Deployment Tool Online Help* and the *Administrator's Guide*.

Specifying Transaction Attributes in an EJB

Transactional attributes are specified on a bean wide basis or on a per method basis for a bean's remote interface. If both levels specify attributes, 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 DD file. For more information, see "EJB iPlanet Application Server XML DTD," on page 249.

Using Bean Managed Transactions

While its preferable to use container managed transactions, your application requirements may necessitate using bean managed transactions. For more information on managing transactions programmatically, see 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 accesses with the iPlanet Application Server. This chapter provides high level JDBC implementation instructions for servlets and EJBs using the iPlanet Application Server; it also describes the specific iPlanet Application Server resources affected by JDBC statements when those resources have clear programming ramifications. In an iPlanet Application Server, EJBs support database access primarily through the JDBC API. The iPlanet Application Server supports the entire JDBC 2.0 API, as well as, many of the emerging JDBC 2.0 extensions, including result set enhancements, batch updates, distributed transactions, row sets, and Java Naming and Directory Interface (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 specification does not make it clear what constitute JDBC resources. In the specifications, some JDBC statements—such as, any `Connection` class methods that close database connections—release resources without specifying exactly what those resources are.

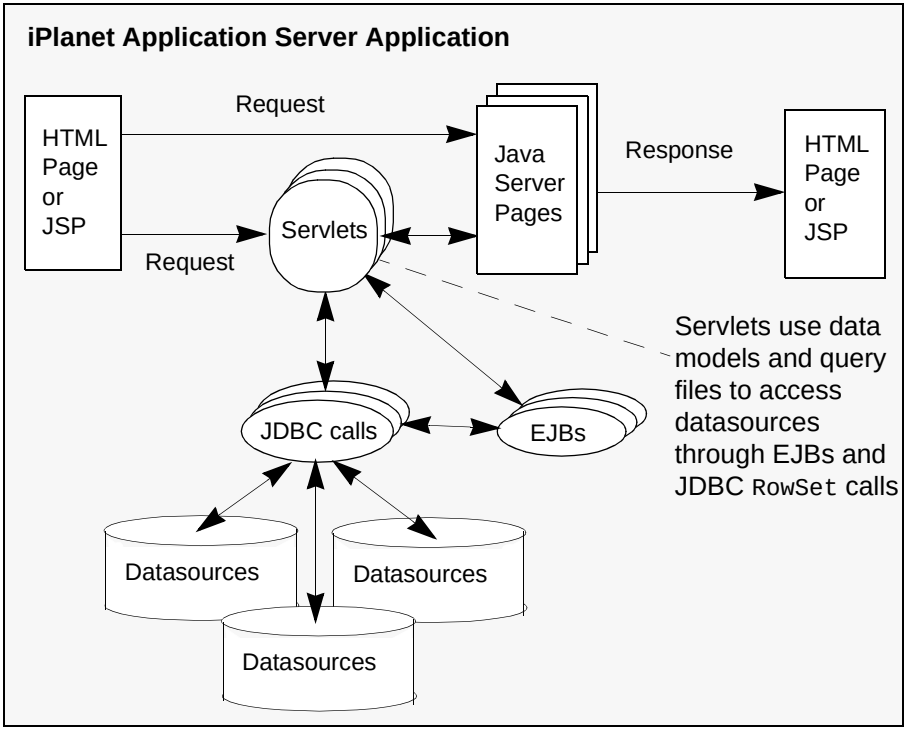
This chapter contains the following sections:

- 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 allows embedding database calls in server applications. That is all you need to know in order to start using JDBC in your server application.

More specifically, JDBC is a set of interfaces that every server vendor, such as iPlanet, must implement according to the JDBC specifications. The iPlanet Application Server provides a JDBC type 2 driver which supports a variety of Enterprise Information Systems (EISs) databases. The driver processes the JDBC statements in your application and routes the SQL arguments they contain to your database engines.



JDBC lets you write high level, easy-to-use programs that operate seamlessly with and across many different databases without you knowing most of the low level database implementation details.

Supported Functionality

The JDBC specification is a broad, database vendor independent set of guidelines. The guidelines encompass the broadest database functionality range possible in a simple framework. At a minimum, JDBC assumes the database supports the SQL-2 database access language. The JDBC specification has three parts:

- JDBC 2.0 describes the **core database access and functionality** that a server vendor must implement to be JDBC compliant. The iPlanet Application Server fully meets the compliance standard. From a database vendor's perspective, JDBC 2.0 describes a database access model that permits full access to the standard SQL-2 language, the standard language portions each vendor supports, and the language extensions each vendor implements.
- JDBC 2.0 describes **additional database access and functionality**. Primarily, this functionality involves support for newly defined SQL-3 features, data types, and mappings. The iPlanet Application Server implementation of JDBC supports most JDBC feature enhancements, but omits support for the new SQL-3 data types, such as *blobs*, *clobs*, and *arrays*. Currently, many database vendors do not fully support them in their relational database management systems. The iPlanet Application Server 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 improved database performance. The iPlanet Application Server JDBC implementation currently supports Java Naming and Directory Interface (JNDI) and *row sets*.

Understanding Database Limitations

When using JDBC in your server applications, you may encounter situations where you do not obtain the results you desire or expect. You may think the problem lies in JDBC or in the iPlanet Application Server JDBC driver implementation. However, the vast majority of these problems are limitations in your database engine.

Because JDBC covers the broadest possible 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 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 have added non standard SQL-2 extensions to their SQL implementation to support their proprietary features. JDBC provides ways to access vendor specific features, but realize these features may not be available for all databases you use.

This is especially true when you build an application that uses databases from two or more 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 maximize JDBC usage, consult your database documentation about which SQL and JDBC aspects they support. Therefore, first eliminate your database as causing the problem before calling iPlanet technical support for database problems.

Understanding the iPlanet Application Server Limitations

Like JDBC, the iPlanet Application Server supports the broadest spectrum of database engines and features. In some cases, the iPlanet Application Server itself or the iPlanet Application Server JDBC driver may not fully support a particular database feature, or it may report incorrect information. If you cannot access a database feature from your iPlanet Application Server application and you have eliminated the database as the problem, check this section in the documentation and the *Release Notes* to determine if the problem you encounter is a documented iPlanet Application Server limitation. If not, fully document the problem and contact iPlanet 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 iPlanet Application Server JDBC driver. Almost all feature limitations apply to JDBC 2.0.

Table 8-1 lists the JDBC features that **are not** supported, either partially or completely in the iPlanet Application Server:

Table 8-1 JDBC Feature Limitations

Feature	Limitation
`Connection.setTransactionIsolation`	Works only with isolation levels supported by your database vendors.

Table 8-1 JDBC Feature Limitations

Feature	Limitation
`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 the iPlanet Application Server. Otherwise an empty string is returned. For ODBC, Informix, and DB2, the driver returns a cursor name if none is specified.
`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.

Table 8-1 JDBC Feature Limitations

Feature	Limitation
`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

Table 8-2 lists the databases currently supported by the iPlanet Application Server.

Table 8-2 Supported Databases

Database	Notes
Oracle	Support is through the Oracle OCI interface. Both Oracle 7 and 8 database instances are supported in a fully multi-threaded environment. The iPlanet Application Server coexists with all Oracle RDBMS tools and utilities, such as SQL*Plus, Server Manager, and Oracle Backup.
Informix	Support is through Informix CLI interface. Both Informix Online Dynamic Server and Informix Universal Server are supported.
Sybase	Support is through Sybase CTLIB. Sybase server version 12 and client version 12 are supported.
Microsoft SQL Server	Support is through the Microsoft ODBC interface. Microsoft SQLServer on Windows NT only is supported.
DB2	Support is through the DB2 CLI client interface. DB2 versions 5.1 and 6.1, and client version 7.1 are supported.
ODBC	The iPlanet Application Server does not specifically support or certify any ODBC 2.0 or 3.0 compliant driver sets, though they may work.

NOTE Because the databases supported by the iPlanet Application Server are constantly being updated and database vendors are consistently upgrading their products, always check with the iPlanet technical support for the latest database support information.

Using JDBC in Server Applications

JDBC is part of the iPlanet Application Server runtime environment. This means JDBC is always available any time you use Java to program an application. In a typical multi-tiered server application, you use JDBC to access an EIS database from a client, from the presentation layer, in servlets, and in EJBs.

However, in practice it makes sense—for security and portability reasons—to restrict database accesses to the middle layers of a multi-tiered server application. In the iPlanet Application Server programming model, this means placing all JDBC calls in servlets and EJBs, with the preference being towards EJBs.

There are two reasons for this programming preference:

- Placing all JDBC calls inside EJBs makes your application more modular and more portable.
- 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 Always use a globally available datasource to create a global (`bean-wide`) connection so that the EJB transaction manager controls the transaction.

Using JDBC in EJBs

Placing 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. Because EJBs are components, use them as building blocks for many applications with little or no changes, and maintain a common interface to your EIS database.

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 transaction management programming using JDBC or `javax.transaction.UserTransaction` is appropriate for an application. In these cases, program the 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 EJB transaction attribute property to `TX_BEAN_MANAGED` in the bean's deployment descriptor.
2. Program the appropriate JDBC or transaction management statements in the bean, including statements to start the transaction, and to commit or roll it back.

Do not program explicit transaction handling in EJBs when 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

Specify or examine the transaction level for a connection using the `setTransactionIsolation()` and `getTransactionIsolation()` methods, respectively. Note that you cannot call `setTransactionIsolation()` during a transaction.

Table 8-3 defines the transaction isolation levels, 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.

Table 8-3 Transaction Isolation Levels

Transaction Isolation Level	Description
TRANSACTION_SERIALIZABLE	Dirty reads, non-repeatable reads and phantom reads are prevented.

Before specifying a bean's transaction isolation level, verify the level is supported by your database management system. Not all databases support all isolation levels. Test your database programmatically by using the method `supportsTransactionIsolationLevel()` in `java.sql.DatabaseMetaData`, as shown 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 JDBC 2.0 API specification.

Using JDBC in Servlets

Servlets are at the heart of an iPlanet Application Server application. 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 an application's work.

The iPlanet Application Server applications use JDBC embedded in EJBs for most database accesses. This is the preferred method for database accesses using the iPlanet Application Server because it enables you to take advantage of the transaction control built-in to 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 a database from EJBs. There is less call overhead, if an application is spread across servers so that EJBs are accessible only through the Java Remote Method Interface (RMI). Use direct database service through servlets sparingly. If providing database access from servlets, restrict access to very short durations, the transaction is read-only, and take advantage of the JDBC 2.0 `RowSet` class.

If access to a database is from a servlet, use the JDBC 2.0 `RowSet` interface to interact with the database. A row set 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 row sets, see "Working with RowSet," on page 187.

Handling Connections

The iPlanet Application Server implements the JDBC 2.0 compliant interface `java.sql.Connection`. The connection behavior depends on if 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 cannot propagate across processes or datasources; it is local to the current process and to the current datasource.

The transaction context on this connection type is managed using the `setAutoCommit()`, `commit()`, and `rollback()` methods.

Registering a Local Datasource

The first step to create a local connection is to register the datasource with the iPlanet Application Server. Once the datasource is registered, the datasource can be used to make connections to the listed database using `getConnection()`.

Register the datasource by creating an XML resource descriptor file that describes the datasource properties. Next, register the properties with the iPlanet Application Server using the Administration Tool or the `resreg` utility. `resreg` takes as its argument, the resource descriptor file name 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 an XML descriptor file like the following, and name it `SampleDS.xml` (use the iPlanet Application Server Deployment Tool to create an 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>
```

Then use this resource descriptor file to register the datasource with the following command:

```
resreg SampleDS.xml
```

For more information about resource descriptor files, see Chapter 10, “Deployment Packaging.” For more information about the iPlanet Application Server Administration Tool, see the *Administrator’s Guide.*

Global Connections

A `Connection` object is called a *global connection* if its transaction context is managed by the 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 for *bean managed transactions*. For more information about transactions, see Chapter 7, “Handling Transactions with EJBs.”

Transaction management methods are disabled for global connections, for example, `setAutoCommit()`, `commit()`, and `rollback()`.

Using Resource Managers

The datasource collection in which a global transaction participates is known as a resource manager. All resources managers must be registered with the iPlanet Application Server and be enabled to participate in global transactions. Resource managers are set up at install time or they are set up using the iPlanet Application Server Administration Tool (for more information, see the *Administrator's Guide*). A global connection must be associated with a resource manager.

Registering a Global Datasource

The first step in creating a global connection is to register the datasource with the iPlanet Application Server. Once the datasource is registered, the datasource is used to make connections to the listed database using `getConnection()`.

Register the datasource by creating an XML resource descriptor file that describes the datasource properties. Next, register the properties with the iPlanet Application Server using the Administration Tool or the `resreg` utility. `resreg` takes as its argument, the resource descriptor file name 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 iPlanet Application Server Deployment Tool to create the 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>
```

Use the resource descriptor file to register the datasource with the following command:

```
resreg GlobalSampleDS.xml
```

For more information about resource descriptor files, see Chapter 10, “Deployment Packaging.” For more information about the iPlanet Application Server Administration Tool, see the *Administrator’s Guide*.

Creating a Global Connection

The following program demonstrates how a datasource is looked up and a connection created from it. As illustrated, 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 the transaction context is managed by the EJB container and global transactions are disabled. With container managed transactions, the transaction context is managed implicitly by the EJB container and with bean managed transactions the transaction context is handled explicitly.

Connection object methods `setAutoCommit()`, `commit()`, and `rollback()` are disabled for this connection type.

For more information on how to enable or disable global transactions in an EJB container, see the *Administrator's Guide*.

Registering a Container Managed Local Datasource

The container managed local datasource registering process is the same as for the local and global datasources. For more information, see "Registering a Local Datasource," on page 176.

Working with JDBC Features

While this chapter is not a JDBC primer, it does introduce how to use JDBC in EJBs with the iPlanet Application Server. The following sections describe various JDBC interfaces and classes that either have special requirements in the iPlanet Application Server environment, or are new JDBC 2.0 features you are encouraged to use when developing an iPlanet Application Server application.

For example, "Working With Connections," on page 180 describes the resources the iPlanet Application Server releases when a connection is closed because this information differs among different JDBC implementations. On the other hand, "Pooling Connections," on page 181 and "Working with RowSet," on page 187 offer more extensive coverage because these are new JDBC 2.0 features that offer increased power, flexibility, and server application speed.

Working With Connections

When opening a JDBC connection, the iPlanet Application Server allocates connection resources. Call `Connection.close()` when a connection is no longer needed, to free the connection resources. Always reestablish connections before continuing database operations after you call `Connection.close()`.

Use `Connection.isClose()` 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. To 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 an application uses several connections, and if connections are frequently opened and closed, the iPlanet Application Server automatically provides connection pooling. Connection pooling provides a connection cache that automatically closes when necessary.

NOTE	Connection pooling is an automatic feature of the iPlanet Application Server; the API is not exposed.

setTransactionIsolation

Not all database vendors support all transaction isolation levels available in JDBC. The iPlanet Application Server permits specifying 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 174.

getTypeMap, setTypeMap

The iPlanet Application Server JDBC driver implementation does not support type mapping, a new SQL-3 feature that most database vendors do not support.

cancel

`cancel()` is supported for all databases that support `cancel()`.

Pooling Connections

Two costlier database operations to execute in JDBC are for creating and destroying database connections. Connection pooling permits a single connection cache for connection requests. 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.

The iPlanet Application Server automatically provides JDBC connection pooling wherever you make JDBC calls. The process of pooling database connections works differently for each connection type.

- 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 reused by transactions that execute on the thread.

- For **container managed local connections**, the `connection.close()` method does not release the connection to the connection pool immediately. When the transaction that the connection is participating in is finished, the connection is released back to the connection pool by the iPlanet Application Server.

In each Java engine, each driver (Oracle, Sybase, Informix and DB2) has its own connection pool. Each connection pool size is according to the application requirements. For more information on the connection pool settings (such as, maximum number of connections, connection timeout and so on), see the *Administrator's Guide.*

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

The iPlanet Application Server supports concurrency for `FORWARD-ONLY READ-ONLY` and for `SCROLL-INSENSITIVE READ-ONLY` result sets. On callable statements, the iPlanet Application Server also supports concurrency for `FORWARD-ONLY UPDATABLE` result sets.

`SCROLL-SENSITIVE` concurrency is not supported.

Updatable Result Set Support

In the iPlanet Application Server, 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	Use join clauses to create read-only result sets against multiple tables; however, these result sets are not updatable.

For Sybase, the select list must include a unique index column. Sybase also permits calling `execute()` or `executeQuery()` to create an updatable result set. However, the statement must 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, as follows:

```
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 result set method, `getCursorName()`, enables the determining of the cursor name used to fetch a result set. If a cursor name is not specified by the query itself, different database vendors return different information. The iPlanet Application Server attempts to handle these differences as transparently as possible. Table 8-4 indicates the cursor name returned by different database vendors if no cursor name is specified in the initial query.

Table 8-4 Cursor Name

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 the iPlanet Application Server. 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

The iPlanet Application Server implements this JDBC method and it only works with *scalar* data types. JDBC 2.0 adds additional method versions that include a map argument. The iPlanet Application Server does not implement maps and ignores map arguments.

getRef, getBlob, getClob, and getArray

References, blobs, clobs, and arrays are new SQL-3 data types. The iPlanet Application Server does not implement these data objects or their methods. However, to work with references, blobs, clobs, and arrays use `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 Use the iPlanet Application Server feature `SqlUtil.loadQuery()` to load an `iASRowSet` with a prepared statement. For more information, see the `SqlUtil` class entry in the *Foundation Class Reference (Java)*.

setObject

This method may only be used with *scalar* data types.

addBatch

This method enables ganging of 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. The iPlanet Application Server does not implement these data objects or the methods that work with them. However, to work with references, blobs, clobs, and arrays use `getBinaryStream()` and `setBinaryStream()`.

getMetaData

Not all database systems return complete metadata information. See your database 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 the iPlanet Application Server.

getRef, getBlob, getClob, getArray

References, blobs, clobs, and arrays are new SQL-3 data types. The iPlanet Application Server does not implement these data objects or the methods that work with them. However, to work with references, blobs, clobs, and arrays use `getBinaryStream()` and `setBinaryStream()`.

Handling Batch Updates

The JDBC 2.0 specification provides a batch update feature to an application to pass multiple SQL update statements (`INSERT`, `UPDATE`, `DELETE`) in a single database request. This statement ganging can result in a significant performance increase when a large number of update statements are pending.

The `Statement` class includes two new methods for executing batch updates:

- `addBatch()` permits adding a SQL update statement (`INSERT`, `UPDATE`, `DELETE`) to a group of statements prior to execution. Only update statements that return a simple update count can be grouped using this method.
- `executeBatch()` permits execution of a collection of SQL update statements as a single database request.

In order to use batch updates, an application must disable auto commit options, as follows:

```
...
// 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();
```

Call `clearBatch()` to remove all ganged statements from a batch operation before `executeBatch()` is called (for example, because an error is detected).

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 cannot be grouped for a batch operation. Also, some databases do not allow data definition 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-in to the iPlanet Application Server EJB container. If an EJB does not specify the `TX_BEAN_MANAGED` transaction attribute, automatic support for distributed transactions in an application is enabled.

In servlets and 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 the iPlanet Application Server as global datasources. In other words, their datasource properties files must include a `ResourceMgr` entry whose value must be configured at install time.

```
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 obtain 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 RowSet

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`, a program must import `javax.sql`, and implement the `RowSet` interface. `RowSet` extends the `java.sql.ResultSet` interface, permitting it to act as a JavaBean component.

Because a RowSet is a JavaBean, you can implement RowSet events and set properties on the RowSet. Furthermore, because RowSet is a ResultSet extension, you can iterate through a RowSet just as you would iterate through a ResultSet.

To fill a RowSet call the RowSet.execute() method. The execute() method uses property values to determine the datasource and retrieve data. The actual properties to set and examine depends upon the implementation of RowSet invoked.

For more information about the RowSet interface, see the JDBC 2.0 Standard Extension API Specification.

Using iASRowSet

The iPlanet Application Server provides a RowSet class called iASRowSet for convenience. iASRowSet extends ResultSet, therefore call methods are inherited from the ResultSet object. iASRowSet overrides the getMetaData() and close() methods of ResultSet. Because iASRowSet is not a driver-level class, it is easier to use than ResultSet.

The RowSet interface is fully supported except as noted in Table 8-5.

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.
setTypeMap()	java.util.Map	None	Map is a JDBC 2.0 feature that is not currently supported.

RowSetReader

`iASRowSet` provides a full `RowSetReader` class implementation.

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 `ResultSet` instead of a single row.

Using CachedRowSet

The JDBC specification provides a `RowSet` class called `CachedRowSet`. `CachedRowSet` permits data retrieval from a datasource, then detaches from the datasource while examining, and modifying the data. A cached row set keeps track of the original data retrieved and any data changes made by an application. If the application attempts to update the original datasource, the row set is reconnected to the datasource, and only those rows that have changed are merged back into the database.

Creating a RowSet

To create a row set in an the iPlanet Application Server application:

```
iASRowSet rs = new iASRowSet();
```

Using JNDI

All JDBC driver managers, such as the JDBC driver manager implemented in the iPlanet Application Server, must find and access a JDBC driver by looking up the driver and a JDBC URL for connecting to the database. However, 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.

JDBC 2.0 specifies using JNDI to provide a uniform, platform and JDBC vendor independent way for an application to find and access remote services over the network. In place of this hard-coded information, JNDI permits assigning a logical name to a particular datasource. Once the logical name is established, you need only modify it a single time to change the deployment and application location.

JDBC 2.0 specifies that all JDBC datasources are 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, just change its entry in the JNDI namespace without having to modify the application.

For more information about JNDI, see the JDBC 2.0 Standard Extension API.

Chapter 9

Developing and Deploying RMI/IIOP-Based Clients

This chapter explains how to access to EJBs via the RMI over IIOP (RMI/IIOP) protocol within an iPlanet Application Server environment.

This chapter contains the following sections:

- Overview of RMI/IIOP Support
- Developing RMI/IIOP Client Applications
- Packaging RMI/IIOP Client Applications
- Configuring RMI/IIOP Support
- Deploying Client Applications
- Running Client Applications
- Troubleshooting
- Performance Tuning RMI/IIOP
- Firewall Configuration for RMI/IIOP
- Viewing RMI/IIOP Log Messages
- Sample Applications

Overview of RMI/IIOP Support

iPlanet Application Server supports access to EJBs via the RMI/IIOP protocol as specified in the Enterprise JavaBeans Specification, V1.1, and the Enterprise JavaBeans to CORBA Mapping specification. These clients use JNDI to locate EJBs and use Java RMI/IIOP to access business methods of remote EJBs.

The following topics are covered in this overview:

- Scenarios
- Architectural Overview
- iPlanet Value-Added Features
- Limitations

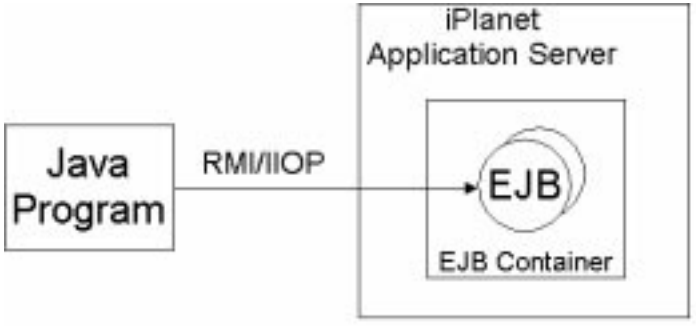
Scenarios

The most common scenarios in which RMI/IIOP clients are employed are when either a stand-alone Java main program or another application server acts as a client to EJBs deployed to iPlanet Application Server.

Stand-Alone Java Main Program

In the simplest case, a stand-alone Java man program running on a variety of operating systems uses RMI/IIOP to access business logic housed in back-end EJB components, as shown in Figure 9-1.

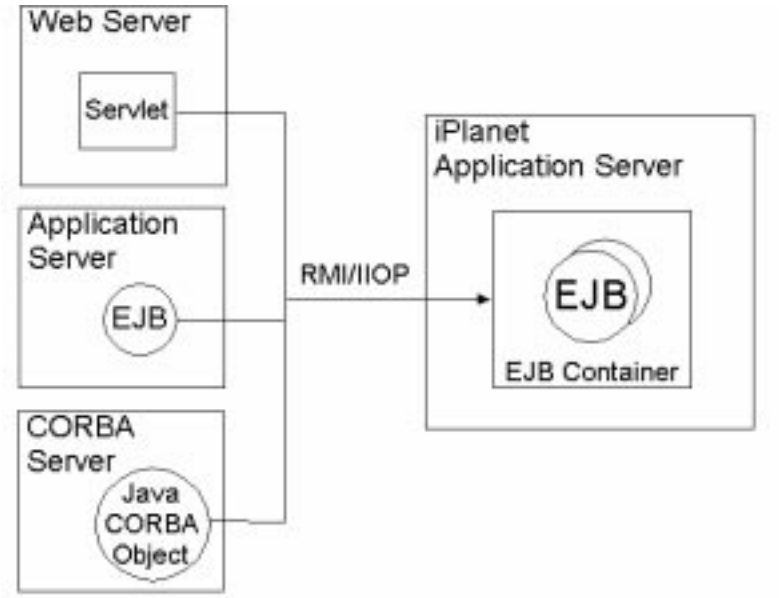
Figure 9-1 Stand-alone Java main program



Server-to-Server

Java-enabled web servers, Java-based CORBA objects, and even other application servers can use RMI/IIOP to access EJBs housed in an iPlanet Application Server, as shown in Figure 9-2.

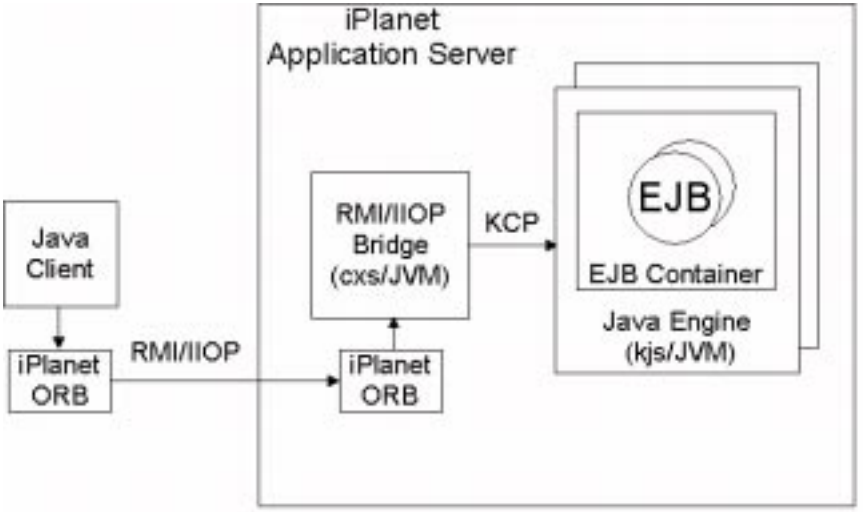
Figure 9-2 Server-to-server



Architectural Overview

RMI/IIOP support in iPlanet Application Server involves a specialized Java Engine process named the CORBA Executive Server (CXS). The CXS acts as a bridge between Java clients using RMI/IIOP and EJBs deployed to one or more Java Engines acting as EJB containers. For every EJB accessed by RMI/IIOP clients, the Bridge process handles the incoming IIOP-based requests and maps these requests to internal calls to EJBs housed within the EJB containers, as shown in Figure 9-3.

Figure 9-3 Architecture



Regardless of the Java 2 environment used on the client side, in this release of iPlanet Application Server, you are required to use the ORB that is bundled as part of the Application Server.

iPlanet Value-Added Features

iPlanet's implementation of RMI/IIOP goes beyond the RMI/IIOP specification by providing the following value-added features:

- Naming Services
- Built-in ORB
- Basic Authentication and EJB Container Integration
- Server-Side Load Balancing
- Scalability
- High Availability
- Minimal Ports Opened in Firewalls

Naming Services

The RMI/IIOP clients use the standard CORBA COS NameService to resolve EJBHome objects. As EJBs are deployed to iPlanet Application Server, they are automatically and dynamically registered in the naming service.

Built-in ORB

iPlanet provides a built-in ORB to support RMI/IIOP access to EJBs. You do not have to install and configure a third party ORB to use RMI/IIOP with iPlanet Application Server.

Basic Authentication and EJB Container Integration

Although the RMI/IIOP standards do not yet define a means of performing basic authentication between an RMI/IIOP client and an EJB server, iPlanet provides such support in the Application Server. This feature enables the EJB deployer to control access to EJBs using standard declarative and programmatic controls that apply to both web and RMI/IIOP clients.

As an RMI/IIOP client authenticates to the iPlanet Application Server, the principal information is automatically propagated to the EJB container for authorization based on the standard EJB security mechanisms. To trigger collection of the client's user name and password, iPlanet provides a client-side callback mechanism that enables an application to obtain a user name and password through application-specific means. Once the user name and password information is collected by the iPlanet RMI/IIOP infrastructure, this information is propagated over IIOP to the Application Server.

Server-Side Load Balancing

As new RMI/IIOP requests arrive at an instance of iPlanet Application Server, the application server load balances these requests against one or more JVMs acting as EJB containers. Load balancing is implemented in a simple round-robin scheme. Upon startup, the application server obtains a list of the available EJB container processes, also known as Java Engines. As home lookup requests arrive from RMI/IIOP clients, the Application Server uses a list of engines to select the target engine on which an EJB home is hosted. Subsequent lookups for that EJB home, bean creations on that home, and business method invocations on the created beans go to the same target engine.

Scalability

Multiple RMI/IIOP processes can be configured for each application server instance. This feature enables system administrators to configure any number of JVMs dedicated to handling incoming RMI/IIOP requests. Client applications can rotor through a list of the available RMI/IIOP processes or use round-robin DNS to implement basic, client-side load balancing. Administrators can also modify the number of processing threads available for each RMI/IIOP and EJB container process to suit the expected loads of the system.

High Availability

The following features contribute to high availability:

- **Auto Restart of Java Engines:** The application server monitors both the Bridge processes as well as the Java Engines supporting the EJB containers. If a process fails, administrative services automatically restart the failed process.
- **Stateful Session Bean Failover:** RMI/IIOP clients can take advantage of the built-in EJB stateful session bean replication feature of iPlanet Application Server. If a Java Engine housing an EJB container fails, then subsequent requests to the stateful session bean continue to be processed once the Java Engine restarts.
- **EJB Handle and Object Reference Failover:** If a Bridge process fails and is automatically restarted, the RMI/IIOP clients can continue to access EJBs without interruption.

Minimal Ports Opened in Firewalls

The RMI/IIOP Bridge processes both name service and business method calls on a common, fixed IP port number. This approach helps to minimize the number of ports opened in firewalls positioned between RMI/IIOP clients and iPlanet Application Server instances on which Bridge processes are configured.

Limitations

RMI/IIOP in iPlanet Application Server has the following limitations:

- It does not support C++ clients.
- It applies only to accessing EJBs.
- General RMI objects cannot be accessed via RMI/IIOP.
- Transaction propagation from Java RMI/IIOP clients is not supported.

Developing RMI/IIOP Client Applications

Developing RMI/IIOP-based client applications to work with iPlanet Application Server is very similar to developing clients for other J2EE-certified application servers. With minimal, if any, changes to the JNDI lookup section of your client, you can reuse your Java client to work with a variety of J2EE application servers.

The following topics are covered in this section:

- JNDI Lookup for the EJB Home Interface
- Client Authentication
- Client-Side Load Balancing and Failover

JNDI Lookup for the EJB Home Interface

One of the first steps in coding an RMI/IIOP client is to perform a lookup of an EJB's home interface. In preparation for performing a JNDI lookup of the home interface, you must first set several environment properties for the `InitialContext`. Then you provide a lookup name for the EJB.

The steps and an example are summarized in the following sections.

- Specifying the Naming Factory Class
- Specifying the Target RMI/IIOP Bridge
- Specifying the JNDI Name of an EJB
- A JNDI Example

Specifying the Naming Factory Class

According to the RMI/IIOP specification, the client must specify `com.sun.jndi.cosnaming.CNCtxFactory` as the value of the `java.naming.factory.initial` entry in an instance of a `Properties` object. This object is then passed to the JNDI `InitialContext` constructor prior to looking up an EJB's home interface. For example:

```
...
Properties env = new Properties();
env.put("java.naming.factory.initial", "com.sun.jndi.cosnaming.CNCtxFactory");
env.put("java.naming.provider.url", "iiop://" + host + ":"+port);
Context initial = new InitialContext(env);
Object objref = initial.lookup("java:comp/env/ejb/MyConverter");
...
```

Specifying the Target RMI/IIOP Bridge

According to the RMI/IIOP specification, your client must set the `java.naming.provider.url` property to a value of the following form:

`iiop://`*server*`:`*port*

The *server* identifies the host on which an iPlanet Application Server instance resides. The *port* identifies a specific RMI/IIOP Bridge process running on the application server host.

Along with the `java.naming.factory.initial` property, you can specify the `java.naming.provider.url property` either on the command line or in the client application's code.

The following is an example of setting the IIOP URL on the Java command line (this command must be all on one line):

```
java -Djava.naming.provider.url="iiop://127.0.0.1 :9010"
-Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
j2eeguide.cart.CartClient
```

In this case, the client application does not need to instantiate a `Properties` object:

```
...
public static void main(String[] args) {
   Context initial = new InitialContext();
   Object objref = initial.lookup("java:comp/env/ejb/MyConverter");
   ...
}
```

As an alternative, you can set the IIOP URL within the client application. In the following example, two command line arguments are passed into the main classes of the client.

```
...
public static void main(String[] args) {
   String host = args[0];
   String port = args[1];
   Properties env = new Properties();

   env.put("java.naming.factory.initial",
      "com.sun.jndi.cosnaming.CNCtxFactory");

   env.put("java.naming.provider.url", "iiop://" + host + ":"+port);

   Context initial = new InitialContext(env);
   Object objref = initial.lookup("java:comp/env/ejb/MyConverter");
...
}
```

Specifying the JNDI Name of an EJB

After creating a new JNDI `InitialContext` object, your client calls the `lookup` method on the `InitialContext` to locate the EJB's home interface. The name of the EJB is provided on the call to `lookup`. When using RMI/IIOP to access remote EJBs, the parameter is referred to as the "JNDI name" of the EJB. Depending on how your client application is packaged, the supported values of the JNDI name vary.

The JNDI Name Without an Application Client Container

When the client is not packaged as part of an Application Client Container (ACC), you must specify the absolute name of the EJB in the JNDI lookup. iPlanet supports the following approaches to performing the JNDI lookup outside of an ACC:

`initial.lookup("ejb/`*ejb-name*`");`

`initial.lookup("ejb/`*module-name/ejb-name*`");`

The *ejb-name* is the name of the EJB as it appears in the `<ejb-name>` element of the EJB's deployment descriptor. For example, here is a lookup using the value MyConverter:

```
initial.lookup("ejb/MyConverter");
```

This lookup requires that the EJB deployment descriptor specify `MyConverter` as the `<ejb-name>`, as follows:

```
<ejb-jar>
   <enterprise-beans>
      <session>
         <ejb-name>MyConverter</ejb-name>
         <home>j2eeguide.converter.ConverterHome</home>
         <remote>j2eeguide.converter.Converter</remote>
         ...
      </session>
   </enterprise-beans>
</ejb-jar>
```

Using only the ejb name in the JNDI lookup on the RMI/IIOP client works properly as long as only one EJB of this name is registered in the Application Server. If you have more than one EJB of this name registered, you must qualify the ejb name with the name of the EJB JAR module in which the EJB of interest exists. You can do this by including the module name in front of the ejb name in the JNDI lookup. The EJB JAR module name is the name of the EJB JAR file minus the `.jar` extension.

In the Converter sample application, since the EJB JAR module name is `j2eeguide-converterEjb` (based on the EJB JAR file name of `j2eeguide-converterEjb.jar`), a lookup based on the module name looks like this:

```
initial.lookup("ejb/j2eeguide-converterEjb/MyConverter");
```

The safe approach is to always use the module name qualifier when performing JNDI lookups from RMI/IIOP clients that do not use Application Client Container packaging. The only drawback of the module name approach is that the client becomes aware of additional aspects of the deployment structure of the server side environment beyond the absolute EJB name.

As of Service Pack 3, you can also use the prefix `java:comp/env/ejb/` when performing lookups via absolute references. For example, the lookup in the Converter sample could be written as follows:

```
initial.lookup("java:comp/env/ejb/MyConverter");
```

Or, with a module name, it could be written as follows:

```
initial.lookup("java:comp/env/ejb/j2eeguide-converterEjb/MyConverter");
```

There is no mechanical difference between supplying this prefix and the first two approaches. You might find the `java:comp/env/ejb/` confusing when used in conjunction with absolute EJB references because this notation is typically used when you are using indirect EJB references.

The JNDI Name When Using an Application Client Container

If you are using an Application Client Container (ACC) to house the client, the JNDI name can use the logical name of the EJB as specified in the `<ejb-ref-name>` element in the ACC deployment descriptor. This approach to specifying the JNDI name of an EJB, although dependent on packaging and running the client in the context of an ACC, is the same approach as used within a servlet or EJB housed within the Application Server.

As is the case for servlets and EJBs that perform lookups on EJBs, the format of the lookup must be as follows:

```
initial.lookup("java:comp/env/ejb/ejb-ref-name");
```

The *ejb-ref-name* is the value specified in the `<ejb-ref-name>` element of the ACC deployment descriptor.

In the following example, since `SimpleConverter` appears in the `<ejb-ref-name>` element of the ACC deployment descriptor, a value of `SimpleConverter` is used in the JNDI lookup:

```
initial.lookup("java:comp/env/ejb/SimpleConverter");
```

The `application-client.xml` file looks like this:

```
<application-client>
   <display-name>converter-acc</display-name>
   <description>
      Currency Converter Application Client Container Sample
   </description>
   <ejb-ref>
      <ejb-ref-name>SimpleConverter</ejb-ref-name>
      <ejb-ref-type>Session</ejb-ref-type>
      <home>j2eeguide.converter.ConverterHome</home>
      <remote>j2eeguide.converter.Converter</remote>
      <ejb-link>Test</ejb-link>
   </ejb-ref>
</application-client>
```

A benefit of using ACC packaging is that the JNDI names specified in the client application are indirectly mapped to the absolute JNDI names of the EJBs. However, this aspect is about the only real benefit of using ACC. See "Using Application Client Container (ACC)," on page 206 for more details.

A JNDI Example

The following client program is taken from the Currency Converter application that is part of the *J2EE Developer's Guide* examples bundled in iPlanet Application Server. See "Sample Applications," on page 227 for more information on the RMI/IIOP samples included with the application server.

```
package j2eeguide.converter;

import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

import j2eeguide.converter.Converter;
import j2eeguide.converter.ConverterHome;

public class ConverterClient {

public static void main(String[] args) {
   try {

      if (args.length != 2) {
         System.out.println("Wrong number of arguments to client");
         System.exit(1);
      }
      String host = args[0];
      String port = args[1];
      Properties env = new Properties();
```

```
            env.put("java.naming.factory.initial",
                "com.sun.jndi.cosnaming.CNCtxFactory");
            env.put("java.naming.provider.url", "iiop://" + host
                + ":"+port);

            Context initial = new InitialContext(env);
            Object objref = initial.lookup("ejb/MyConverter");

    // Alternatively, the module name could be used as a qualifier:.
    // Object objref =
    // initial.lookup("ejb/j2eeguide-converterEjb/MyConverter");

            ConverterHome home
                =(ConverterHome)PortableRemoteObject.narrow(objref,
                ConverterHome.class);

            Converter currencyConverter = home.create();

            double amount = currencyConverter.dollarToYen(100.00);

            System.out.println(String.valueOf(amount));

            amount = currencyConverter.yenToEuro(100.00);

            System.out.println(String.valueOf(amount));

        }

        catch (Exception ex) {
            System.err.println("Caught an unexpected exception!");
            ex.printStackTrace();
        }
    }
    }
```

Client Authentication

To take advantage of the optional authentication mechanism for RMI/IIOP clients, you must provide a security principal class that implements the `com.netscape.ejb.client.IUserPrincipal` interface. This class is instantiated once by the client side iPlanet RMI/IIOP infrastructure as the JNDI `lookup` method is called. The client side RMI/IIOP infrastructure calls the `setPrincipal` method of this interface before the JNDI lookup triggers a call to the remote name services.

The security principal class must be named in the client's properties and the class must be present in the client's CLASSPATH to enable the RMI/IIOP infrastructure to load the class during execution of the client.

For example, in the Converter sample application, you could add a third property specifying the security principal class to be instantiated as the JNDI lookup is performed:

```
...
Properties env = new Properties();
env.put("java.naming.factory.initial",
    "com.sun.jndi.cosnaming.CNCtxFactory");
env.put("java.naming.provider.url", "iiop://" + host + ":"+port);
env.put("com.netscape.ejb.client.PrincipalClass",
    "j2eeguide.converter.RmiPrincipal");

Context initial = new InitialContext(env);
Object objref = initial.lookup("ejb/MyConverter");
...
```

The `RmiPrincipal` class is the class that you develop that implements the `com.netscape.ejb.client.IUserPrincipal` interface.

Sample Principal Class

The `IUserPrincipal` interface can be implemented in several ways. The simplest is to pop up a dialog in the setPrincipal 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 are used to set the security context propagated by the client.

The RMI/IIOP Bridge attempts to authenticate the user and password with the iPlanet Application Server security manager. If an authentication exception occurs in Bridge the client side ORB is notified and the `setPrincipal` method is called to obtain the correct user/password information. The client side RMI/IIOP infrastructure retries the request automatically three times, after which an authentication exception is generated on the client side.

```
...
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;
    }
}
```

Another valid implementation of `IUserPrincipal` supports multiple user identities in the same client JVM. This is done by using `ThreadLocal` variables to store the user name and password. In this case, the methods in the `IUserPrincipal` implementation must to be `ThreadLocal` aware.

Client-Side Load Balancing and Failover

Although iPlanet Application Server provides server-side load balancing and failover for RMI/IIOP access, you may consider implementing client side approaches to further enhance the performance and availability of your application.

Manual Selection from the List of Known Bridges

You can create a wrapper class to round-robin through a set of known bridge host name and port combinations on behalf of the client business application. If a communication exception occurs for one of the host name and port combinations, the wrapper class attempts to use the next host name and port combination in the list.

For example, the following exception is thrown by the underlying client classes when the remote RMI/IIOP Bridge cannot be contacted:

```
javax.naming.CommunicationException: Cannot connect to ORB.  Root
exception is org.omg.CORBA.COMM_FAILURE:
```

Your client wrapper code can catch this exception and select the next available *host_name*:*port* pairing to re-attempt access to the EJB.

Round Robin DNS

To implement a simple load balancing scheme without making source code changes to your client, you can leverage the round-robin feature of DNS. In this approach, you define a single virtual host name representing multiple physical IP addresses on which RMI/IIOP Bridge processes are listening. Assuming that you configure all of the RMI/IIOP Bridge process to listen on a common IIOP port number, the client application can use a single *host_name*:*IIOP_port* during the JNDI lookup. The DNS server resolves the host name to a different IP address each time the client is executed.

After developing the client application, you must package your application in preparation for deployment.

Packaging RMI/IIOP Client Applications

You can package RMI/IIOP Client Applications in these ways:

- Using the Assembly Tool GUI
- Automating Reassembly Using Ant
- Using Application Client Container (ACC)

Using the Assembly Tool GUI

The iPlanet Application Server Deployment Tool automatically generates a JAR file containing EJB-specific home and remote interface and stub classes when you indicate that an EJB is accessible via IIOP. As an alternative to copying individual class files to the client, this JAR file can be deployed as part of the client application.

The Deployment Tool does not support packaging of applications to be deployed as part of an Application Client Container.

Automating Reassembly Using Ant

If you have an interest in a command line means of packaging RMI/IIOP client applications, it is recommended that you review the Ant-based `build.xml` files supplied as part of the sample applications. The `build.xml` files for RMI/IIOP-based samples contain an `install_client` target that can be easily enhanced to assemble a self-contained client JAR file in much the same manner as the Deployment Tool creates a JAR file of client-oriented classes.

Using Application Client Container (ACC)

Although iPlanet does not recommend deployment of client applications in Application Client Containers, this deployment and runtime method is supported as part of the J2EE specification. This approach is not recommended because, in the current state of the ACC specification, using ACC introduces additional complexity with minimal benefit. Furthermore, due to the limited definition of ACC within the J2EE v1.2, support for ACC varies widely across J2EE application servers.

If you choose to experiment with ACC on iPlanet Application Server, take the following deployment steps into consideration:

- The `iasacc.jar` JAR file supplied as part of iPlanet Application Server must be in the client's CLASSPATH. This file can be copied from the following location to the client environment:

  install_dir`/ias/classes/java/iasacc.jar`

  Including this file eliminates the need to include the `iasclient.jar` file in the client's environment.

- A J2EE v1.2-compliant EAR file needs to be created. This EAR file must contain:

  - The RMI/IIOP client application classes, home and remote interfaces and stubs.
  - A J2EE v1.2 XML descriptor file named application-client.xml. For example:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application-client PUBLIC '-//Sun Microsystems,
Inc.//DTD J2EE Application Client 1.2//EN'
'http://java.sun.com/j2ee/dtds/application-client_1_2.dtd'>

<application-client>
   <display-name>converter-acc</display-name>
   <description>
   Currency Converter Application Client Container Sample
   </description>
   <ejb-ref>
          <ejb-ref-name>SimpleConverter</ejb-ref-name>
          <ejb-ref-type>Session</ejb-ref-type>
          <home>j2eeguide.converter.ConverterHome</home>
```

```
                        <remote>j2eeguide.converter.Converter</remote>
                        <ejb-link>Test</ejb-link>
                    </ejb-ref>
                </application-client>
```

- An iPlanet Application Server specific XML descriptor file (typically named `ias-application-client.xml`). This descriptor maps EJB references to absolute EJB names.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ias-java-client-jar PUBLIC '-//Sun Microsystems, Inc.//DTD iAS
Enterprise JavaBeans 1.0//EN'
'http://developer.iplanet.com/appserver/dtds/IASjava_client_jar_1_0.dtd'>
<ias-java-client-jar>
        <ejb-ref>
                <ejb-ref-name>SimpleConverter</ejb-ref-name>
                <jndi-name>ejb/MyConverter</jndi-name>
        </ejb-ref>
</ias-java-client-jar>
```

- To invoke the client through the Application Client Container, use the following command:

`java com.netscape.ejb.client.AppContainer` *client_ear_file* `-iasXml` *ias_xml_file*

Configuring RMI/IIOP Support

To enable RMI/IIOP access to EJBs deployed to iPlanet Application Server, you must configure both the Application Server and client environments, as described in these sections:

- Server Configuration
- Client Configuration

The following configuration steps are required only once; they do not need to be repeated as you deploy EJBs and client applications.

Server Configuration

If your installation of iPlanet Application Server does not already have the RMI/IIOP Bridge process configured, you must start the iPlanet Application Server Administrative tool to add an RMI/IIOP bridge process to the application server environment.

1. Start the iPlanet Application Server Administration Tool

   On UNIX:

   install_dir`/ias/bin/ksvradmin`

   On Windows NT:

   Start->Programs->iPlanet Application Server->iAS Administration Tool

2. Connect to your application server instance and double click on the server name icon to see a list of the processes defined for this instance of the Application Server. You should see at least one `kjs` and possibly a single `kxs` process (the `kxs` process is not required for RMI/IIOP access to EJBs). If you see a `cxs` process, you already have an RMI/IIOP Bridge process defined in your application server instance. In this case, double click the `cxs` process entry, note the IIOP port number, and continue to the next section. If you don't see a Bridge process, continue to the next step to define one.

3. Select any of the existing process entries and then select File->New->Process.

4. Select `cxs` from the pull-down list of process types and enter a port number (for example, port 10822) that does not conflict with the other port numbers already in use by the `kjs` and `kxs` processes. Take the default IIOP port number (9010) as long as it does not conflict with other port assignments in your system environment. Click on OK to instantiate the process.

5. After several seconds, you see the RMI/IIOP Bridge process running in the Application Server environment. This process, along with all of the other application server processes listed in the Administrative Tool, is automatically started as the application server is restarted.

6. On UNIX, you can also check for the existence of the IIOP bridge process from the command line. For example (each command is all on one line):

```
ps -ef | grep iiop
root 1153 1 0 17:00:15 ? 0:00 /bin/sh /usr/iPlanet/ias6/ias/bin/kjs -cset CCS0
-eng 3 -iiop -DORBinsPort=9010
```

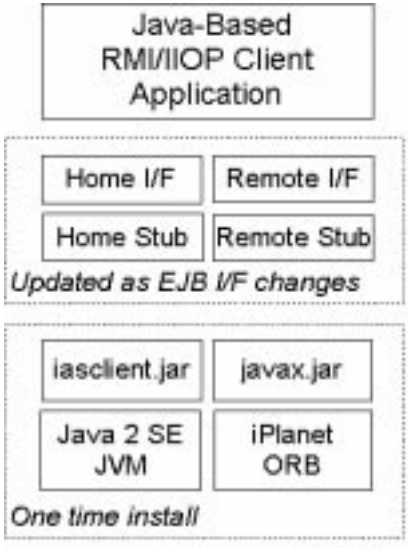
This output shows an iPlanet Java Engine process started with the `-iiop` option. This option informs this instance of the Java Engine to start itself as an RMI/IIOP Bridge process rather than a J2EE web and EJB container process.

Instantiating a `cxs` process completes the server side configuration for RMI/IIOP support.

Client Configuration

To enable a Java application client to access EJBs housed in iPlanet, you must ensure that a suitable Java 2 environment, the iPlanet ORB, and several JAR files are available on the client system, as in Figure 9-4.

Figure 9-4 Client configuration



The steps are explained in the following sections:

- Configuring a Java 2 Environment and iPlanet ORB
- Installing RMI/IIOP Client Support Classes

Configuring a Java 2 Environment and iPlanet ORB

A Java 2 environment and the iPlanet ORB must be present on the client to support communication to remote EJBs via RMI/IIOP. Either the Java 2environment bundled as part of the iPlanet Application Server or one of the tested variants described in the section "Using an Existing JDK," on page 211 must be used on the client. Other Java 2 environments are likely to work properly, but these environments are not supported by iPlanet.

Using the Bundled JDK

Because it is the platform on which iPlanet performs the bulk of its RMI/IIOP testing, the recommended Java 2 platform for client side RMI/IIOP-based applications is the Java 2 environment that is bundled as part of the application server. To use this JVM on the client side, you can simply copy the Java 2 environment from an iPlanet installation to your client environment and set the PATH appropriately to pick up the appropriate java executable file. Since the bundled Java 2 environment includes the iPlanet ORB, you do not need to modify the Java 2 environment after you copy it to the client side.

The bundled Java 2 platform is in the following location on your Application Server installation:

install_dir`/ias/usr/java/`

To copy the server's JVM environment to your client, follow these steps:

1. Navigate to *install_dir*`/ias/usr/`.
2. Copy the entire `java/` directory to your client environment. You can zip or tar the `java/` directory, transfer the archive to the client system, and expand it into a directory of your choice.
3. Set your client's PATH to include *client_side_JVM_directory*`/java/bin`.
4. Execute `java -fullversion` to ensure that the appropriate JDK (1.2.2) is being used. On UNIX, execute `which java` to check your work.

Now that you've installed the bundled JDK along with the iPlanet ORB, you need to install several supporting JAR files in your client environment. Proceed to "Installing RMI/IIOP Client Support Classes," on page 215 to install these JAR files.

Using an Existing JDK

Basic testing of several distributions of the Java 2 environment have demonstrated that, with minor setup steps, you can leverage an existing Java 2 environment in support of RMI/IIOP clients accessing EJBs housed in iPlanet Application Server. In these cases, you must copy the iPlanet ORB files from an iPlanet Application Server environment to the pre-existing JVM on the client system.

The following combinations of operating systems and Java 2 platforms have been tested with iPlanet Application Server:

- Solaris and Java 2 1.2
- Solaris or Linux and Java 1.3
- Windows 98 or Windows and Java 2 1.2
- Windows 98 or Windows and Java 2 1.3

Other combinations of operating systems and Java 2 platforms are likely to work properly with RMI/IIOP and iPlanet Application Server, but no testing has been performed on other combinations. Regardless of the combination chosen, you should ensure that you test your configuration thoroughly prior to making a determination that it is suitable for production use.

Solaris and Java 2 1.2

In this scenario, you have already installed a Java 2 1.2 environment on a Solaris system and you plan to use this JVM as the platform for your RMI/IIOP client.

In the following instructions, `JAVA_HOME` is used as the directory in which the JDK 1.2 distribution has been installed. For example:

```
export JAVA_HOME=/usr/java1.2
```

1. Copy the Java extensions directory from an iPlanet Application Server Solaris installation to your Solaris client system.

Copy the following directory:

install_dir`/ias/usr/java/jre/lib/ext`

to your Solaris client's JDK installation:

`$JAVA_HOME/jre/lib/ext`

Ensure that the `sparc/` directory containing shared object files is copied as part of this step. The iPlanet ORB, native serialization files and other support files are copied to your client in this step.

2. Copy the `orb.properties` file from your iPlanet installation:

   install_dir`/ias/usr/java/jre/lib/orb.properties`

   to your client's JDK installation:

   `$JAVA_HOME/jre/lib/`

3. Set the PATH to make sure the DLLs can be accessed by the client application:

   ```
   export PATH=$JAVA_HOME/bin:$JAVA_HOME/jre/lib/ext/i386:$PATH
   ```

Now that you've configured the existing JDK to use the iPlanet ORB, you need to install several supporting JAR files in your client environment. Proceed to "Installing RMI/IIOP Client Support Classes," on page 215 to install these JAR files.

Solaris or Linux and Java 1.3

In this scenario, you have already installed a Java 2 1.3 environment on either a Solaris or Linux system and you plan to use this JVM as the platform for your RMI/IIOP client. The following approach was tested with both Solaris and RedHat 6.2.

1. Create a directory on the client to hold the iPlanet ORB. For example:

   ```
   mkdir -p /opt/iplanet/orb
   ```

2. Copy the following JAR files from the iPlanet Application Server installation to your Linux system to an appropriate directory on the client system, for example, to `/opt/iplanet/orb/`.

   install_dir`/ias/usr/java/jre/lib/ext/rmiorb.jar`

   install_dir`/ias/usr/java/jre/lib/ext/iioprt.jar`

3. Set the environment. For example:

   ```
   JAVA_HOME=/opt/jdk1.3
   PATH=:$JAVA_HOME/bin:$JAVA_HOME/jre/lib/i386:$PATH
   CLASSPATH=/opt/iplanet/orb/iioprt.jar:/opt/iplanet/orb/rmiorb.jar
   LD_LIBRARY_PATH=$JAVA_HOME/jre/lib:$JAVA_HOME/jre/lib/i386
   export JAVA_HOME PATH CLASSPATH LD_LIBRARY_PATH
   ```

4. When you execute the client application, you must specify the ORB classes associated with the iPlanet ORB. If you do not specify the iPlanet ORB classes, the ORB classes bundled in Java 2 1.3 are used. Since the 1.3 ORB classes are incompatible with the iPlanet ORB, errors result when the 1.3 ORB classes are not overridden.

   You can specify the iPlanet ORB classes as properties on the command line (this command must be all on one line):

```
java -Dorg.omg.CORBA.ORBClass=com.netscape.ejb.client.ClientORB
-Dorg.omg.CORBA.ORBSingletonClass=com.sun.corba.ee.internal.corba.ORBSingleton
j2eeguide.converter.ConverterClient ias_host 9010
```

5. When you execute your client application, you may encounter the following error:

   ```
   ERROR! The shared library ioser12 could not be found.
   ```

   Although this error may occur when using Java 2 1.3 on the client side, RMI/IIOP access to EJBs has been demonstrated successfully using Solaris and RedHat Linux clients based on these setup instructions.

Now that you've configured the existing JDK to use the iPlanet ORB, you need to install several supporting JAR files in your client environment. Proceed to "Installing RMI/IIOP Client Support Classes," on page 215 to install these JAR files.

Windows 98 or Windows and Java 2 1.2

In this scenario, you have already installed a Java 2 1.2 environment on either Windows 98 or Windows, and you plan to use this JVM as the platform for your RMI/IIOP client.

In the following instructions, `JAVA_HOME` is used as the directory in which the JDK 1.2 distribution has been installed. For example:

```
set JAVA_HOME=JDK1.2.2
```

1. Copy the Java extensions directory from an iPlanet Application Server Windows installation to your Windows 98 or Windows client system.

   Copy the directory:

   install_dir`\ias\usr\java\jre\lib\ext`

   to your client's JDK installation:

   `%JAVA_HOME%\jre\lib\ext`

   Ensure that the `i386\` directory containing DLLs is copied as part of this step. The iPlanet ORB native serialization DLL files and other support files are copied to your client in this step.

2. Copy the `orb.properties` file from your iPlanet installation:

   install_dir`\ias\usr\java\jre\lib\orb.properties`

   to your client's JDK installation:

   `%JAVA_HOME%\jre\lib\`

3. Set the PATH to make sure the DLLs can be accessed by the client application:

   ```
   set PATH=%JAVA_HOME%\bin;%JAVA_HOME%\jre\lib\ext\i386;%PATH%
   ```

Now that you've configured the existing JDK to use the iPlanet ORB, you need to install several supporting JAR files in your client environment. Proceed to "Installing RMI/IIOP Client Support Classes," on page 215 to install these JAR files.

Windows 98 or Windows and Java 2 1.3

In this scenario, you have already installed a Java 2 1.3 environment on either Windows 98 or Windows and you plan to use this JVM as the platform for your RMI/IIOP client.

1. Create a directory on the client to hold the iPlanet ORB. For example:

   `c:\iplanet\orb\`

2. Copy the following JAR files from the iPlanet Application Server installation to the ORB directory on your Windows system to an appropriate directory on the client system, for example, to `c:\iplanet\orb\`.

   install_dir`\ias\usr\java\jre\lib\ext\rmiorb.jar`

   install_dir`\ias\usr\java\jre\lib\ext\iioprt.jar`

3. From within your Windows client, copy the native serialization DLL from the JDK installation area to another directory. For example, copy:

   ```
   %JAVA_HOME%\jre\bin\ioser12.dll
   ```

   to:

   ```
   c:\iplanet\orb\
   ```

4. Before running the client application, ensure that the PATH and CLASSPATH settings include the iPlanet ORB JAR files:

```
set JAVA_HOME=c:\jdk1.3

set PATH=%JAVA_HOME%\bin;%PATH%

set CLASSPATH=c:\iplanet\orb\iioprt.jar;c:\iplanet\orb\rmiorb.jar;%CLASSPATH%
```

5. When you execute the client application, you must specify the ORB classes associated with the iPlanet ORB. If you do not specify the iPlanet ORB classes, the ORB classes bundled in Java 2 1.3 are used. Since the 1.3 ORB classes are incompatible with the iPlanet ORB, errors result when the 1.3 ORB classes are not overridden.

   You can specify the iPlanet ORB classes as properties on the Java command line (the command must be all on one line):

```
java -Dorg.omg.CORBA.ORBClass=com.netscape.ejb.client.ClientORB
-Dorg.omg.CORBA.ORBSingletonClass=com.sun.corba.ee.internal.corba.ORBSingleton
j2eeguide.converter.ConverterClient ias_host 9010
```

Now that you've configured the existing JDK to use the iPlanet ORB, you need to install several supporting JAR files in your client environment. Proceed to "Installing RMI/IIOP Client Support Classes," on page 215 to install these JAR files.

Installing RMI/IIOP Client Support Classes

Regardless of the Java 2 platform used on the client side, the client's CLASSPATH must include the file `iasclient.jar`, an iPlanet-specific JAR file containing several security-related classes supporting iPlanet's client authentication feature (if you are using ACC, `iasclient.jar` is replaced by `iasacc.jar`). The standard `javax.jar` file must also be included in your client CLASSPATH. This file contains standard Java interfaces for naming services and other Java extensions.

These JAR files can be copied from an iPlanet installation to your client environment and added to the client's CLASSPATH. On UNIX, you can find these files in the following location of an iPlanet Application Server installation:

install_dir/ias/classes/java/iasclient.jar

install_dir/ias/classes/java/javax.jar

On Windows, you can find these files in the following location of an iPlanet Application Server installation:

install_dir/ias/classes/java/iasclient.jar

install_dir/ias/lib/java/javax.jar

Once you've copied these supporting files to the client environment, you must configure the client's CLASSPATH to include the JAR files.

RMI/IIOP Client Access to EJBs on Same System

If you are experimenting with RMI/IIOP client access using a client that is on the same machine as the application server, you can take a shortcut to setting up the PATH and CLASSPATH variables. Simply reference the existing, pre-installed copies of the javax.jar, iasclient.jar, and the JVM in usr/java/bin/. For example, to test RMI/IIOP access locally, set the CLASSPATH variable as follows:

On Windows NT:

```
set CLASSPATH=d:\iplanet\ias6\ias\lib\java\javax.jar;
d:\iplanet\ias6\ias\classes\java\iasclient.jar;%CLASSPATH%
```

(The Windows System PATH environment variable already contains usr/java/bin/ of the bundled JDK, so there is no need to set this again on Windows.)

You could set the Windows System CLASSPATH to avoid having to manually set the variable.

On UNIX:

```
export CLASSPATH=/usr/iplanet/ias6/ias/classes/java/javax.jar:
/usr/iplanet/ias6/ias/classes/java/iasclient.jar:$CLASSPATH
```

On UNIX, you must also modify the PATH to include the bundled JDK directory:

```
export PATH=/usr/iplanet/ias6/ias/usr/java/bin:$PATH
```

RMI/IIOP Client Access to EJBs from a Remote System

If you are using a remote client system, follow these steps to establish the appropriate PATH and CLASSPATH settings.

On UNIX:

Set your PATH environment variable to include the appropriate Java 2 `bin/` directory:

```
export PATH=Java2_install_dir/usr/java/bin:$PATH
```

Set your CLASSPATH to include the standard Java Extension classes and the iPlanet RMI/IIOP client support JAR:

```
export CLASSPATH=/opt/rmi-client/iasclient.jar:
/opt/rmi-client/javax.jar:$CLASSPATH
```

Double check the CLASSPATH to ensure that it is set correctly (your CLASSPATH may vary from the one shown below):

```
echo $CLASSPATH
```

```
/opt/rmi-client/iasclient.jar:/opt/rmi-client/javax.jar:
```

On Windows NT:

Set your PATH environment variable to include the appropriate Java 2 `bin/` directory:

```
set PATH=Java2_install_dir\usr\java\bin;%PATH%
```

Set your CLASSPATH to include the standard Java Extension classes (`javax.jar`) and the iPlanet client support JAR (`iasclient.jar`):

```
set CLASSPATH=d:\rmi-client\javax.jar;d:
\rmi-client\iasclient.jar;%CLASSPATH%
```

Deploying Client Applications

As you develop client applications, you will need to deploy a number of files from your development environment to the client system. This section addresses the underlying steps required to deploy an RMI/IIOP-capable client application in the following sections:

- Client Deployment
- Deployment Tools
- Server CLASSPATH Setting (SP2 and Prior)

Client Deployment

In addition to ensuring that client application classes are available on the client system, you must ensure that EJB-specific home and remote interfaces and their corresponding stubs are deployed to the client system. For example, in the Converter sample application, the following classes must be copied to the client system:

Home and Remote Interface Classes:

`ConverterHome.class`

`Converter.class`

EJB-Specific iPlanet Client Stubs:

`_Converter_Stub.class`

`_ConverterHome_Stub.class`

Deployment Tools

The Deployment Tool creates a JAR file containing only the home and remote interfaces and the RMI/IIOP stub classes. The tool does not currently package the rest of your client application classes and resources.

You can easily automate assembly of your client application via the Java-based Ant build facility. Refer to the RMI/IIOP sample applications for examples of using Ant to both package and deploy client applications.

Server CLASSPATH Setting (SP2 and Prior)

This section applies to iPlanet Application Server 6.0 Service Pack 2 (SP2) and earlier. Service Packs 3 and later do not require the following configuration step. If you are using SP3 or beyond, skip to the next section, “Running Client Applications,” on page 219.

In iPlanet Application Server Service Pack 2 and earlier, to load EJB classes, the RMI/IIOP Bridge process must be able to access the EJB stubs and home and remote interfaces via the application server’s CLASSPATH. Before the first execution of RMI/IIOP-based Java application client in SP2 or earlier, you must first modify the CLASSPATH of the application server.

With the advent of iPlanet Application Server 6.0 SP2, registration of EJB-based applications results in the EJB JAR file being expanded to the application server's deployment directory. By default, when a J2EE application such as `j2eeguide-converter.ear` is deployed to the application server, the embedded EJB JAR file, `j2eeguideEjb.jar` in this example, is expanded to:

install_dir`/ias/APPS/j2eeguide-converter/j2eeguide-converterEjb/`

When a stand-alone EJB JAR module (or WAR module) is deployed to iPlanet Application Server, the default expansion location for the stand-alone module is:

install_dir`/ias/APPS/modules/j2eeguide-converterEjb/`

Prior to running the RMI/IIOP client, you must add the appropriate module directory to the CLASSPATH of the application server.

Running Client Applications

If your client is a Java `main` program, then as long as the client environment is set appropriately and you are using a compatible JVM, you merely need to execute the `main` class. Depending on whether you are passing the IIOP URL components (host and port number) on the command line or obtaining this information from a properties file, the exact manner in which you execute the main program will vary. For example, the ConverterClient sample is executed in the following manner:

`java j2eeguide.converter.ConverterClient` *host_name port*

The *host_name* is the name of the host on which an RMI/IIOP Bridge is listening on the specified *port*.

Troubleshooting

When running an RMI/IIOP client, you may encounter error conditions on the client. To view the RMI/IIOP Bridge logs, see "Viewing RMI/IIOP Log Messages," on page 225. Table 9-1 lists common symptoms and fixes for common RMI/IIOP configuration problems.

If you are running the RMI/IIOP client application under load and are experiencing issues, see "Recognizing Performance Issues," on page 222 to understand how to troubleshoot load-related issues.

Table 9-1 Troubleshooting

Symptom	Probable Cause	Corrective Action
The client throws the following exception during JNDI lookup: `org.omg.CORBA.INITIALIZE: can't instantiate default ORB implementation`	The client CLASSPATH does not include the `iasclient.jar` file. The client PATH does not pickup appropriate java command. Either the JVM bundled with the application server or a suitable pre-existing JVM must be used.	Ensure that the client configuration steps were followed; see "Client Configuration," on page 209.
The client experiences a CORBA communication failure exception: `javax.naming.CommunicationException: Cannot connect to ORB. Root exception is org.omg.CORBA.COMM_FAILURE:`	Connection to the RMI/IIOP bridge fails because of one of the following reasons: • IIOP host and/or port number are incorrect. • RMI/IIOP Bridge process has not been started. • RMI/IIOP Bridge process was started, but has not finished initializing. • Client machine cannot access the network. • Firewall rules do not allow access to the Application Server system.	Ensure that the RMI/IIOP Bridge process is configured and started; see "Server Configuration," on page 208. Ensure that the client machine has network access and that intermediate firewalls are not blocking access.

Table 9-1 Troubleshooting

Symptom	Probable Cause	Corrective Action
a) The client appears to hang and then experiences an out of memory exception: `Exception in thread "main" java.lang.OutOfMemoryError.` b) The RMI/IIOP Bridge throws one of the following exceptions repeatedly: **Name Not Found:** `[01/May/2001 08:20:14:4] info: GDS-007: finished a registry load [01/May/2001 08:20:14:6] info: PROT-006: new connection established SendRemoteReq status=0x0 javax.naming.NameNotFoundException: EjbContext: exception on getHome(), com.nets cape.server.eb.UncheckedException: unchecked exception thrown by impl com.kivasoft.eb.boot.EBBootstrapImpl @1fca24a; nested exception is:` **Class not Found:** `[24/Jan/2001 12:25:52:9] error: EBFP-unserialize: error during unserialization of method, exception = java.lang.ClassNotFoundException: j2eeguide.confirmer.ejb_stub_Confirm erHome java.lang.ClassNotFoundException: j2eeguide.confirmer.ejb_stub_Confirm erHome at java.lang.Throwable.fillInStackTrace (Native Method)` **Class Cast Exceptions**	The JNDI name as specified in the client application is not correct. OR (Pre SP3) The expanded EJB JAR directory has not been added to the server CLASSPATH or the server has not been restarted since the EJB JAR directory was added to the CLASSPATH.	Correct the JNDI name used by the client. OR Set application server's CLASSPATH
The client application encounters a naming communication exception: `javax.naming.CommunicationException`	The Directory Server associated with the Application Server is not running.	Start the Directory Server.

Performance Tuning RMI/IIOP

For deployment environments in which you expect the RMI/IIOP path to support more than a handful of concurrent users, you should experiment with the tuning guidelines described in this section. The default configuration of the JVM and the underlying OS do not yield optimal performance and capacity when you are using RMI/IIOP.

This section covers the following topics:

- Recognizing Performance Issues
- Basic Tuning Approaches
- Enhancing Scalability

Recognizing Performance Issues

Before exercising your RMI/IIOP client application under load, ensure that you've verified that basic mechanical tests are completed successfully.

As you begin exercising the client application under load, you may experience the following exceptions on the RMI/IIOP client:

`org.omg.CORBA.COMM_FAILURE`

`java.lang.OutOfMemoryError`

`java.rmi.UnmarshalException`

If you've verified that the basic mechanics of your application are working properly and you experience any one of these exceptions while load testing your application, see the next section to learn how to tune the RMI/IIOP environment.

Basic Tuning Approaches

You should experiment with the following tuning recommendations in order to find the best balance for your specific environment.

Solaris File Descriptor Setting

On Solaris, setting the maximum number of open files property using `ulimit` has the biggest impact on your efforts to support the maximum number of RMI/IIOP clients. The default value for this property is `64` or `1024` depending on whether you are running Solaris 2.6 or Solaris 8. To increase the hard limit, add the following command to `/etc/system` and reboot it once:

```
set rlim_fd_max = 8192
```

You can verify this hard limit by using the following command:

```
ulimit -a -H
```

Once the above hard limit is set, you can increase the value of this property explicitly (up to this limit) using the following command:

```
ulimit -n 8192
```

You can verify this limit by using the following command:

```
ulimit -a
```

For example, with the default ulimit of 64, a simple test driver can support only 25 concurrent clients, but with `ulimit` set to `8192`, the same test driver can support 120 concurrent clients. The test driver spawned multiple threads, each of which performed a JNDI lookup and repeatedly called the same business method with a think (delay) time of 500ms between business method calls, exchanging data of about 100KB.

These settings apply to both RMI/IIOP clients (on Solaris) and to the RMI/IIOP Bridge installed on a Solaris system. Refer to Solaris documentation for more information on setting the file descriptor limits.

Java Heap Settings

Apart from tuning file descriptor capacities, you may want to experiment with different heap settings for both the client and Bridge JVMs. Refer to the JDK 1.2.2. documentation for information about modifying the default heap size.

Enhancing Scalability

Beyond tuning the capacity of a single Bridge process and client systems, you can improve the scalability of the RMI/IIOP environment by using multiple RMI/IIOP Bridge processes. You may find that configuring multiple Bridge processes on the same application server instance improves the scalability of your application deployment. In some cases, you may want to use a number of application server instances each configured with one or more Bridge processes.

In configurations where more than one Bridge process is active, you can partition the client load by either statically mapping sets of clients to different Bridges or by implementing your own logic on the client side to load balance against the known Bridge processes.

Firewall Configuration for RMI/IIOP

If the RMI/IIOP client is communicating through a firewall to the iPlanet Application Server, you must enable access from the client system to the IIOP port used by the RMI/IIOP Bridge processes. Since the client's port numbers are assigned dynamically, you must open up a range of source ports and a single destination port to allow RMI/IIOP traffic to flow from a client system through a firewall to an instance of the Application Server.

A snoop-based trace of the IIOP traffic between two systems during a single execution of the Converter sample application follows. The host `swatch` is the RMI/IIOP client, while the host `mamba` is the destination or Application Server system. The port number assigned to the RMI/IIOP Bridge process is `9010`. Note that the two dynamically assigned ports (`33046` and `33048`) are consumed on the RMI/IIOP client, while only port 9010 is used to communicate with the Bridge process.

```
swatch -> mamba.red.iplanet.com TCP D=9010 S=33046 Syn Seq=140303570 Len=0 Win=24820
Options=<nop,nop,sackOK,mss 1460>
mamba.red.iplanet.com -> swatch TCP D=33046 S=9010 Syn Ack=140303571 Seq=1229729413 Len=0 Win=8760
Options=<mss 1460>
swatch -> mamba.red.iplanet.com TCP D=9010 S=33046 Ack=1229729414 Seq=140303571 Len=0 Win=24820
swatch -> mamba.red.iplanet.com TCP D=9010 S=33046 Ack=1229729414 Seq=140303571 Len=236 Win=24820
mamba.red.iplanet.com -> swatch TCP D=33046 S=9010 Ack=140303807 Seq=1229729414 Len=168 Win=8524
swatch -> mamba.red.iplanet.com TCP D=9010 S=33046 Ack=1229729582 Seq=140303807 Len=0 Win=24820
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Syn Seq=140990388 Len=0 Win=24820
Options=<nop,nop,sackOK,mss 1460>
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Syn Ack=140990389 Seq=1229731472 Len=0 Win=8760
Options=<mss 1460>
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229731473 Seq=140990389 Len=0 Win=24820
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229731473 Seq=140990389 Len=285 Win=24820
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140990674 Seq=1229731473 Len=184 Win=8475
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229731657 Seq=140990674 Len=0 Win=24820
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229731657 Seq=140990674 Len=132 Win=24820
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140990806 Seq=1229731657 Len=25 Win=8343
```

```
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229731682 Seq=140990806 Len=0 Win=24820
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229731682 Seq=140990806 Len=124 Win=24820
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140990930 Seq=1229731682 Len=0 Win=8219
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140990930 Seq=1229731682 Len=336 Win=8219
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229732018 Seq=140990930 Len=120 Win=24820
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140991050 Seq=1229732018 Len=0 Win=8099
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140991050 Seq=1229732018 Len=32 Win=8099
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229732050 Seq=140991050 Len=120 Win=24820
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140991170 Seq=1229732050 Len=0 Win=7979
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140991170 Seq=1229732050 Len=32 Win=7979
swatch -> mamba.red.iplanet.com TCP D=9010 S=33046 Fin Ack=1229729582 Seq=140303807 Len=0 Win=24820
mamba.red.iplanet.com -> swatch TCP D=33046 S=9010 Ack=140303808 Seq=1229729582 Len=0 Win=8524
mamba.red.iplanet.com -> swatch TCP D=33046 S=9010 Fin Ack=140303808 Seq=1229729582 Len=0 Win=8524
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Fin Ack=1229732082 Seq=140991170 Len=0 Win=24820
swatch -> mamba.red.iplanet.com TCP D=9010 S=33046 Ack=1229729583 Seq=140303808 Len=0 Win=24820
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140991171 Seq=1229732082 Len=0 Win=7979
mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Fin Ack=140991171 Seq=1229732082 Len=0 Win=7979
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229732083 Seq=140991171 Len=0 Win=24820
```

Viewing RMI/IIOP Log Messages

Log messages generated by the RMI/IIOP path can be monitored by reviewing the log file generated by the RMI/IIOP Bridge process. Since the RMI/IIOP Bridge process is a form of a Java Engine (`kjs`), you monitor these logs in the same manner as you would monitor the Java Engines supporting the web and EJB containers. To view the appropriate log file, you must identify the Java Engine that is playing the role of the RMI/IIOP Bridge.

Monitoring Logs on Windows

By default, on a Windows installation of iPlanet Application Server, the Java Engine log files are not automatically displayed during startup of the Application Server. Most developers find it convenient to enable automatic display of console log information by performing the following steps:

1. Select Start->Settings->Control Panel.
2. Double click on Services.
3. Find the “iPlanet Application Server 6.0” entry and select it.
4. Click on Startup.
5. Click on “Allow Service to Interact with Desktop” and click on OK.
6. Click on Stop to stop the Application Server.
7. Click on Start to start the Application Server.

As the application server starts, a number of MS DOS output windows appear on the desktop. A single output window is present for each physical process in the application server. As the engines start, look for the Java Engines and, in particular, the engine that specifies the port number defined in the CXS (Bridge) process.

To enable vertical scroll bars in these output windows, follow these steps:

1. Select the MS DOS icon at the upper left of the output window.
2. Select Properties.
3. Select Layout.
4. Set the Screen Buffer Size Height to 200 or as desired.
5. Answer Yes when asked to apply these changes to all invocations of this window.

Monitoring Logs on UNIX

On UNIX, most developers use the `tail -f` command to monitor the application server log files of the process of interest. To monitor the Java Engine logs in this manner, follow these steps:

1. Navigate to the logs directory:

   `cd` *install_dir*`/ias/logs`

2. Execute the `tail` command on one of the Java Engine (`kjs`) and the Executive Service (`kxs`) processes:

   ```
   tail -f kjs_2*
   ```

   You must select the appropriate Java Engine log file to monitor. Java Engines are numbered according to how they are defined in the Administration Tool. Although the CXS (Bridge) process is typically the highest numbered Java Engine log file, double check the port number information within the log file to confirm which log files is generated by the CXS process.

3. Press Control-C to kill the `tail` command.

Sample Applications

A list of RMI/IIOP-oriented samples is available under the following location of your web server's document root or under the installation directory of the Application Server:

`http://`*webserver_host*`/ias-samples/ -> RMI/IIOP`

install_dir`/ias/ias-samples/index.html -> RMI/IIOP`

Converter Sample Application

The Currency Converter sample application from Sun's *J2EE Developer's Guide* has been bundled with iPlanet Application Server. This sample has been augmented with detailed setup instructions for deploying the application to iPlanet Application Server. It is recommended that you follow the detailed setup instructions for this sample and exercise the Converter sample prior to deploying other RMI/IIOP-based applications. Currency Converter setup documentation and source code are available at the following locations:

install_dir`/ias/ias-samples/j2eeguide/docs/converter.html`

install_dir`/ias/ias-samples/j2eeguide/converter/src/`

Other RMI/IIOP Sample Applications

Many of the *J2EE Developer's Guide* samples bundled with iPlanet Application Server include RMI/IIOP client programs. These are relatively simple samples that demonstrate various facets of the EJB specification. You can find these samples at:

install_dir`/ias/ias-samples/j2eeguide/docs/index.html`

Chapter 10

Deployment Packaging

This chapter describes the iPlanet Application Server modules content and how they are packaged to create an iPlanet Application Server application `.ear` file used for application deployment. The iPlanet Application Server modules include J2EE standard elements and iPlanet Application Server specific elements. Only iPlanet Application Server 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
- RMI/IIOP Client XML DTD
- Resource XML DTD

Overview of Packaging and Deployment

An iPlanet Application Server application is comprised of one or more iPlanet Application Server application modules, an iPlanet Application Server Deployment Descriptor (DD) and a J2EE application DD.

All items are packaged, using the Java ARchive (`.jar`) file format, into one file with an extension of `.ear`. Package definitions must be used in the source code so the class loader can properly locate the classes.

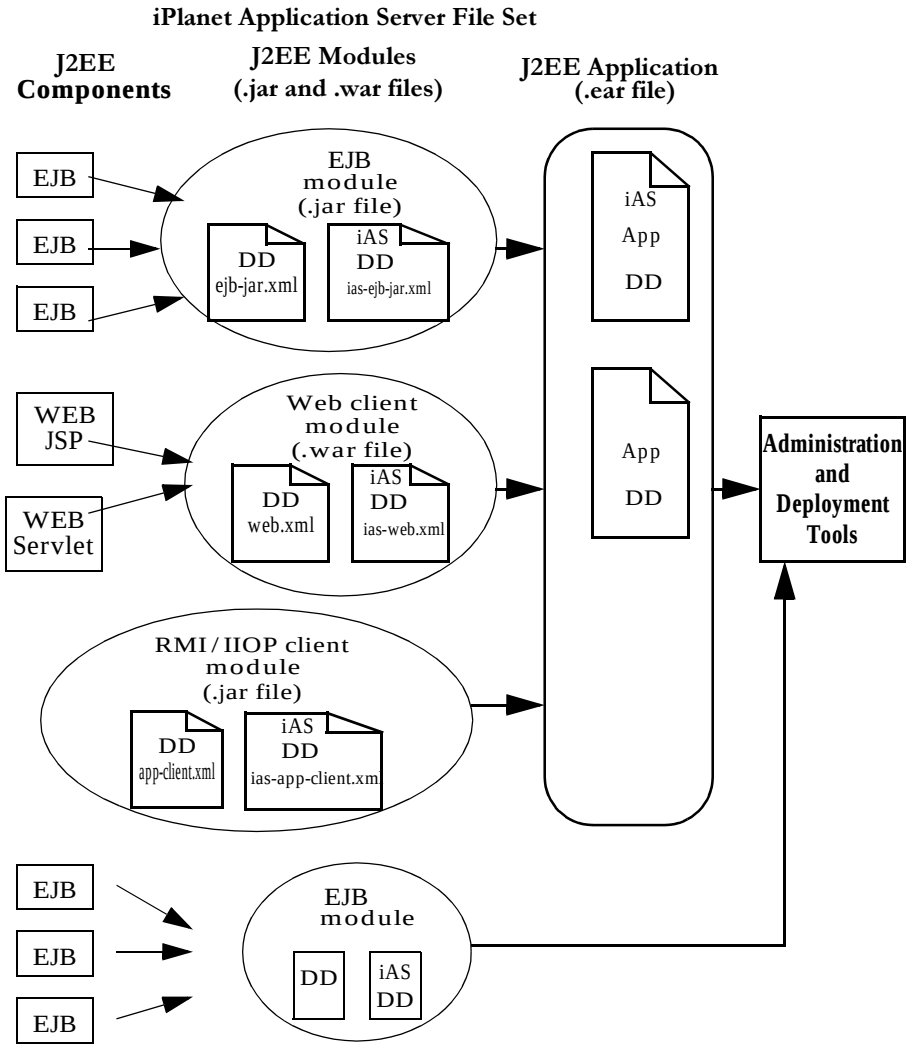
The DDs are used by the iPlanet Application Server Deployment Tool to deploy the application components and to register the resources with the iPlanet Application Server.

Each application module consists of either an EJB, web application, RMI/IIOP application, or resource components, as well as an iPlanet Application Server DD, and a J2EE DD. For example, an EJB module would have one or more EJB components, an EJB DD and an iPlanet Application Server EJB DD. There are two DDs for each module; one is a J2EE standard DD and the other is an iPlanet Application Server specific DD specified in this chapter.

DDs application module portions are standardized in the J2EE specification, v1.1. For more information on these standards, see 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 following diagram illustrates how components are packaged into modules and then assembled into an iPlanet Application Server application `.ear` file ready for deployment.



Introducing XML DTDs

The Document Type Definition (DTD) defines the DDs XML grammar. There are two DD levels; application level descriptors and component level descriptors. Applications are comprised of one or more components, one J2EE application DD and one iPlanet Application Server application DD.

The J2EE platform provides packaging and deployment facilities. These facilities use JAR files as the standard package for components and applications, and XML-based DDs for customizing parameters. For more information on the J2EE packaging and deployment process, see Developing Enterprise Applications with the J2EE, v 1.0, Chapter 7.

Application Deployment Descriptor

An application DD, lists the application components as modules. An iPlanet Application Server applications have two application DDs; one is a standard J2EE application DD and the other is an iPlanet Application Server application DD. The iPlanet Application Server application DD describes iPlanet specific deployment information for the iPlanet Application Server value-added features.

Component Deployment Descriptors

On the component level, each component group is packaged into a module with a J2EE component DD and an iPlanet Application Server component DD. An iPlanet Application Server application supports web application, EJB, RMI/IIOP Client, and resource components. For each component type, there is a DTD to define the XML elements supported for that descriptor type.

Creating Deployment Descriptors

All DDs for an iPlanet Application Server application are created using the Deployment Tool. For more information on these procedures, see the *Deployment Tool Online Help*.

Deployment Descriptors

The iPlanet Application Server require DDs to run an application. The DDs are XML files containing metadata describing the deployment information about the J2EE modules (such as servlets, JSPs and EJBs) that make up an application. The information in each XML file is stored in an iPlanet Application Server internal registry.

Each application module must have an iPlanet Application Server DD file, and a J2EE DD file. Additionally, each application component must be associated with a Globally Unique IDentifier, or a `GUID`.

The following lists the DD types supported by the iPlanet Application Server:

- application DD and iPlanet Application Server application DD
- web application DD and an iPlanet Application Server web application DD
- EJB DD and an iPlanet Application Server EJB DD
- application client DD and an iPlanet Application Server RMI/IIOP client DD
- iPlanet Application Server resource DD

Document Type Definition

The DTD describes the DD files structure and class properties. Each DD has exactly one element that completely contains all other elements (or subelements).

The element descriptions found in XML files are presented in a table format. These element tables have several fields to describe the element's purpose and setting parameters. Some elements are hierarchical, meaning the parameters have other elements (or subelements). If a parameter contains an element, the element description is found in another table describing the element. Table 10-1 shows the supported DTD entries.

Table 10-1 Document Type Definition

Type	Description
Element	Element name as it appears in the XML file and an element description.
Sub Elements	Lists the elements contained by this element.

The iPlanet Application Server Registry

The iPlanet Application Server 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 the iPlanet Application Server gains access to servlets, EJBs, and other application resources is called registration, because it involves placing entries in the iPlanet Application Server registry for each item.

You can change some information in the registry at runtime using the iPlanet Application Server Administrator Tool. For more information about the registry and the Administrator Tool, see the iPlanet Application Server Deployment Tool Help and the *Administrator's Guide*.

A Globally Unique Identifier

A `GUID` is a 128-bit hexadecimal number assigned to EJBs, servlets, and optionally to JSPs. They are automatically generated by the Deployment Tool.

`GUID`s are guaranteed to be globally unique, which makes them ideal for identifying components in a large scale heterogeneous system such as an iPlanet Application Server application.

`GUID`s are normally assigned automatically by the Deployment Tool. You can manually generate a `GUID` 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 DTD describes the application DD. There is a J2EE application DD, defined by the J2EE specification and an iPlanet Application Server application DD that is defined in this chapter.

The Deployment Tool is used to create the application DD and also to deploy the application. For more information on these procedures, see the *Deployment Tool Online Help*.

J2EE Application DTD

For more information about the application DTD description, see the J2EE specification, v1.2 Chapter 8.4.

iPlanet Application Server Application DTD

This is the iPlanet Application Server specific DTD for an application `.ear` file.

Table 10-2 shows the `ias-app` root element of the iPlanet Application Server application DD.

Table 10-2 `ias-app` Root Element

Sub Element	Repeat Rule	Contains	Default	Description
`role-mapping`	zero or more	elements	none	This field creates the mapping between the role name as known in the `AppComponent`, and then mapped onto one or more LDAP defined user, group, and so on. The deployment code simply treats each `role-IMPL` as an opaque string interpreted by the security infrastructure.

Table 10-3 shows the `role-mapping` sub element which maps role names to LDAP user, groups, and so on.

Table 10-3 `role-mapping` Sub Element

Sub Element	Repeat Rule	Contains	Default	Description
`role-name`	one and only one	string	none	The role name referred to in the `<security-role>` element.
`role-impl`	one and only one	elements	none	A string that represents an LDAP group and / or user that makes up a particular `role-name`. A `role-impl` is any number of groups and / or users.

Table 10-4 shows the `role-impl` sub element which IMPLs role names to LDAP user, groups, and so on.

Table 10-4 `role-impl` Sub Element

Sub Element	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 describes a web application, the web application module and the web application DD. DDs are created using the Deployment Tool. For more information, see the iPlanet Application Server Deployment Tool Help and the *Administrator's Guide*.

Web Application Overview

Web applications run on web servers and may consist of servlets, JSPs, JSP Tag libraries, HTML pages, classes and other resources. A web application's location is rooted at a specific path within the web server. A web application's instance must only be run on one Virtual Machine (VM) at any given time, unless the application is marked as distributable by its DD. When marked as distributable, the application may run on more than one VM at any given time and must follow a more restrictive rule set outlined by the Java Servlet 2.2 specification.

A web application is a composite of the following items:

- Servlets
- JSPs
- Utility Classes
- Static documents (HTML, images, sounds, and so on)
- Client side applets, beans and classes
- Descriptive meta information bundling the above items together

A web application is created by first assembling all needed web components into a web application module along with its module DD. Next, the web application module is packaged with all other modules that are used by the J2EE application along with the application DD into the final web application that is ready for deployment. For more information on J2EE assembly and deployment, see the J2EE specification, Chapter 8.

Web Application XML DTD

This section provides the XML DTD for the iPlanet Application Server specific web application DD. For more information on the standard J2EE application DD, see the J2EE specification, section 8.4.

The web application DD supports element definitions that provide the following information:

- servlet information
- session information
- EJB reference information
- Resource reference information
- Specifying servlet information

Element for Specifying an iPlanet Application Server Web Application

Table 10-5 shows the `<ias-web-app>` element and sub elements used with the iPlanet Application Server web application DD root element.

Table 10-5 `<ias-web-app>` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`servlet`	zero or more	elements	none	Contains the servlet configuration information.
`session-info`	zero or one	elements	none	Specifies the session information.
`ejb-ref`	zero or more	elements	none	Specifies the absolute JNDI name storage location of the corresponding J2EE XML `ejb-ref` entry.
`resource-ref`	zero or more	elements	none	Specifies the absolute JNDI name storage location of the `ejb-link` in the corresponding J2EE XML file `ejb-ref` entry.
`nlsinfo`	zero or one	elements	none	NLS settings descriptor.
`role-mapping`	zero or many	elements	none	LDAP role mapping descriptor.

Elements for Specifying Servlet Configuration Information

Table 10-6 shows the `servlet` sub element contains configuration information about a servlet.

Table 10-6 `servlet` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`servlet-name`	one and only one	string	none	The servlet name. This name must match the `servlet-name` parameter in the J2EE web app XML exactly.
`guid`	one and only one	string	none	A string representing the `guid` for the servlet.
`servlet-info`	zero or one	elements	none	Optional servlet characteristics.
`validationRequired`	zero or one	boolean	“false”	Specifies if the input parameter needs to be validated.
`error-handler`	zero or one	string	none	Describes the servlet error handler.
`parameters`	zero to more	elements	none	Describes all 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.

Elements for Specifying Servlet Characteristics

Table 10-7 shows the `servlet-info` sub element which is used to describe the optional characteristics about a servlet.

Table 10-7 `servlet-info` Sub Elements

Sub Element	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 continues to be routed to the same engine.
`encrypt`	zero or one	boolean	“false”	Optional flag indicating whether communications to the servlet are encrypted (“true”) or not (“false”).

Table 10-7 `servlet-info` Sub Elements *(Continued)*

Sub Element	Repeat Rule	Contains	Default	Description
`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`.
`server-info`	zero or many	elements	none	Optional server information including server and load balancing enabling and / or disabling.
`server-ip`	one and only one	string	none	Server IP address.
`server-port`	one and only one	string	none	Executive Server's port number.
`sticky-lb`	zero or many	boolean	`servlet -info sticky` setting	Sets sticky load balancing. Legal values are `true` or `false`. If set overrides the 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`.

Elements for Specifying Servlet Validation

Table 10-8 shows the `validation-required` sub element which is used to verify the input about a servlet should be validated.

Table 10-8 `validation-required` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`validation-required`	one and only one	boolean	`false`	Specifies whether or not the input parameters should be verified.

Elements for Specifying Servlet Caching

Table 10-9 shows the `caching` sub element, which is used to describe caching criteria for the servlet. `caching` is disabled by not defining the caching element.

Table 10-9 `caching` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`cache-timeout`	one and only one	integer	none	Sets the servlet caching timeout (in seconds). If the value is 0, caching is disabled.
`cache-size`	one and only one	integer	none	Sets the cache size. 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; for details, see "Caching Servlet Results" on page 293.	none	Criteria expression containing a string of comma delimited descriptors, each descriptor defining a match with one of the input parameters to the servlet.
`cache-option`	one and only one	String of either `TIMEOUT_CREATE` or `TIMEOUT_LASTACCESS`	`TIMEOUT_ LASTACCESS`	Sets the cache timeout option.

Examples for Setting Cache Criteria and Cache Option

The following examples provide some common usages and cache criteria element settings.

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 if the `stock` input parameter value is `NSCP`

Example 3

```
<cache-criteria>*</cache-criteria>
```

This means caching is enabled whenever the input parameter values are the same as the cached value.

Example 4

```
<cache-criteria>dept=sales|marketing|support</cache-criteria>
```

This means caching is enabled if the `dept` parameter value is sales, marketing, or support.

Example 5

```
<cache-criteria>salary=40000-60000</cache-criteria>
```

This means caching is enabled when the input parameter value of `salary` is between `40000` and `60000`.

Example 6

```
<cache-option>TIMEOUT_CREATE</cache-option>
```

This means it takes the cache timeout value from the creation time.

Example 7

```
<cache-option>TIMEOUT_LASTACCESS</cache-option>
```

This means it takes the cache timeout based on the last accessed time.

Elements for Specifying Servlet Parameters

Table 10-10 shows the `parameters` element which is used to describe the input parameters to be validated.

Table 10-10 `parameters` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`param`	zero or more	elements	none	Specifies each parameter by name and the rules applied to it for validation.

Elements for Specifying Servlet Sub Parameters

Table 10-11 shows the `param` sub elements where each parameter is represented by a name and the rules that are applied to it for validation.

Table 10-11 `param` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`param-name`	one and only one	string	none	Input parameter name.
`input-fields`	one and only one	elements	none	This describes the input parameter details.

Elements for Specifying Servlet Input Field

Table 10-12 shows the `input-field` sub elements which is used to detail the input parameter.

Table 10-12 `input-field` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`input-required`	zero or one	boolean	none	Specifies whether the input parameter is required to exist, that is, 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.
`param-error-handler`	zero or one	string	none	Specifies the parameter error handler.

Elements for Specifying Servlet Parameter Groups

Table 10-13 shows the `param-group` sub elements where each parameter group is represented by an event source name and the associated parameters.

Table 10-13 `param-group` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`param-group-name`	one and only one	string	none	Parameter group name.
`param-input`	one or more	string	none	Parameter input name associated with the parameter group.

Elements for Specifying Session Information

Table 10-14 shows the `session-info` elements which is used to specifies session information.

Table 10-14 `session-info` Sub Elements

Sub Element	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	Number of session timeout minutes before a timeout. If unspecified, a system wide default session timeout is used
`secure`	zero or one	boolean	`false`	Specifies the session can only be visible to a secure (HTTPS) server.

Table 10-14 `session-info` Sub Elements *(Continued)*

Sub Element	Repeat Rule	Contains	Default	Description
`domain`	zero or one	string name of the domain that set the cookie	none	Specifies the application domain used to set the session domain cookie. 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 `Who.acme.com`, `bar.asme.com`, and so on.
`path`	zero or one	String value of the URL for the session cookie starting with "`/`".	The URL that created the cookie.	Specifies the session cookie path. A non-existent path implies the same path as the one set in the cookie is used. For example, the path `/phoenix` matches `/phoenix/types/bird.html` and `/phoenix/birds.html`.
`scope`	zero or one	String identifying the other application.	none	Grouping name that selects what other applications can access the session. For example, if the domain is set to `acme.com`, then the session is visible to `Who.acme.com`, `bar.acme.com`, and so on.
`dsync-type`	zero or one	string of either `dsync-local` or `dsync-distributed`	none	Specifies the DSync session type.

Elements for Specifying EJB Reference Information

Table 10-15 shows the `ejb-ref` sub elements which are the absolute `jndi-name` storage place for the `ejb-link` in the corresponding J2EE XML file `ejb-ref` entry.

Table 10-15 `ejb-ref` Sub Elements

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

Table 10-16 shows the `resource-ref` sub elements which are the absolute `jndi-name` storage place for the `resource-ref` in the corresponding J2EE XML file `resource-ref` entry.

Table 10-16 `resource-ref` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`resource-ref-name`	one and only one	string	none	The `resource-ref` name 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

Table 10-17 shows the `nlsinfo` sub elements which contains the configuration information about the application's NLS settings.

Table 10-17 `nlsinfo` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`locale-charset-map`	zero or many	elements	none	Contains locale and the corresponding character set.
`default-locale`	one and only one	string	none	Default locale.

Elements for Specifying Locale Character Sets

Table 10-18 shows the `locale-charset-map` sub elements which contains the descriptor information for locale and corresponding character sets.

Table 10-18 `locale-charset-map` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`locale`	one and only one	string	none	Locale name.
`charset`	one and only one	string	none	Default locale.

Elements for Specifying Role Mapping

Table 10-19 shows the `role-mapping` sub elements which contains the descriptor information for mapping roles to LDAP user, groups, and so on.

Table 10-19 `role-mapping` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`role-name`	one and only one	string	none	Role name 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 which makes up a particular `role-name`. A `role-impl` could be any number of groups and / or users.

Elements for Specifying Role IMPL

Table 10-20 shows the `role-impl` sub elements which contains the descriptor information for role implementation.

Table 10-20 `role-impl` Sub Elements

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 describes the EJB DTD used by the EJB deployment descriptors. The DDs are created using the Deployment Tool. For more information on creating DDs, see the *Deployment Tool Online Help*.

EJB JAR File Contents

The standard format used to package enterprise beans is the EJB-JAR file. The 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 DD as well as all class files for the following:

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

In addition, the EJB-JAR file must contain the class files for all 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, the iPlanet Application Server 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 are used. In certain cases, you may want to ensure that parameters passed to a bean are always passed by value. The iPlanet Application Server supports the marking of a bean or even a particular method within a bean as requiring `pass-by-value` semantics. The parameter passing method used by the EJB is defined by the `pass-by-value` element. For more information, see the `pass-by-value` element description in the `session` (Table 10-23) or `entity` element (Table 10-24). Because this option decreases performance by greatly increasing call overhead, the default value is `false`.

EJB iPlanet Application Server XML DTD

The following is the iPlanet Application Server specific XML DTD for EJB-JAR files.

Elements for Specifying EJB-JAR

Table 10-21 shows the `ias-ejb-jar` element which is the iPlanet Application Server web application DD root element.

Table 10-21 `ias-ejb-jar` Element

Sub Element	Repeat Rule	Contains	Default	Description
`enterprise-beans`	one and only one	element	none	The `enterprise-beans` element contains declarations for one or more enterprise beans.

Elements for Specifying Enterprise Beans

Table 10-22 shows the `enterprise-beans` sub element which contains declarations for one or more enterprise beans.

Table 10-22 `enterprise-beans` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`session`	one or the other	element	none	An element that declares all iPlanet Application Server specific session bean related deployment information
`entity`	one or the other	element	none	An element that declares all iPlanet Application Server specific entity bean related deployment information

Elements for Specifying Session

Table 10-23 shows the `session` sub element which declares all iPlanet Application Server specific session bean related deployment information. The `ejb-name` *must* match 1 to 1 with the `ejb-name` declared in the J2EE XML file.

Table 10-23 `session` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`ejb-name`	one and only one	string	none	The EJB name.

Table 10-23 `session` Sub Elements *(Continued)*

Sub Element	Repeat Rule	Contains	Default	Description
`guid`	one and only one	string	none	The EJB `guid` 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 the Administration Tool.
`pass-by-value`	one and only one	boolean	none	If “true”, marshalling of all call parameters to the EJB are 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	The session timeout in minutes.
`ejb-ref`	zero or more	elements	none	The absolute `jndi-name` storage place for the `ejb-link` in the corresponding J2EE XML file `ejb-ref` entry.
`resource-ref`	zero or more	elements	none	The absolute `jndi-name` storage place for the `resource-ref` in the corresponding J2EE XML file `resource-ref` entry.
`failoverrequired`	zero or one	boolean	none	Indicates if failover is required.

Elements for Specifying Entity

Table 10-24 shows the `entity` sub element which declares all iPlanet Application Server specific entity bean related deployment information. The `ejb-name` *must* match 1 to 1 with the `ejb-name` declared in the J2EE XML file.

Table 10-24 `entity` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`ejb-name`	one and only one	string	none	The EJB name.
`guid`	one and only one	string	none	The EJB `guid` 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 the Administration Tool.

Table 10-24 `entity` Sub Elements *(Continued)*

Sub Element	Repeat Rule	Contains	Default	Description
`pass-by-value`	one and only one	boolean	none	If “true”, marshalling of all call parameters to the EJB are 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.
`pool-manager`	zero or one	elements	none	Descriptor for cache pool attributes.
`ejb-ref`	zero or more	elements	none	The absolute `jndi-name` storage place for the `ejb-link` in the corresponding J2EE XML file `ejb-ref` entry.
`resource-ref`	zero or more	elements	none	The absolute `jndi-name` storage place for the `resource-ref` in the corresponding J2EE XML file `resource-ref` entry.
`failover-required`	zero or one	boolean	`false`	Indicates if failover is required.
`iiop`	zero or one	boolean	`false`	Indicates if a bean is RMI/IIOP Client enabled.
`role-mapping`	zero or many	elements	none	Descriptor that creates role mapping.

Elements for Specifying Persistence Manager

Table 10-25 shows the `persistence-manager` sub element which defines all persistence manager specific information.

Table 10-25 `persistence-manager` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`factory-class-name`	one and only one	string	none	Persistence manager name factory class.
`properties-file-location`	one and only one	string	none	Properties file location in a JAR file.

Elements for Specifying Pool Manager

Table 10-26 shows the `pool-manager` sub element which defines all pool manager specific information.

Table 10-26 `pool-manager` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`commit-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 does not cache a "ready" instance. For more information, see the EJB v1.1 specification, section 9.1.10.
`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 the Administration Tool.
`ready-pool-maxsize`	one and only one	positive integer or "0" for infinite	infinite	Maximum size of the ready cache in entry numbers. This value can be changed during runtime by the Administration Tool.
`free-pool-maxsize`	one and only one	positive integer or "0" for infinite	infinite	Maximum size of the instance free pool in entry numbers. This value can be changed during runtime by the Administration Tool.

Elements for Specifying EJB Reference

Table 10-27 shows the `ejb-ref` sub element which are the absolute `jndi-name` storage places for the `ejb-link` in the corresponding J2EE XML file `ejb-ref` entry.

Table 10-27 `ejb-ref` Sub Elements

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

Table 10-28 shows the `resource-ref` sub element which are the absolute `jndi-name` storage places for the `resource-ref` in the corresponding J2EE XML file `resource-ref` entry.

Table 10-28 `resource-ref` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`resource-ref-name`	one and only one	string	none	The `resource-ref` name in the corresponding J2EE XML file `resource-ref` entry.
`jndi-name`	one and only one	string	none	The absolute `jndi-name`.

Elements for Specifying Role Mapping

Table 10-29 shows the `role-mapping` sub elements which are the mapping roles descriptors for the LDAP user, groups, and so on.

Table 10-29 `role-mapping` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`role-name`	one and only one	string	none	Role name 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.

Elements for Specifying Roll Implementation

Table 10-30 shows the `role-impl` sub elements which are the role implementation descriptors.

Table 10-30 `role-impl` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`group`	zero or many	string	none	LDAP specific string for a particular LDAP group.
`user`	zero or many	string	none	LDAP specific string for a particular LDAP user.

RMI/IIOP Client XML DTD

The RMI/IIOP Client is an iPlanet Application Server specific type of J2EE client. A RMI/IIOP Client supports the standard J2EE Application Client specifications, and in addition, supports direct access to the iPlanet Application Server. For more information on RMI/IIOP Clients, refer to Chapter 9, "Developing and Deploying RMI/IIOP-Based Clients."

A RMI/IIOP Client JAR file contains two DDs that are generated by the Deployment Tool. One DD is specified by the J2EE application client XML DTD, that can be found in the J2EE Specification, v1.0 Chapter 9 Application Clients. The other DD contains the iPlanet Application Server specific RMI/IIOP Client elements; for more information, see "iPlanet Application Server RMI/IIOP Client XML DTD" on page 254.

For a sample RMI/IIOP Client DD file, see "RMI/IIOP Client DD XML Files" on page 338.

iPlanet Application Server RMI/IIOP Client XML DTD

The `ias-java-client-jar` element is the RMI/IIOP Client's DD root element.

Elements for Specifying EJB Reference Information

Table 10-31 shows the `ejb-ref` sub elements which are the absolute `jndi-name` storage places for the `ejb-link` in the corresponding J2EE XML file `ejb-ref` entry.

Table 10-31 `ejb-ref` Sub Elements

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

Table 10-32 shows the `resource-ref` sub elements which are the absolute `jndi-name` storage places for the `resource-ref` in the corresponding J2EE XML file `resource-ref` entry.

Table 10-32 `resource-ref` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`resource-ref-name`	one and only one	string	none	The `resource-ref` name 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 iPlanet Application Server resource, such as a JDBC datasource, Java Mail and JMS have a resource XML file. The XML file contains entries that are used to register the resource with the iPlanet Application Server. These entries define the way the iPlanet Application Server connects to the resource. These files are generated by the Deployment Tool. This section describes the resource XML file entries. For information on how to generate these files, see the *Deployment Tool Online Help*.

Datasource XML DTD

This section describes the XML DTD for the iPlanet Application Server datasource.

Element for Specifying Datasources

Table 10-33 shows the `ias-Datasource-jar` sub element which is the resource DD root element.

Table 10-33 `ias-Datasource-jar` Sub Element

Sub Element	Repeat Rule	Contains	Default	Description
`ias-resource`	one and only one	element	none	Common element for all resource DDs.

Element for Specifying iPlanet Application Server Resources

Table 10-34 shows the `ias-resource` sub element which is the descriptor used for all resources.

Table 10-34 `ias-resource` Sub Element

Sub Element	Repeat Rule	Contains	Default	Description
`resource`	one and only one	elements	none	Common element for all resource DDs.

Elements for Specifying Resources

Table 10-35 shows the `resource` sub elements which are the descriptors used for all resources.

Table 10-35 `resource` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`jndi-name`	one and only one	string	none	The absolute `jndi-name` of the resource factory (for example, `jdb/Who`).
`jdbc`	one or the other	elements	none	Descriptor for the JDBC datasource.
`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.

Elements for Specifying JDBC Datasources

Table 10-36 shows the `jdbc` sub elements which are the descriptors used for the JDBC datasource.

Table 10-36 `jdbc` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`database`	one and only one	string	none	Database name to connect to.
`datasource`	one and only one	string	none	Assigned datasource name.
`username`	one and only one	string	none	Valid database user name.

Table 10-36 `jdbc` Sub Elements *(Continued)*

Sub Element	Repeat Rule	Contains	Default	Description
`password`	one and only one	string	none	Valid user name password.
`driver-type`	one and only one	string field which contains one of the following: `ORACLE_OCI` (Oracle) `DB2_CLI` (DB2) `INFORMIX_CLI` (Informix) `SYBASE_CTLIB` (Sybase) `ODBC` (ODBC)	none	EIS specific JDBC driver.
`resource-mgr`	zero or one	string	none	If this attribute is set, the datasource is available for distributed transactions through 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 you create for a resource manager under the `RESOURCEMGR` key.

RMI/IIOP Client Datasource XML DTD

This section describes the XML DTD for a RMI/IIOP Client datasource.

Elements for Specifying Java Client Resources

Table 10-37 shows the `ias-javaclient-resource` sub elements which are the RMI/IIOP Client's datasource XML DD root elements.

Table 10-37 `ias-javaclient-resource` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`jdbc`	one or the other	elements	none	Descriptor for RMI/IIOP 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`.

Elements for Specifying JDBC Settings

Table 10-38 shows the `jdbc` sub elements which are the JDBC settings descriptors.

Table 10-38 `jdbc` Sub Elements

Sub Element	Repeat Rule	Contains	Default	Description
`driverClass`	one and only one	elements	none	Valid driver class.
`connectUr1`	one and only one	string	none	Valid URL to connect to.
`userName`	one and only one	string	none	Valid user name.
`password`	one and only one	string	none	Valid user name password.

Chapter 11

Creating and Managing User Sessions

This chapter describes how to create and manage a session that allows users 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 a 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.

The iPlanet Application Server supports the servlet standard session interface, called `HttpSession` for all session activities. This interface enables you to write portable, secure servlets.

Additionally, the iPlanet Application Server provides an additional interface, called `HttpSession2`, which provides support for a servlet security framework, as well as, sharing sessions between servlets and older iPlanet Application Server components (that is, AppLogics).

Behind the scenes, there are two session styles, distributable and local. 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 (that is, bound to an individual server). Sticky load balancing is automatically set for application servlets configured to use the local session model. You determine which session style to use in the application configuration file. For more information about session-related elements in the application configuration file, see Chapter 10, “Deployment Packaging.”

Sessions and Cookies

A cookie is a small collection of information that can be transmitted to a calling browser and then retrieves it on each subsequent call from the browser so that the server can recognize calls from the same client. A cookie is returned with each 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 the URLs where the session ID appears in the URLs from that client.

Sessions and Security

The iPlanet Application Server security model is based on an authenticated user session. Once a session has been created the application user is authenticated (if used) and logged in to the session. Each interaction step from the servlet that receives an EJB request, generates content to a JSP to format the output and is aware the user is properly authenticated.

Additionally, you can specify that a session cookie is only passed on a secured connection (that is, 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, first create a session using the `HttpServletRequest` method `getSession()`. Once the session is established, examine and set its properties using the provided methods. If desired, set the session to time out after being inactive for a defined time period or invalidate it manually. You can also bind objects to the session which store 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 shown in the following example:

```
HttpSession mySession = request.getSession();
```

`getSession()` returns the valid session object associated with the request, identified in the session cookie which is encapsulated in the request object. Calling the method with no arguments, creates a session if one does not already exist which is associated with the request. Additionally, calling the method with a Boolean argument creates a session only if the argument is `true`.

The following example shows the `doPost()` method from a servlet which 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

   }
}
```

NOTE The `getSession()` method should be called before anything is written to the response stream. Otherwise the `SetCookie` string is placed in the HTTP response body instead of the HTTP header.

For more information about `getSession()`, see the Java Servlet Specification v2.2.

Examining Session Properties

Once a session ID has been established, use the methods in the `HttpSession` interface to examine session properties, and methods in the `HttpServletRequest` interface to examine request properties that relate to the session.

Table 11-1 shows the methods to examine session properties.

Table 11-1 `HttpSession` Methods

`HttpSession` **method**	**Description**
`getCreationTime()`	Returns the session time in milliseconds since January 1, 1970, 00:00:00 GMT.
`getId()`	Returns the assigned session identifier. An HTTP session's identifier is a unique string which is created and maintained by the server.
`getLastAccessedTime()`	Returns the last time the client sent a request carrying the assigned session identifier (or `-1` if its a new session) in milliseconds since January 1, 1970, 00:00:00 GMT.
`isNew()`	Returns a Boolean value indicating if the session is new. Its a new session, if the server has created it and the client has not sent a request to it. This means, the client has not *acknowledged* or *joined* the session and may not return the correct session identification information when making its next request.

For example:

```
String mySessionID = mySession.getId();
if ( mySession.isNew() ) {
    log.println(currentDate);
    log.println("client has not yet joined session " + mySessionID);
}
```

Table 11-2 shows the methods to inspect request object properties that relate to the session:

Table 11-2 `HttpServletRequest` Methods

`HttpServletRequest` **Methods**	**Description**
`getRemoteUser()`	Gets the requesting user name (HTTP authentication can provide the information). Returns null if the request has no user name information.
`getRequestedSessionId()`	Returns the session ID specified with the request. This may differ from the session ID in the current session if the session ID given by the client is invalid and a new session was created. Returns null if the request does not have a session associated with it.
`isRequestedSessionIdValid()`	Checks if the request is associated to a currently valid session. If the session requested is not valid, it is not returned through the `getSession()` method.
`isRequestedSessionIdFromCookie()`	Returns true if the request's session ID provided by the client is a cookie, or false otherwise.
`isRequestedSessionIdFromURL()`	Returns true if the request's session ID provided by the client is a 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` Methods	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 names of all values bound to the 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 the iPlanet Application Server `RowSets` and JDBC `ResultSets` are not `serializable` and cannot 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 require you to know when they are placed in or removed from, a session. To obtain this information, implement the `HttpSessionBindingListener` interface in those objects. When your application stores or removes data with the session, the servlet engine checks whether the object being bound or unbound implements `HttpSessionBindingListener`. If it does, the interface automatically notifies the object that it is bound or unbound.

Invalidating a Session

Specify the session to invalidate itself automatically after being inactive for a defined time period. Alternatively, invalidate the session manually with the `HttpSession` method `invalidate()`.

TIP The session API does not provide an explicit session logout API, so any *logout* implementation 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 removed.

Setting a Session Timeout

Session timeout is set using the `ias-specific` Deployment Descriptor. For more information, see the `session-info` element in Chapter 10, “Deployment Packaging.”

Controlling the Session Type

To control the session type set the appropriate elements in the iPlanet Application Server specific XML file. For more information, see the `session-info` element in Chapter 10, “Deployment Packaging.”

Sharing Sessions with AppLogics

Servlet programmers can use the iPlanet Application Server 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 iPlanet Application Server 6.0. `HttpSession2` interface adds security and direct distributable sessions manipulation.

Additionally, if you establish a session in an AppLogic using `loginSession()` and you want to access the session from a servlet, you must call the `setSessionVisibility()` method in the `AppLogic` class to instruct the session cookie to transmit to servlets as well as AppLogics. Additionally, this must be completed 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()`, refer to the `AppLogic` class in the *Foundation Class Reference (Java)*. 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 with components that perform user authentication, and access authorization to servlets and EJB business logic.

This chapter contains the following sections:

- iPlanet Application Server Security Goals
- iPlanet Application Server Specific Security Features
- iPlanet Application Server Security Model
- Security Responsibilities Overview
- 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 RMI/IIOP Clients
- Guide to Security Information
- Web Server to Application Server Component Security

iPlanet Application Server Security Goals

In an enterprise computing environment there are many security risks. The iPlanet Application Server's goal is to provide highly secure, interoperable, and distributed component computing based on the J2EE security model. The security goals for the iPlanet Application Server include:

- Full compliance with the J2EE v1.2 security model (for more information, see the J2EE specification, v1.2 Chapter 3 Security)
- Full compliance with the EJB v1.1 security model (for more information, see the Enterprise JavaBean specification v1.1 Chapter 15 Security Management). This includes EJB role-based authorization.
- Full compliance with the Java Servlet v2.2 security model (for more information, see the Java Servlet specification, v2.2 Chapter 11 Security). This includes servlet role-based authorization.
- Support for single *signon* across all iPlanet Application Server applications.
- Security support for RMI/IIOP Clients.
- Use of LDAP as the backend for security and allows user administration during runtime.
- Implements declarative iPlanet Application Server specific XML-based role mapping information.
- The iPlanet Application Server specific XML files with declarative security created by the iPlanet Application Server Deployment Tool.
- Backwards compatibility with AppLogic security APIs.

iPlanet Application Server Specific Security Features

The iPlanet Application Server supports the J2EE v1.2 security model, as well as the following features which are specific to the iPlanet Application Server:

- Single signon across all iPlanet Application Server applications.
- Security for RMI/IIOP Clients.
- iPlanet Application Server specific XML-based role mapping information.

- The GUI-based Deployment Tool is used to build XML files containing the security information.
- User administration LDAP during runtime.
- LDAP is used as the backend for security.

iPlanet Application Server Security Model

Secure applications require a client to be authenticated as a valid application user and have authorization to access the EJB business logic. The iPlanet Application Server supports security for both web and RMI/IIOP clients.

Web clients use a browser and a web server to communicate using HTTP with servlets running on the iPlanet Application Server. These clients require communication with servlets and JSPs to extend the web server functionality.

Applications with secure web and EJB containers may enforce the following security processes for web clients:

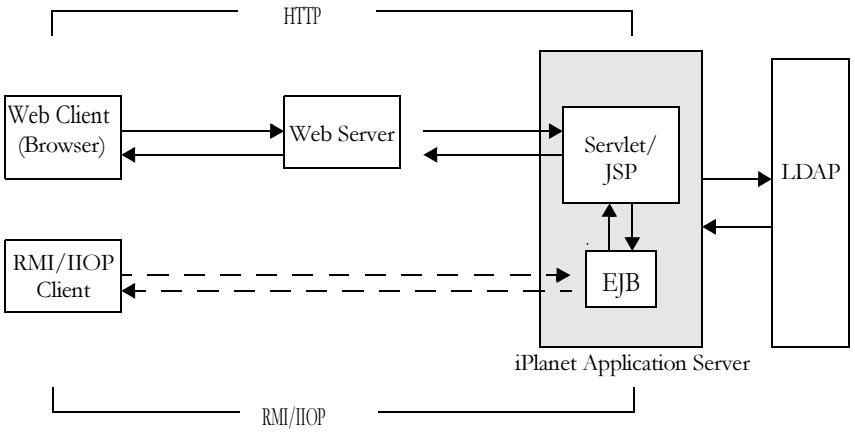
- authenticate the caller
- authorize the caller for access to the URL
- authorize the caller for access to the EJB business methods

RMI/IIOP clients communicate over a bridge using RMI/IIOP to directly access EJBs running on the iPlanet Application Server. RMI/IIOP clients directly invokes bean methods.

Applications with secure EJB containers may enforce the following security processes for RMI/IIOP clients:

- authorize the caller for access to the EJB business methods

The following diagram shows the iPlanet Application Server security model.



Web Client and URL Authorizations

Secure web containers may have authentication and authorization properties. The 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 authentication information (for example. username and password) from the web client and passing it to the iPlanet Application Server.

The iPlanet Application Server consults the security policies (derived from the Deployment Descriptor (DD)) associated with the web resource to determine the security roles used to permit resource access. The web container tests the user credentials against each role to determine if it can map the user to the role. The LDAP server, an enterprise-wide directory service for managing information about users, groups and roles, obtains the user 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 DD with

authorization properties which 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.

RMI/IIOP Client Invocation of Enterprise Bean Methods

For RMI/IIOP clients, a secure EJB container consults it's security policies to determine if the caller has the authority to access the bean method. This process is the same for both web and RMI/IIOP clients.

Security Responsibilities Overview

A J2EE platform's primary goal is to isolate the developer from the security mechanism details and facilitate a secure application deployment in diverse environments. This goal is addressed by providing mechanisms for the application security specification requirements declaratively and outside the application.

Application Developer

The application developer supplies the programmatic security including:

- Specifying security levels.
- Verifies the security permission levels when secure operations are being accessed.

Application Assembler

The application assembler or application component provider must identify all security dependencies embedded in a component including:

- All role names used by the components that call `isCallerInRole` or `isUserInRole`.
- References to all external resources accessed by the components.
- References to all intercomponent calls made by the component.

- Recommended that the assembler identify all method calls of each component's feature parameters and return values are to be protected for confidentiality and/or integrity. The Deployment Descriptor (DD) is used for this purpose.

Application Deployer

The iPlanet Application Server Deployment Tool is used to map the views provided by the assembler to the policies and mechanisms specific to the operational environment. The security mechanisms configured by the application deployer are implemented by the containers on behalf of the components hosted in the containers.

The application deployer takes all component security views provided by the assembler and uses them to secure a particular enterprise environment in the application, including:

- Assigning user groups to security levels.
- Refines the privileges required to access component methods and defines the correspondence between the security attributes presented by the callers and the container privileges.

Common Security Terminology

The most common security processes are authentication, authorization, and roll mapping, the following sections define their terminology.

Authentication

Authentication verifies the user. For example, the user may enter a username and password in a web browser and if those credentials match the permanent profile stored in the LDAP server then the user is authenticated. The user is associated with a security identity for the remainder of the session.

Authorization

Authorization permits a user to perform the desired operations, after being authenticated. For example, 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 the 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 who 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 are specific to a deployment domain. The deployer's role is to map roles into one or more groups.

In the iPlanet Application Server, roles correspond to user groups configured in the directory server. LDAP groups can contain both users and other groups.

Container Security

The component containers are responsible for providing J2EE application security. There are two security forms 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, as specified by the J2EE security model, to make business logic decisions based on the caller or remote user's security role. Programmatic security should only be used when declarative security alone is insufficient to meet 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. The iPlanet Application Server supports these interfaces as specified in the specification. For more information on programmatic security, see section 3.3.6 Programmatic Security, in the J2EE Specification,v1.2, and "Programmatic Login," on page 276.

Declarative Security

Declarative security is when the security mechanism for an application is declared and handled externally to the application. DDs are used by the iPlanet Application Server to describe the J2EE application's security structure, including security roles, access control, and authentication requirements.

The DDs for security aware applications, web-app containers, and EJB containers, have XML tags as security elements to express the application's security characteristics. Security characteristics include authentication and authorization.

The iPlanet Application Server supports the DTDs specified by J2EE v1.2 and has additional security elements included in the DDs.

Declarative security is the application deployer's responsibility. The XML DDs are generated by the iPlanet Application Server Deployment Tool. For more information, see the iPlanet Application Server Deployment Tool and the *Administrator's Guide*.

Application Level Security

The application XML DD contains authorization descriptors for all user roles when accessing the application's servlets and EJBs. On the application level, all roles used by any application container must be listed in this file. These roles are described by the `role-name` element in the application XML DD file. The 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 authenticates users and authorizes 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 DD.

User Authentication by Servlets

The three web-based login mechanisms required by the J2EE Specification, v1.2 are supported by the iPlanet Application Server. These three mechanisms include:

- HTTP Basic Authentication
- Secure Socket Layer Mutual Authentication
- Form-Based Login
- Programmatic Login

The web application DD `login-config` element describes the authentication method used, the application's realm name used by the HTTP basic authentication, and the form login mechanism's attributes.

The `login-config` element syntax is as follows:

```
<!ELEMENT login-config
(auth-method?,realm-name?,from-login-config?)>
```

For more information regarding web application DD elements, see Chapter 13, Deployment Descriptor of the Java Servlet Specification, v2.2.

HTTP Basic Authentication

HTTP basic authentication (RFC2068) is supported by the iPlanet Application Server. The HTTP basic authentication protocol indicates the HTTP realm by which access is being negotiated. Because passwords are sent with base64 encoding, this authentication type is not very secure.

Secure Socket Layer Mutual Authentication

Secure Socket Layer (SSL) 3.0 and the means to perform mutual (client/server) certificate-based authentication is a J2EE Specification, v1.2 requirement. This security mechanism provides user authentication using HTTPS (HTTP over SSL).

The iPlanet Application Server 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 login screen's look and feel 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 login form action must always be `j_security_check`.

The following is an HTML sample showing how to program the form in an HTML page:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_user_name">
<input type="password" name="j_password">
</form>
```

Programmatic Login

Programmatic login enables a user to log in programmatically in both the web container and the EJB container. Programmatic login is useful for these reasons:

- It provides user authentication flexibility
- It provides an API for logout

- It is simple and extensible
- It requires fewer method calls than other types of Authentication, for example form-based, which uses an intermediate servlet
- It provides a common interface across the web and EJB containers

Form-Based vs. Programmatic Login

Suppose a web resource is deployed with security constraints using form-based authentication. To access any of these resources, the web connector must call `FormAuthServlet`, which checks if the user is already logged in. If the user is not, the login page is displayed to enable authentication.

For programmatic login, web resources are deployed without security constraints. When a user accesses a web resource, `FormAuthServlet` is not called. Instead, the `IProgrammaticLogin.login` method is called, which authenticates the user explicitly. If this method fails, an `AuthenticationException` is thrown, otherwise, the user is logged in.

The IProgrammaticLogin Interface

The `com.iplanet.ias.security.IProgrammaticLogin` interface enables a user in web or EJB container to perform login programmatically. This interface provides the following methods:

- `login`
- `logout`
- `isLoggedIn`
- `loggedUserName`

The interface is implemented by two java classes:

- The WebProgrammaticLogin Class
- The EjbProgrammaticLogin Class

Although you can create your own class that implements `IProgrammaticLogin`, this is not recommended. The provided classes save you from having to deal with the login API directly.

The WebProgrammaticLogin Class

The `com.iplanet.ias.security.WebProgrammaticLogin` class initializes the data members for programmatic login using the web container. You can use this class as is or create a subclass. Its signature is as follows:

```
public class WebProgrammaticLogin extends java.lang.Object
implements IProgrammaticLogin
```

Its one constructor is as follows:

```
public WebProgrammaticLogin(
   javax.servlet.ServletContext p_ServletContext,
   javax.servlet.http.HttpServletRequest p_HttpServletRequest,
   javax.servlet.http.HttpServletResponse p_HttpServletResponse)
throws NullValueException
```

A `com.iplanet.ias.security.NullValueException` is thrown if any of the required `WebProgrammaticLogin` input parameters are null. Its signature is as follows:

```
public class NullValueException extends java.lang.Exception
```

Its one constructor is as follows:

```
public NullValueException(java.lang.String Msg)
```

`WebProgrammaticLogin` methods are described in the following sections.

The login Method

The `login` method allows a user to log in programmatically. Its signature is as follows:

```
public void login(java.lang.String UserName, java.lang.String
Password) throws ProgAuthenticationException, NullValueException
```

The `login` method:

- makes sure the user name and password are not null
- checks if another user is logged in
- checks if `ServletContext`, `HttpRequest`, or `HttpResponse` are null
- performs the authentication

A `com.iplanet.ias.security.NullValueException` is thrown if any of the required `login` input parameters are null.

A `com.iplanet.ias.security.ProgAuthenticationException` is thrown if the authentication is unsuccessful. Its signature is as follows:

```
public class ProgAuthenticationException extends
com.netscape.server.servlet.servletrunner.AuthenticationException
```

Its one constructor is as follows:

```
public ProgAuthenticationException(java.lang.String Msg)
```

The logout Method

The `logout` method allows a user to log out. Its signature is as follows:

```
public void logout(boolean flag)
```

What `logout` does depends on the setting of `flag`:

- If `flag` is `false`, removes the principal attribute from the session (soft logout)
- If `flag` is `true`, invalidates the session (deep logout)

The isLoggedIn Method

The `isLoggedIn` method returns `true` if a user is already logged in. Its signature is as follows:

```
public boolean isLoggedIn()
```

The loggedUserName Method

The `loggedUserName` method returns the principal name of the logged user, or null if no user is logged in. Its signature is as follows:

```
public java.lang.String loggedUserName()
```

The EjbProgrammaticLogin Class

The `com.iplanet.ias.security.EjbProgrammaticLogin` class initializes the data members for programmatic login using the EJB container. You can use this class as is or create a subclass. Its signature is as follows:

```
public class EjbProgrammaticLogin extends java.lang.Object
implements IProgrammaticLogin
```

Its one constructor is as follows:

```
public EjbProgrammaticLogin() throws NullValueException
```

A `com.iplanet.ias.security.NullValueException` is thrown if the `SecurityContext` member variable is null when creation of an `EjbProgrammaticLogin` instance is attempted. Its signature is as follows:

```
public class NullValueException extends java.lang.Exception
```

Its one constructor is as follows:

```
public NullValueException(java.lang.String Msg)
```

`EjbProgrammaticLogin` methods are described in the following sections.

The login Method

The `login` method allows a user to log in programmatically. Its signature is as follows:

```
public void login(java.lang.String userName, java.lang.String
password) throws ProgAuthenticationException, NullValueException
```

The `login` method:

- makes sure the user name and password are not null
- checks if another user is logged in
- checks if `SecurityContext` is null
- performs the authentication

A `com.iplanet.ias.security.NullValueException` is thrown if any of the required `login` input parameters are null.

A `com.iplanet.ias.security.ProgAuthenticationException` is thrown if the authentication is unsuccessful. Its signature is as follows:

```
public class ProgAuthenticationException extends
com.netscape.server.servlet.servletrunner.AuthenticationException
```

Its one constructor is as follows:

```
public ProgAuthenticationException(java.lang.String Msg)
```

The logout Method

The `logout` method allows a user to log out. Its signature is as follows:

```
public void logout(boolean flag)
```

For the EJB container, this method removes the principal name of the logged user from the `SecurityContext` regardless of the `flag` value.

The isLoggedIn Method

The `isLoggedIn` method returns `true` if a user is already logged in. Its signature is as follows:

```
public boolean isLoggedIn()
```

The loggedUserName Method

The `loggedUserName` method returns the principal name of the logged user, or null if no user is logged in. Its signature is as follows:

```
public java.lang.String loggedUserName()
```

User Authorization by Servlets

Servlets can be configured to only permit access to user's with the appropriate authorization level. This is done by using the iPlanet Application Server Deployment Tool to generate DDs for the application `.ear` 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 declare all role names permitted by the application. These security roles are scoped to the J2EE web application DD.

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 declares all roles authorized to have access. The `security-rol-ref` and `role-link` elements in the web-app XML DD links the authorized roles to the application level role name.

The application assembler is responsible for linking all 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 `role-link` value element must be one of the security role names defined in a `security-role` element.

The following DD example shows how to link the security role reference to the security role.

```
<!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 permitted to the 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 DD 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 users with the appropriate authorization level. This is done by using the iPlanet Application Server Deployment Tool to generate the DD for the application `.ear` and EJB `.jar` files.

EJBs can use programmatic login just as servlets do. For more information, see “Programmatic Login,” on page 276.

Defining Roles

The deployer assigns the user groups and user accounts defined in the operational environment, to security roles defined by the application assembler.

The application assembler defines one or more roles in the DD. The application assembler then assigns the enterprise bean's home and remote interfaces method groups to the security roles to define the application’s security view.

The application assembler is responsible for defining the following:

- Each security role using a `security-role` element
- Uses the `role-name` element to define the security role name
- Optionally, can use the `description` element to provide a security role description

The security roles defined by the `security-role` elements are scoped to the `ejb-jar` file level and apply to all enterprise beans in the `ejb-jar` files. (The J2EE specification does not say a way to define global roles, that is those roles global to the container).

The following is an example of a security role definition in a DD:

```
...
<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>
```

Defining Method Permissions

The application assembler defines the method permissions relation in the DD 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 listed security roles are allowed to invoke all 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 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 DD.

```
...
<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>
...
```

There is no interaction here. The Deployment Tool converts these into security elements.

Security Role References

The bean provider is responsible for declaring in the `security-rol-ref` elements of the DD all security role names used in the enterprise bean.

The application assembler is responsible for linking all 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 `role-link` element value must be one of the security role names defined in a `security-role` element.

The following DD 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 payroll department's employees. 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>
```

User Authentication for Single Sign-on

The single sign-on across applications on the iPlanet Application Server is supported by the iPlanet Application Server servlets and JSPs. This feature allows multiple applications that require the same user sign-on information, to share this information between them, rather than having 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 the user information can be used by each individual airline site without requiring another sign on.

How to Configure for Single Sign-on

The iPlanet Application Server specific 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 Deployment Tool. This section concentrates on how the `session-info` element's security fields in the DD

work together to perform the single sign-on authentication. For details on how to create an the iPlanet Application Server specific web container DD, see the iPlanet Application Server Deployment Tool and the *Administrator's Guide*. For a complete description of all the `session-info` fields, refer to Chapter 10, "Deployment Packaging."

Table 12-1 shows the `session-info` element fields used in the authentication process:

Table 12-1 Security Fields for Single Sign-on

Field	Description
`domain`	This field specifies the domain to send back a cookie from the browser. By default (if the user does not specify a domain), the URL domain that sets the cookie is assumed to be the domain. The user can set the domain to any domain that he wishes the cookie to be sent to. The domain must have at least two periods, and sometimes may have three (for example. `.acme.com` or `.acme.co.in`).

`path`	This field specifies the session cookie's path; 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` sends the cookie back when either of the following URLs is accessed:
	`http://my.Who.com/phoenix/birds.html`
	or
	`http://my.Who.com/phoenix/bees.html`
	The path must begin with a "`/`". If the path is not set, the default path is assumed to be the URL setting the cookie.

`scope`	This field specifies a grouping name that "associates" applications sharing the same user session; that is, signing on to an application automatically allows the user to access the other applications without signing on to them. The grouped applications should have the same scope field value in their respective iPlanet Application Server specific web XML DD files.

Single Sign-on Example

Consider two applications hosted on the iPlanet Application Server named `AirlineSearch` and `AirlineBooking`. Both are part of `myairlines.com` domain and require users to be authenticated to access resources within these two applications. `AirlineSearch` allows the user to search different airlines available and `AirlineBooking` allows users to make bookings using the user's special preferences, for example, seating, menu, departure times, and so on.

The `ias-web.xml` for both `AirlineSearch` and `AirlineBooking` contains the following:

```
<session-info>
   <path>/iASApp</path>
   <scope>AirlineSignon</scope>
</session-info>
```

Now the user first accesses the services provided by the `AirlineSearch` application using the following URL:

`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 flights and 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 applications are within the same domain, the domain field is not set in this example. But this can be extended to share sign-on information between multiple domains.

User Authentication for RMI/IIOP Clients

Security on a RMI/IIOP client path is integrated into the iPlanet Application Server security infrastructure. The CXS uses the iPlanet Application Server 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 object type 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 hardcoded to three (3).

For more information on elements in the RMI/IIOP client DD, see "RMI/IIOP Client XML DTD," on page 254.

Guide to Security Information

Each information type below 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, and so on.

Location:
Directory Server

How to Create:
Create using Mission Console or programmatically using the LDAP SDK. For more information, see the iPlanet Application Server Deployment Tool Help and the *Administrator's Guide*.

Security Roles

Role that defines an application function, made up of a number of users and/or groups. LDAP groups function as roles in the iPlanet Application Server.

Location:
Directory Server

How to Create:
Use the iPlanet Application Server Deployment Tool.

How To Access:
Use `isCallerInRole()` to test for a user's role membership.

Web Server to Application Server Component Security

Beginning with iPlanet Application Server 6.0 SP2, developers can selectively encrypt the traffic between the web servers and the KXS per component. The encryption is done using 128 bit keys and RSA Bsafe3.0 library. It is recommended that developers turn on encryption judiciously for those components (servlets / JSPs) that require high security, such as credit card information gathering servlets, login servlets, and so on.

To enable encryption of the traffic between these components, you must enable the application server itself to support encryption. The steps required are:

1. Set `CCS0\\SECURITY\\EnableEncryption=D` (for Domestic 128 bit, data type `String`).
2. Create an entry or value `CCS0\\SECURITY\\LogEncryption=1` (data type `integer`). If you want to verify the encryption log messages in the KXS logs.
3. Create a key `CCS0\\EXTENSIONS\\CRYPTEXT\\CRYPTSVC\\ENGINES\\0`.
4. Re-start the web server and iPlanet Application Server.

For every component that needs encryption enabled, follow these steps:

1. Register the application using `j2eeappreg`, `webappreg`, or `iasdeploy` (recommended).
2. Set `<encrypt>true</encrypt>` in the `ias-web.xml` file for the component (servlet / JSP) that you wish to encrypt.

To verify that encryption is enabled and working fine, open the KXS logs and search for messages similar to

```
[11/Jan/2001 19:58:43:0] info: CRYPT-003: Encrypting 2309 bytes,
keysize = 128 bits

[11/Jan/2001 19:58:43:5] info: NSAPICLI-012: plugin reqstart,
tickct: 1903570535

[11/Jan/2001 19:58:43:5] info: NSAPICLI-009: plugin reqexit:
0s+.12995s. (198114 0537)

[11/Jan/2001 19:58:52:2] info: CRYPT-004: Decrypting 1897 bytes,
keysize = 128 bits
```

Chapter 13

Taking Advantage of the iPlanet Application Server Features

This chapter describes how to implement the iPlanet Application Server features in your application. The iPlanet Application Server provides many additional features to augment your servlets for use in an iPlanet Application Server 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 Microsystems standards and conforms to these future standards.

This chapter contains the following sections:

- Accessing the Servlet Engine
- Caching Servlet Results
- Using a Startup Class

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 the iPlanet Application Server is a special class called an `AppLogic`. `AppLogics` are iPlanet Application Server components that interact with the core server. In previous iPlanet Application Server releases, `AppLogics` were part of the application model, though for current and future releases they are solely available to access the iPlanet Application Server internal features.

Each servlet is scoped in an AppLogic. You can access an AppLogic instance controlling a servlet using the getAppLogic() method in the iPlanet Application Server feature interface HttpServletRequest2. When you do this, you also gain access to the server context. These activities are necessary to take advantage of other iPlanet Application Server features, as described in the following sections.

Accessing the Servlet's AppLogic

To access the controlling AppLogic, cast the request object as an HttpServletRequest2. This interface provides access to the AppLogic through the getAppLogic method, which returns a handle to the superclass.

The following example servlet header shows how to access an 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 iPlanet Application Server features require an IContext object. IContext defines a server context view. For more information, see the IContext interface section in the *Foundation Class Reference (Java)*.

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 shown 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 292, and obtain the context from the AppLogic's context member variable, as shown 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 shown 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

The iPlanet Application Server has the ability to cache a servlet's results in order to make subsequent calls to the same servlet faster. The iPlanet Application Server caches the request results (for example, a servlet's execution) for a specific amount of time. In this way, if another data call occurs the iPlanet Application Server can return the cached data instead of performing the operation again. For example, if your servlet returns a stock quote that updates every 5 minutes, you set the cache to expire after 300 seconds.

Whether to cache results and how to cache them, depends on the data type involved. For example, it makes no sense to cache the results of a quiz submission 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 iPlanet Application Server 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 iPlanet Application Server feature.

Table 13-1 shows the caching settings in a servlet configuration file.

Table 13-1 Servlet Cache Settings

Name	Type	Value
`cache-timeout`	Integer	Optional. Elapsed time (in seconds) before the servlet's memory cache is released.
`cache-size`	Integer	Optional. Servlet memory cache size (in KB).
`cache-criteria`	String	Optional. Criteria expression string containing comma-delimited descriptors. Each descriptor defines a match with one servlet input parameter.
`cache-option`	String	Optional. Sets the cache timeout option to either `TIMEOUT_CREATE` or `TIMEOUT_LASTACCESS`.

For more information on these settings, see "Elements for Specifying Servlet Caching," on page 241.

The `cache-criteria` field sets criteria to determine if servlet results are cached. This field tests one or more fields in the request. This allows conditionally cache results based on value or presence of one or more fields. If the tests succeed, the servlet results are cached.

Table 13-2 shows the `cache-criteria` field syntax.

Table 13-2 `CacheCriteria` Field

Syntax	Description
arg	Tests whether an *arg* value is in the input parameter list. For example, if the field is set to `"EmployeeCode"`, results are cached if a request contains an `"EmployeeCode"` field.
arg=v	Tests whether *arg* matches *v* (a string or numeric expression). For example, if the field is set to `"stock=NSCP"`, results are cached if the request contains a `stock` field with the value `NSCP`. Assign an asterisk (`*`) to the argument to cache a new results set when the servlet runs with a different value. For example, if the criteria is set to `"EmployeeCode=*"`, results are cached if the request object contains a field called `"EmployeeCode"` and the value is different from the currently cached value.

Table 13-2 `CacheCriteria` Field

Syntax	Description
arg=v1 | v2	Tests whether an *arg* matches a list value (*v1*, *v2*, and so on). For example: `"dept=sales|marketing|support"`.
arg=n1-n2	Test whether an *arg* number is within the given range. For example: `"salary=40000-60000"`.

Using a Startup Class

A startup class is a user-defined class object that is automatically loaded into memory when the iPlanet Application Server starts up. It performs initialization tasks within the Application Server environment. The characteristics of a `StartupClass` object are:

- It spans through the life of server in which it runs.
- It is notified when the server shuts down.
- It runs within the JVM of a `kjs` process, so each `kjs` process owns one instance of the `StartupClass` object.

A startup class must meet the following requirements:

- It must be in the package `com.iplanet.ias.startup`.
- It must be named `StartupClass`.
- It must implement the interface `com.iplanet.ias.startup.IStartupClass`.

NOTE Only one `StartupClass` object can be deployed to an Application Server instance.

The following sections describe how to create and use a startup class:

- The IStartupClass Interface
- Building the Startup Class
- Deploying the Startup Class
- How kjs Handles the StartupClass Object

The IStartupClass Interface

The `StartupClass` class must implement the `IStartupClass` interface. The `IStartupClass` interface defines two methods:

- `public void startUp() throws StartupClassException`

  This method is called to perform activities when `kjs` starts up (`kjs` calls the `StartupClass` default constructor, which calls this method). This method may perform any action. It is invoked after the `kjs` engine establishes all the relevant contexts, so it can access EJB and JDBC resources.

  If an exception occurs, this method throws a `com.iplanet.ias.startup.StartupClassException`.

- `public void shutDown()`

  This method should deallocate any resources allocated during startup.

A `com.iplanet.ias.startup.StartupClassException` is thrown if the `startUp` method fails. Its signature is as follows:

```
public class StartupClassException extends java.lang.Exception
```

Its one constructor is as follows:

```
public StartupClassException(java.lang.String msg)
```

Building the Startup Class

Building the class is supported through Ant (although using Ant is not required). Building the `StartupClass` file and any dependent java files in the *install_dir*`/startup` directory is recommended, because the files necessary for building it are there. Make sure you have done these things first:

- Include *install_dir*`/bin` in the Shell's PATH environment variable.
- Include the path to the JDK in the Shell's PATH environment variable.
- If you are not building the `StartupClass` file in the *install_dir*`/startup` directory, copy the `StartupClass.java`, `startup.properties`, and `build.xml` files from that directory into the build directory you are using.

The following are the build options:

`build compile`	Compiles all the java files in the *install_dir*`/startup` directory and places the class files under the `classes` subdirectory.
`build jar`	Runs `build compile`, then jars the class files into the `startup.jar` file and places this file under the `classes` subdirectory.
`build clean_jar`	Removes the `startup.jar` file.
`build clean`	Removes the `classes` subdirectory and its subdirectories.
`build deploy`	Deploys the `startup.jar` file to *install_dir*`/STARTUPCLASS`.
`build`	The default build, which runs `build clean`, `build compile`, `build jar`, and `build deploy` in that order.

NOTE The name of the `.jar` file must be `startup.jar`.

Deploying the Startup Class

Deployment is supported through the `iasdeploy` tool. There are two kinds of deployment:

- Local deployment

  ```
  iasdeploy deploystartup path/startup.jar
  ```

  For example:

  ```
  iasdeploy deploystartup /iasroot/ias/startup/classes/startup.jar
  ```

- Remote deployment

  ```
  iasdeploy deploystartup -host server -port port -user userName
  -password password path/startup.jar
  ```

  For example:

  ```
  iasdeploy deploystartup -host myserver -port 80 -user jjones
  -password secret /iasroot/ias/startup/classes/startup.jar
  ```

The *path* can be the following:

- The relative path to the `startup.jar` file from the directory in which `iasdeploy` is run
- The absolute path, for example *install_dir*`/startup/classes`

Concurrent deployment on multiple machines is not supported by the `iasdeploy` tool.

The `startup.jar` file is deployed to the *install_dir*`/STARTUPCLASS` directory.

NOTE	If the startup class is deployed to a directory other than *install_dir*`/STARTUPCLASS`, an informational message is generated in the `kjs` log when the application server starts up.

NOTE	The Deployment Tool does not provide support for assembling startup class modules.

How kjs Handles the StartupClass Object

For each `kjs` process that runs in its own JVM, there is one instance of the `StartupClass` object.

Inside the `run` method of `com.kivasoft.engine.Engine.java`, the `StartupClass` object is created by the system class loader after the environment is set up, but before any other method is performed. The `startUp` method is performed when the `StartupClass` object is created.

If the `startUp` method executes successfully, `kjs` runs until it shuts down normally. At graceful shutdown (with `iascontrol stop`), `kjs` calls the `shutDown` method of the `StartupClass` object.

If the `startUp` method throws a `com.iplanet.ias.StartupClassException`, `kjs` calls the `shutDown` method, and the `StartupClass` object is immediately garbage collected. Then `kjs` exits.

NOTE	Since each `kjs` process has its own copy of the `StartupClass` object, you should design a startup class with caution. It is recommended that you take care of synchronization issues for shared resources.

Appendix A

Using the Java Message Service

This appendix describes how to use the Java Message Service (JMS) API. The iPlanet Application Server allows third party JMS provider integration into its Java environment, and provides two value-added features: connection pooling and user identity mapping.

This appendix contains the following sections:

- About the JMS API
- Enabling JMS and Integrating Providers
- Using JMS in Applications
- JMS Administration
- Sample Applications
- JMS Future in the iPlanet Application Server

About the JMS API

JMS is a J2EE API which provides a standard set of Java language interfaces to an Enterprise Messaging System, often referred to as *message oriented middleware.* These interfaces are implemented by the JMS provider. The iPlanet Application Server supports the iPlanet Message Queue and the JMS provider for IBM MQ Series. For more information about the iPlanet Message Queue, refer to the following documentation:

`http://docs.iplanet.com/docs/manuals/javamq.html`

The JMS web page at `http://java.sun.com/products/jms/index.html` describes JMS's purpose 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.

The iPlanet Application Server also includes JMS Connection Pooling and User Identity Mapping. These are provided through an administrative framework and the iPlanet Application Server 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 the Connection Factories.

In order to maximize portability of an application between JMS providers, provider specific messaging aspects are encapsulated in administered objects. JMS administered objects implement one of the following four JMS interfaces, two for each messaging style:

- `Destination`
  - `Queue`
  - `Topic`
- `ConnectionFactory`
  - `QueueConnectionFactory`
  - `TopicConnectionFactory`

JMS providers supply classes that implement these interfaces. Administration tools are used to create and configure administered object class instances, and to configure them to the deployment requirements. Administrators use the tools to set provider specific parameters.

This programming model allows JMS programs to be written that are completely provider independent. Applications look up the administered objects by name using JNDI.

The following sample 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 the application did not hardcode the resource names, 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 iPlanet Application Server deployment manager does not support JMS resource references.

Enabling JMS and Integrating Providers

The iPlanet Application Server includes the software to integrate JMS providers, but it must be enabled. For information about how to integrate a JMS provider with the iPlanet Application Server, see the following documentation:

install_dir`/ias/ias-samples/jms/docs/index.html`

Using JMS in Applications

JMS support for the iPlanet Application Server is based entirely on standard J2EE APIs. Application components using the value-added features are portable with other J2EE environments. This section discusses some issues that you should consider when using JMS in applications deployed on the iPlanet Application Server.

JNDI and Application Component Deployment

JMS objects are stored by the administration tools in the JMS subcontext of the iPlanet Application Server root JNDI name space. 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 name beginning with `jms/`. Greater flexibility can be achieved by using J2EE resource references. This was demonstrated in the sample shown on page 301, where the name looked up for the factory was `java:comp/env/jms/theFactory`. In the iPlanet Application Server JMS, JMS resource references are not supported. JMS objects should be referenced directly.

Connection Factory Proxy

The iPlanet Application Server 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 proxies, which are used transparently to the application.

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 is 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 re-uses the connection.

User Identity Mapping

The `ConnectionFactoryProxy` also provides user identity mapping. JMS providers do not use the same security infrastructure as the application server and thus have different user name spaces. User identity mapping provides administrators flexibility in designing their security infrastructure.

Two mapping forms 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 iPlanet Application Server 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 proxy user name and password. 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 iPlanet Application Server proxy implementation 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 an entries list, each referenced by a unique user ID key and containing two values:

- `jmsUserName`
- `jmsPassWord`

The administrator creates the map using the `jmsuadm` tool. The entry values are used when creating a connection. For example, when an application creates a connection using the proxy with:

```
connection = proxy.createQueueConnection(userString,
passWordString);
```

The iPlanet Application Server proxy looks up the given `userString` entry in the map. If it finds an entry, the proxy passes `jmsUserName` and `jmsPassWord` values from the 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 JMS provider factory (`providerFactory`).

ConnectionFactoryProxies and Application Created Threads

A servlet can create Java threads, but it is not recommended. User created threads are not 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 results 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 crash KJS.

JMS Features Not Supported

The iPlanet Application Server does not support the JMS `XAConnection` and server session pools features described in the JMS specification.

JMS Administration

The JMS API depends on administered objects for portability. Provider specific deployment aspects are encapsulated in administered objects which allow portable application code. In the iPlanet Application Server environment JMS administration consists of four tasks:

- Creating JMS provider factories and destinations
- Creating user ID maps
- Creating `ConnectionFactoryProxies`
- Modifying the iPlanet Application Server registry connection pooling parameters

JMS Object Administration Tools

Each JMS product should include an administration program. This tool creates objects and binds them to names in the iPlanet Application Server JNDI. This section describes the Java properties and system paths required to configure a tool to work with the JMS JNDI context. Consult your provider documentation for how specific tools are configured. (A script for launching the administration tool for IBM MQ JMS for the iPlanet Application Server is described in the next section.)

Table A-1 shows the property values used to access the JMS context when creating the `InitialContext`.

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 iPlanet Application Server installation location, for 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 launches the IBM MQ JMS administration program is included in the iPlanet Application Server. It is located in `GX_ROOTDIR/jms/bin`. The administration program is a Java class. `mqjmsadm` is an interactive command line program that accepts administrator input or from an input file.

The operation is described in the MQSeries documentation for JMS Administration. `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
```

The response is:

```
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  theProviderFactorycom.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 subcontexts generate error messages.

Connection Factory Proxy Administration

Connection factory proxies are created with the `jmspadm` command (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`, 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 for proxy operation control 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 `jmsuadm` command. 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 map names.

For security purposes, the map contents cannot be listed. Administrators should protect the input files carefully.

The input file format is XML. The public name for the DTD is:

```
-//Sun Microsystems, Inc.//DTD iAS JMS User Identity Map 1.0//EN
```

The following example input file contains the two JMS users mappings:

```
<?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 all of the following three elements as noted in the above example:

- `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 iPlanet Application Server registry. If desired, these may be adjusted using the `kregedit` program in the iPlanet Application Server bin directory.

The parameters are stored in the key:

```
SOFTWARE\iPlanet\ApplicationServer\6.0\CCS0\POOLS\JMSConnectionPool
```

Table A-2 shows the parameter names and default values:

Table A-2 Parameter Names and Default Values for Connection Pooling

Parameter	Default Value	Description
`MaxPoolSize`	20	Maximum number of pooled JMS connections
`SteadyPoolSize`	10	Number of steady state connections
`MaxWait`	32 seconds	Time client waits for a connection
`UnusedMaxLife`	300 seconds	Time unused connections are deleted
`DebugLevel`	1	0-turns off logging
		1-logs callback messages
		2-logs all messages
		(see the KJS log file)
`MonitorInterval`	60	Time between messages

Connections are deleted when 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:

install_dir`/ias/ias-samples/jms`

JMS Future in the iPlanet Application Server

The following topics cover future JMS releases and how they apply to the iPlanet Application Server.

Default JMS Provider

A future J2EE standard will require that the environment include a JMS provider.

Message Driven Enterprise Java Beans

J2EE and the iPlanet Application Server do not currently support application components that receive scalable messages. A future release of J2EE will include support for “Message Driven Enterprise JavaBeans,” which are activated in response to the receipt of JMS messages. The application framework allows for scalable message receipt.

Using JMS in distributed transactions

The iPlanet Application Server does not currently support JMS resources in global transactions.

Appendix B

Dynamic Reloading

Servlets, JSPs, and EJBs can be dynamically reloaded while the server is running. This allows you to change applications without restarting the server.

Whenever a reload is done, the sessions at that transit time become invalid. The client must restart the session.

NOTE An EJB's interfaces and helper classes are not dynamically reloadable, so if you change them, you must restart the server.

If an EJB changes during a session, the EJBContainer serializes the states of the EJB instances involved in the session and deserializes them after recreating the pool of instances.

NOTE A resource such as a file that is accessed by a servlet, JSP, or EJB must be in a directory pointed to by the class loader's classpath. For example, the web class loader's classpath includes these directories:

modulename`/WEB-INF/classes`
modulename`/WEB-INf/compiled_jsp`
modulename`/WEB-INF/lib`

If a servlet accesses a resource, it must be in one of these directories or it will not be loaded properly.

Dynamic reloading is built into the server for servlets, JSPs, and EJBs. Changes made while the iPlanet Application Server is running are picked up the next time a request arrives for that servlet, JSP, or EJB.

Dynamic reloading for all classes (except unregistered JSPs, which are always reloaded) can be turned on or off using the registry entry `SYSTEM_JAVA\Versioning\Disable` (you can edit the registry using the `kregedit` tool). By default it is set to `1`, indicating that dynamic reloading is disabled. A value of `0` enables dynamic reloading.

Appendix C

Sample Deployment Files

This appendix contains sample iPlanet Application Server Deployment Descriptor (DD) files used for application and component deployment.

This appendix contains the following sample DD XML files:

- Application DD XML Files
- Web Application DD XML Files
- EJB-JAR DD XML Files
- RMI/IIOP Client DD XML Files
- Resource DD XML Files

Application DD XML Files

The application DD gives a top level view of all application contents. There are two types of application DDs; one is the J2EE application DD and the other is the iPlanet Application Server application DD. These descriptors are XML files specified by the DTDs.

The J2EE application DD is described by the J2EE specification, v2.1 Section 8.4 “J2EE:application XML DTD.” The iPlanet Application Server application DD is described by the iPlanet Application Server web application DTD described in Chapter 10, “Deployment Packaging.”

Sample Application DD XML File

This section provides an example of a J2EE application 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>
```

Web Application DD XML Files

The web application DD conveys the elements and configuration information of a web application between Developers, Assemblers, and Deployers. These descriptors are XML files specified by DTDs.

The Web application ARchive (`.war`) file contains a J2EE web application DD and an iPlanet Application Server web application DD. The J2EE web application DD is described by the Java Servlet Specification, v2.2 Chapter 13 "Deployment Descriptors." The iPlanet Application Server application DD is described by the iPlanet Application Server web application DTD described in Chapter 10, "Deployment Packaging."

Sample Web Application DD XML File

This section provides a J2EE web application DD XML file example. 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>
```

Sample iPlanet Application Server Web-App DD XML File

This section provides an example of an iPlanet Application Server web application DD XML file. The iPlanet Application Server web application DD that follows, has a file name of `ias-web.xml`.

```
<?xml version="1.0"?>
<!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 DD in the format defined by the Enterprise JavaBeans Specification, v1.1 and an iPlanet Application Server EJB DD in the format defined by Chapter 10, “Deployment Packaging.”

Sample J2EE EJB-JAR DD XML File

This section provides an example of a J2EE EJB 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.e
    <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 iPlanet Application Server EJB-JAR DD XML File

This section provides an example of an iPlanet Application Server EJB-JAR DD XML file. The following EJB-JAR DD 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>300</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>300</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>300</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>300</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>300</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>
                <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>
                <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>
```

iPlanet Application Server Client DD XML Files

The following is a sample iPlanet Application Server DD XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<ias-application-client>
    <ejb-ref>
        <ejb-ref-name>External</ejb-ref-name>
```

```
        <jndi-name>ejb/com.sun.cts.tests.appclient.deploy.ejb.ejbref.
              Test</jndi-name>
    </ejb-ref>
    <ejb-ref>
        <ejb-ref-name>External1</ejb-ref-name>
        <jndi-name>ejb/com.sun.cts.tests.appclient.deploy.ejb.ejbref.
              Test1</jndi-name>
    </ejb-ref>
</ias-application-client>
```

RMI/IIOP Client DD XML Files

The following is a sample RMI/IIOP client DD XML file.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application-client PUBLIC '-//Sun Microsystems, Inc.//DTD
J2EE Application Client 1.2//EN'
'http://java.sun.com/j2ee/dtds/application-client_1_2.dtd'>

<application-client>
    <display-name>appclient_ejb_depC_ejbref_client</display-name>
    <description>CTS appclient ejbref test</description>
    <ejb-ref>
        <ejb-ref-name>External</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>com.sun.cts.tests.appclient.deploy.ejb.ejbref.
              TestHome</home>
        <remote>com.sun.cts.tests.appclient.deploy.ejb.ejbref.
              Test</remote>
        <ejb-link>Test</ejb-link>
    </ejb-ref>
    <ejb-ref>
        <ejb-ref-name>External1</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>com.sun.cts.tests.appclient.deploy.ejb.ejbref.
              Test1Home</home>
        <remote>com.sun.cts.tests.appclient.deploy.ejb.ejbref.
              Test1</remote>
    </ejb-ref>
</application-client>
```

Resource DD XML Files

The following is a sample resource DD XML 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 iPlanet Application Server 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* and *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. See *web application*.

application event A named action that you register with the iPlanet Application Server 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 The iPlanet Application Server-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 iPlanet Application Server.

AppPath An the iPlanet Application Server 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 the iPlanet Application Server directory. (`BasePath` is also a the iPlanet Application Server 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 the iPlanet Application Server registry entry that contains the directory where the iPlanet Application Server is installed, including the iPlanet Application Server subdirectory (other iPlanet products can 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 row set 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 row set 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* and *classpath*. Normally referred to in terms of its location in the file system, 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 *class file* and *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* and *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 applications 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 iPlanet Application Server 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 (for instance, 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 multi-tiered, 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 *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 multi-user aware. 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. See *local transaction*.

group A group of users that are related in some way, maintained by a local system administrator. See *user* and *role*.

GUID 128-bit hexadecimal number, guaranteed to be globally unique, used to identify components in an iPlanet Application Server 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*.

iPlanet Application Server registry A collection of application metadata, organized in a tree, that is continually available in active memory or on a readily-accessible Directory Server.

iPlanet Application Server RowSet A `RowSet` object that incorporates the iPlanet Application Server 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. The iPlanet Application Server 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 through the network.

load balancing A technique for distributing the user load evenly among multiple servers in a cluster. 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. See *global transaction*.

memory cache An iPlanet Application Server 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 stored in a common directory. They are often 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 the iPlanet Application Server, 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 the iPlanet Application Server, 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 (for instance, 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 the iPlanet Application Server gains access to a servlet, Enterprise JavaBean (EJB), and other application resource, so named because it involves placing entries in the iPlanet Application Server 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, for instance, 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* and *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 the iPlanet Application Server, 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 life span of the client request is the same as the life span of the session bean. Session beans can be stateless or stateful, and they can be transaction aware. See *stateful session EJB*, *stateless session EJB*, and *entity EJB*.

session timeout A specified duration after which the iPlanet Application Server 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 iPlanet Application Server 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. 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 through 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 iPlanet Application Server 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*, and *global transaction*.

transaction manager Object that controls a global transaction, normally using the XA protocol. See *global transaction*.

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 (for example, HTTP, FTP), a domain (for example, `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 *group* and *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

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W