

Programmer's Guide (Java)

Netscape Application Server

Version 4.0

Netscape Communications Corporation ("Netscape") and its licensors retain all ownership rights to the software programs offered by Netscape (referred to herein as "Software") and related documentation. Use of the Software and related documentation is governed by the license agreement accompanying the Software and applicable copyright law.

Your right to copy this documentation is limited by copyright law. Making unauthorized copies, adaptations, or compilation works is prohibited and constitutes a punishable violation of the law. Netscape may revise this documentation from time to time without notice.

THIS DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. IN NO EVENT SHALL NETSCAPE BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND ARISING FROM ANY ERROR IN THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION ANY LOSS OR INTERRUPTION OF BUSINESS, PROFITS, USE, OR DATA.

The Software and documentation are copyright ©1999 Netscape Communications Corporation. All rights reserved.

Netscape, Netscape Navigator, Netscape Certificate Server, Netscape DevEdge, Netscape FastTrack Server, Netscape ONE, SuiteSpot, and the Netscape N and Ship's Wheel logos are registered trademarks of Netscape Communications Corporation in the United States and other countries. Other Netscape logos, product names, and service names are also trademarks of Netscape Communications Corporation, which may be registered in other countries. Other product and brand names are trademarks of their respective owners.

The downloading, exporting, or reexporting of Netscape software or any underlying information or technology must be in full compliance with all United States and other applicable laws and regulations. Any provision of Netscape software or documentation to the U.S. Government is with restricted rights as described in the license agreement accompanying Netscape software.



Recycled and Recyclable Paper

Version 4.0

Part Number 151-07596-00

©1999 Netscape Communications Corporation. All Rights Reserved

Printed in the United States of America. 00 99 98 5 4 3 2 1

Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, CA 94043

Contents

Preface	11
Using the Documentation	11
About This Guide	15
What You Should Already Know	16
How This Guide Is Organized	16
Documentation Conventions	17
Related Information	18
Chapter 1 Overview of NAS Applications	21
About Netscape Application Server	22
Client Tier	23
Server Tier	23
Data Tier	24
Sample Three-Tiered Applications	24
The Application Model	26
Design-Time Advantages	27
Deployment-Time Advantages	28
Functional View of the Application Model	28
The Presentation Layer	29
About Servlets	29
About JavaServer Pages	30
Application Processing	30
The Business Logic Layer	30
About EJB Functionality	31
About Session Beans and Entity Beans	31
The Data Access Layer	31
How You Create Application Components	32
About Class Libraries	33
About Interfaces	33

Chapter 2 Designing NAS Applications	35
Identifying Application Requirements	35
Defining the Requirements	36
Matching Requirements to the Application Model	36
Assembling the Development Team	37
The Architect	37
Team Roles for the Presentation Layer	38
Team Roles for the Business Logic Layer	39
Team Roles for the Data and Legacy Access Layer	41
Designing the User Interface	41
Guidelines for Effective Development	42
Easing Development	42
Maintaining or Reusing Code	43
Improving Performance	43
Planning for Scalability	44
Chapter 3 Controlling Applications with Servlets	45
Introducing Servlets	46
Servlets in NAS Applications	47
How Servlets Work	48
How Servlets Are Configured	50
Important Servlet Files and Locations	51
Deploying Servlets	52
Designing Servlets	52
Choose a Component: Servlet or JSP	53
Choose Servlet Type: HttpServlet or GenericServlet	53
Create Standard or Non-Standard Servlets	54
Planning for Code Re-Use	54
How Many Servlets for Each Interaction?	54
Creating Servlets	57
Writing the Servlet's Class File	57
Creating the Servlet's Configuration File	67
Accessing Optional NAS Features	68

Invoking Servlets	68
Calling a Servlet With a URL	69
Calling a Servlet Programmatically	70
Chapter 4 Presenting Application Pages with JavaServer Pages	73
Introducing JavaServer Pages	73
How JSPs Work	74
Designing JSPs	75
Designing for Ease of Maintenance: How Many JSPs?	75
Designing for Portability: Generic JSPs	76
Creating JSPs	76
Using Java Beans	77
Embedded Java	88
Including Other JSPs	90
Page Content Elements	95
Invoking JSPs	98
Calling a JSP With a URL	98
Invoking a JSP Programmatically	99
Chapter 5 Introducing Enterprise JavaBeans	101
What Enterprise JavaBeans Do	102
What Is an Enterprise JavaBean?	103
Understanding Client Contracts	104
Understanding Component Contracts	104
Understanding Jar File Contracts	105
Session Beans and Entity Beans	106
Understanding Session Beans	106
Understanding Entity Beans	107
The Role of EJBs in NAS Applications	107
Designing an Object-Oriented Application	108
Planning Guidelines	109
Using Session Beans	110
Using Entity Beans	110
Planning for Failover Recovery	111
Working with Databases	111

Deploying EJBs	112
Chapter 6 Using Session EJBs to Manage Business Rules	113
Introducing Session EJBs	113
Session Bean Components	115
Creating the Remote Interface	116
Creating the Class Definition	117
Creating the Home Interface	119
Additional Session Bean Guidelines	120
Creating Stateless or Stateful Beans	120
Accessing NAS Functionality	120
Serializing Handles and References	121
Managing Transactions	122
Committing a Transaction	122
Accessing Databases	123
Determining the Business Rules in Your Applications	123
Choosing a Coarse Bean Granularity	123
Chapter 7 Building Business Entity EJBs	125
Introducing Business Entity EJBs	126
How an Entity Bean Is Accessed	127
Entity Bean Components	128
Creating the Class Definition	128
Creating the Home Interface	134
Creating the Remote Interface	135
Additional Entity Bean Guidelines	136
Accessing NAS Functionality	136
Serializing Handles and References	137
Managing Transactions	137
Committing a Transaction	138
Handling Concurrent Access	138
Accessing Databases	140
Determining the Business Objects in Your Applications	140
Choosing a Coarse Bean Granularity	140

Chapter 8 Handling Transactions with EJBs	143
Understanding the Transaction Model	144
Specifying Transaction Attributes in an EJB	145
Using Bean Managed Transactions	146
Restrictions on Transaction Attributes	146
Setting Isolation Levels	147
Chapter 9 Using JDBC for Database Access	149
Introducing JDBC	150
Supported Functionality	151
Understanding Database Limitations	151
Understanding NAS Limitations	152
Supported Databases	154
Using JDBC in Server Applications	155
Using JDBC in EJBs	155
Using JDBC in Servlets	157
Handling Connections	158
Local Connections	158
Global Connections	159
Working with JDBC Features	161
Working With Connections	162
Pooling Connections	163
Working with ResultSet	163
Working with ResultSetMetaData	165
Working with PreparedStatement	165
Working with CallableStatement	166
Handling Batch Updates	166
Creating Distributed Transactions	167
Working with Rowsets	169
Using JNDI	171
Chapter 10 Creating Configuration Files	173
Introducing Configuration Files	173
The NAS Registry	174
GUIDs	174

Creating Servlet and Application Configuration Files	175
NTV Syntax	175
Manually Registering Servlets to NAS	176
Application Configuration Information	177
Servlet Configuration Information	181
Creating EJB Property Files	191
Introducing Bean Property Files	191
Declaring a Deployment Descriptor	192
Declaring a Session Bean Descriptor	194
Declaring an Entity Bean Descriptor	196
Declaring Control Descriptors	197
Creating Access Control Entries	202
Example Bean Property File	205
Manually Editing Property Files	207
Creating Datasource Property Files	208
Datasource Property File Entries	209
Manually Registering Datasources to NAS	209
Chapter 11 Creating and Managing User Sessions	211
Introducing Sessions	211
Sessions and Cookies	212
Sessions and Security	212
How to Use Sessions	213
Creating or Accessing a Session	213
Examining Session Properties	214
Binding Data to a Session	216
Invalidating a Session	216
Controlling the Type of Session	217
Sharing Sessions with AppLogics	218
Chapter 12 Writing Secure Applications	219
Understanding the Security Model	220
Security Concepts	221
Programmatic Security	223
Declarative Security	224

Groups and Roles	225
Security and Cookies	225
Guide to Security Information	226
Authenticating a User	228
Logging In to the Session	229
Checking Identity	229
Checking Permission	232
Logging Out of the Session	232
Access Control Lists	233
Creating and Using Declarative ACLs	233
Creating and Using Programmatic ACLs	234
Chapter 13 Taking Advantage of NAS Features	235
Accessing the Servlet Engine	236
Accessing the Servlet's AppLogic	236
Accessing the Server Context	237
Caching Servlet Results	237
Using Application Events	239
The Application Events API	240
Creating a New Application Event	241
Sending and Receiving Email from NAS	243
Accessing the Controlling AppLogic	243
Receiving Email	244
Sending Email	246
Netscape Application Builder Features	248
Validating Form Field Data	248
Creating Named Form Action Handlers	253
Example Validation and Form Action Handler	254
Chapter 14 Inside the Online Bookstore Sample Application ..	259
Design Overview	259
Application Model	260
Application Flow	260

Presentation Flow	262
Directory Structure	266
Servlets	267
EJB Functionality	268
Appendix A Summary of Standard APIs	271
The Java Servlet API	272
The javax.servlet Package	272
The javax.servlet.http Package	272
The Enterprise JavaBeans API	273
The javax.ejb Package	273
The javax.ejb.deployment Package	273
The JDBC Core API	274
The JDBC Standard Extensions API	276
Appendix B Dynamic Reloading	277
How Dynamic Reloading Works	278
Summary of Related Registry Entries	278
Glossary	281
Index	295

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

This preface contains the following sections:

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

Using the Documentation

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

Note that the printed manuals are also available as online files in PDF and HTML format. See the page `installdir/nas/docs/index.htm`, where `installdir` is the directory in which you installed NAS.

For information about	See the following	Shipped with
Late-breaking information about the software and the documentation	<code>readme.htm</code>	NAS 4.0 Developer Edition (Solaris), NAS 4.0, NAB 4.0
Installing Netscape Application Server and its various components (Web Connector plug-in, Netscape Application Server Administrator), and configuring the sample applications	Installation Guide	NAS 4.0 Developer Edition (Solaris), NAS 4.0
Installing Netscape Application Builder	<code>install.htm</code>	NAB 4.0
Basic features of NAS, such as its software components, general capabilities, and system architecture	Overview	NAS 4.0 Developer Edition (Solaris), NAS 4.0, NAB 4.0
Deploying Netscape Application Server at your site, by performing the following tasks: <ul style="list-style-type: none"> • Planning your Netscape Application Server environment • Integrating the product within your existing enterprise and network topology • Developing server capacity and performance goals • Running stress tests to measure server performance • Fine-tuning the server to improve performance 	Deployment Guide	NAS 4.0

For information about	See the following	Shipped with
Administering one or more application servers using the Netscape Application Server Administrator tool to perform the following tasks: <ul style="list-style-type: none"> • Deploying applications with the Deployment Manager tool • Monitoring and logging server activity • Setting up users and groups • Administering database connectivity • Administering transactions • Load balancing servers • Managing distributed data synchronization 	Administration Guide	NAS 4.0
Migrating your applications to the new Netscape Application Server 4.0 programming model from version 2.1, including a sample migration of an Online Bank application provided with Netscape Application Server	Migration Guide	NAS 4.0 Developer Edition (Solaris), NAS 4.0, NAB 4.0

For information about	See the following	Shipped with
<p>Creating NAS 4.0 applications within an integrated development environment by performing the following tasks:</p> <ul style="list-style-type: none"> • Creating and managing projects • Using wizards • Creating data-access logic • Creating presentation logic and layout • Creating business logic • Compiling, testing, and debugging applications • Deploying and downloading applications • Working with source control • Using third-party tools 	User's Guide	NAB 4.0
<p>Creating NAS 4.0 applications that follow the new open Java standards model (Servlets, EJBs, JSPs, and JDBC), by performing the following tasks:</p> <ul style="list-style-type: none"> • Creating the presentation and execution layers of an application • Placing discrete pieces of business logic and entities into Enterprise Java Bean (EJB) components • Using JDBC to communicate with databases • Using iterative testing, debugging, and application fine-tuning procedures to generate applications that execute correctly and quickly 	Programmer's Guide (Java)	NAS 4.0 Developer Edition (Solaris), NAB 4.0

For information about	See the following	Shipped with
Using the public classes and interfaces, and their methods in the Netscape Application Server class library to write Java applications	Server Foundation Class Reference (Java)	NAS 4.0 Developer Edition (Solaris), NAB 4.0
Creating NAS C++ applications using the NAS class library by performing the following tasks: <ul style="list-style-type: none"> • Designing applications • Writing AppLogics • Creating HTML templates • Creating queries • Running and debugging applications 	Programmer's Guide (C++)	Order separately
Using the public classes and interfaces, and their methods in the Netscape Application Server class library to write C++ applications	Server Foundation Class Reference (C++)	Order separately

About This Guide

This guide describes how to create applications intended to run on Netscape Application Server.

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

What You Should Already Know

This guide assumes you are familiar with the following topics:

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

How This Guide Is Organized

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

The first part provides an overview of the Netscape Application Server programming model and environment. This part includes the following topics:

- Chapter 1, “Overview of NAS Applications”
- Chapter 2, “Designing NAS Applications”

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

- Chapter 3, “Controlling Applications with Servlets”
- Chapter 4, “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 5, “Introducing Enterprise JavaBeans”
- Chapter 6, “Using Session EJBs to Manage Business Rules”

- Chapter 7, “Building Business Entity EJBs”
- Chapter 8, “Handling Transactions with EJBs”
- Chapter 9, “Using JDBC for Database Access”

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

- Chapter 10, “Creating Configuration Files”
- Chapter 11, “Creating and Managing User Sessions”
- Chapter 12, “Writing Secure Applications”
- Chapter 13, “Taking Advantage of NAS Features”
- Chapter 14, “Inside the Online Bookstore Sample Application”

The appendixes include the following reference material:

- Appendix A, “Summary of Standard APIs”
- Appendix B, “Dynamic Reloading”

Finally, a Glossary and Index are provided.

Documentation Conventions

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

This guide uses URLs of the form:

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

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

This guide uses the following font conventions:

- The monospace font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.

- *Italic* type is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.

Related Information

Specifications related to the NAS programming model are provided in the directory *installdir/nas/docs/javaspec*, where *installdir* refers to the directory in which you installed NAS. You can find a directory of all NAS-related documentation at *installdir/nas/docs/index.htm*.

The official specifications are maintained at the following URLs. Note that these sites do not necessarily contain the versions of these specifications that are supported by NAS. See the index listed above for links to specification versions supported by NAS.

Servlets	http://java.sun.com/products/servlet
JavaServer Pages (JSPs)	http://java.sun.com/products/jsp
Enterprise JavaBeans (EJBs)	http://java.sun.com/products/ejb
Java Naming and Directory Interface (JNDI)	http://java.sun.com/products/jndi
Java Database Connectivity (JDBC)	http://java.sun.com/products/jdbc

Additionally, we recommend the following resources:

Programming with Servlets and JSPs

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

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

The web site <http://www.servletcentral.com>

Programming with EJBs

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

The web site <http://ejbhome.iona.com>

Programming with JDBC

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

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

Overview of NAS Applications

This chapter introduces Netscape Application Server and key concepts for developing, deploying, and managing web applications.

The following topics are included in this chapter:

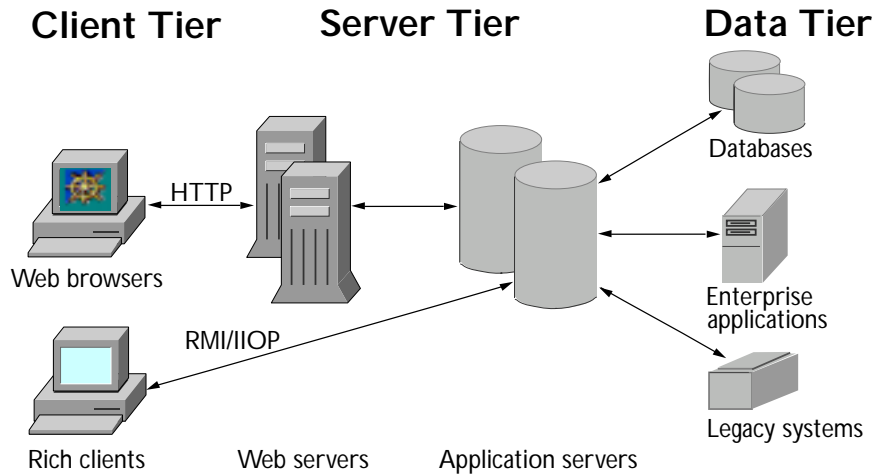
- About Netscape Application Server
- Sample Three-Tiered Applications
- The Application Model
- The Presentation Layer
- The Business Logic Layer
- The Data Access Layer
- How You Create Application Components

About Netscape Application Server

Netscape Application Server is the middleware between enterprise data sources and the clients that access these data sources. Business code is stored and processed on Netscape Application Server (NAS) rather than on clients. An application is deployed and managed in a single location, and the application is accessible to large numbers of heterogeneous clients.

NAS applications run in a distributed, multitiered environment. This means that an enterprise system might consist of several application servers (computers running the Netscape Application Server software), along with multiple database servers and web servers. Your application code can be distributed among the application servers.

Overall, the machines and software involved are divided into a client tier, a server tier, and a data tier, as shown in the following figure:



Client Tier

The client tier is the user interface. End users interact with client software to use the application. For the current release of NAS, the client software is assumed to be a web browser (such as Netscape Navigator). In future releases, NAS will also support end-user access through rich clients running RMI over IIOP for requests originating from CORBA clients.

Server Tier

The server tier consists of at least one web server and at least one application server. Netscape Application Server handles requests from clients by running the appropriate application code, then returns the results to the clients over the Internet or over an intranet. For web clients, a web server stands between the clients and the application server, passing HTTP requests and responses back and forth among the servers and clients.

The application server runs the NAS software as well as your server-side application code. You write this portion of the application code using standard Java APIs and, optionally, NAS feature APIs.

The standard Java APIs include the following:

- Servlet API
- Enterprise JavaBeans API
- Java Database Connectivity API

Sun Microsystems drives both the development of these APIs and the publication of the API specifications. These processes include participation from key vendors of Java technology and from the general public. As a result, these APIs are emerging as an industry standard. All specifications are accessible from `installdir/nas/docs/index.htm`, where `installdir` is the location in which you installed NAS.

In addition to using the standard APIs, you can also develop custom NAS features or provide backward-compatibility to existing NAS applications. To do so, use the Netscape Application Server Foundation Class Library. For more information about this library, see the *Netscape Application Server Foundation Class Reference*.

Data Tier

The data tier consists of one or more enterprise data sources, such as

- mainframe or other legacy systems
- relational databases
- LDAP servers
- client/server applications; for example, Enterprise Resource Planning (ERP) applications such as SAP R/3

The data tier stores the data that forms the basis of an application. For example, a relational database may be used to keep track of customer information, account information, and inventory. Code in the data tier can be created using a variety of database applications, SQL tools, or other third-party applications. Describing how to write this code is outside the scope of this guide.

As a developer of NAS applications, you must create the business logic that accesses the data tier. To do so, you typically use the Java Database Connectivity (JDBC) API.

Sample Three-Tiered Applications

The Online Bank sample application, which is shipped with Netscape Application Server, provides online banking and customer management. Online Bank is a good example of a three-tiered Netscape Application Server application. The following steps describe a typical user session with this application.

1. The user logs in.

To begin working with the application, the user connects to the web through a browser and requests the URL of the application's first page, the login page.

2. The server responds.

After typing a user ID and password, the user clicks the Login button. This causes an action request to be sent to the web server, which forwards it to NAS. NAS handles requests by running the appropriate application code.

In this case, the code needed is that which logs in the user. To validate the user's password, the application accesses the database tier, where information about users is stored. For example, this information might be stored in an LDAP server or in a relational database.

3. The server returns results.

After looking up the user's ID and password in the database, the application can validate the user and send a response back to the client tier. The response is a web page containing the main menu of the application; the page is sent to the user's web browser (assuming the user was validated successfully).

4. The user continues using the application.

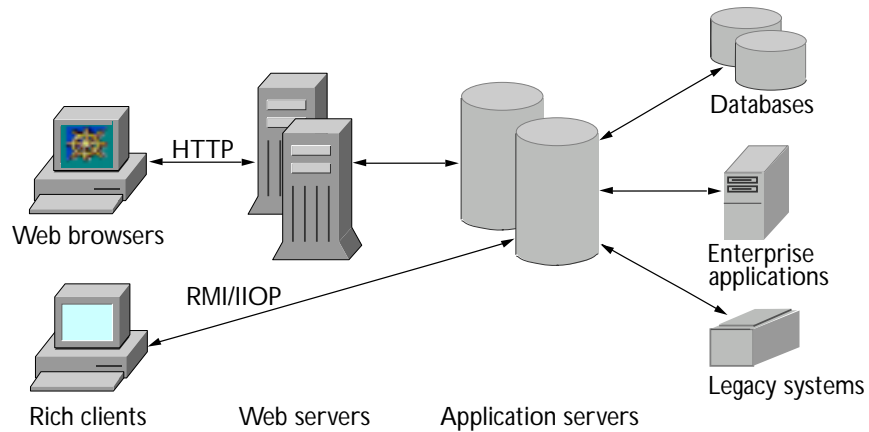
The user might next ask for the current balance in a checking account. NAS handles this request by querying the database, formatting the data into a report, and sending the report back to the user.

This cycle of request, query, and response is repeated many times as the customer makes other selections and navigates through the application. Requests come in from the client tier and are processed in the server tier, often by accessing the database tier. Responses are then sent back from the server to the client.

Note An additional sample three-tiered application based on an online bookstore is described in detail in Chapter 14, "Inside the Online Bookstore Sample Application."

The Application Model

An application model is the conceptual division of a software application into functional components. The application model for Netscape Application Server is shown in the following figure:



The application model consists of three layers:

- presentation
- business logic
- data and legacy access

Each layer is further described in separate sections of this chapter.

The application model is based on Java standards. By implementing these widely adopted standards, the NAS application model makes development more component oriented and promotes code reuse.

The following table lists the standards-based application components associated with each area of the application model:

Functional area	Application component
Presentation logic and layout	Java servlets for logic; JavaServer Pages (JSPs) for layout.
Business logic	Enterprise JavaBeans
Data and legacy access	Enterprise JavaBeans that use either JDBC interfaces, distributed transactions, or both.

A **Java servlet** is an application component—a Java object that is an instance of the `Servlet` class. The `Servlet` class is among the code defined in the Servlet API specification.

A **JavaServer Page** (JSP) is an application component used for presenting dynamically generated content. JSPs are text files written in a combination of standard HTML tags, JSP tags, and Java code.

Enterprise JavaBeans is a component model for developing and deploying Java applications in a multitiered, distributed architecture. The components themselves are referred to as Enterprise JavaBeans (EJBs).

The **JDBC API** is a standards-based set of classes and interfaces that enable developers to create data-aware EJBs.

Design-Time Advantages

By dividing an application into functional areas, developers more clearly understand how to design and build an application and what tools to use. In addition, the development cycle is more efficient because each type of developer—web master, database expert, Java programmer, user interface designer, and so on—can focus on distinct areas of the application.

Furthermore, dividing presentation functionality into a separate layer makes the application more flexible. For example, when you want to change the presentation style of an e-commerce application, you can do so without changing the logic in the business layer or data access layer.

Deployment-Time Advantages

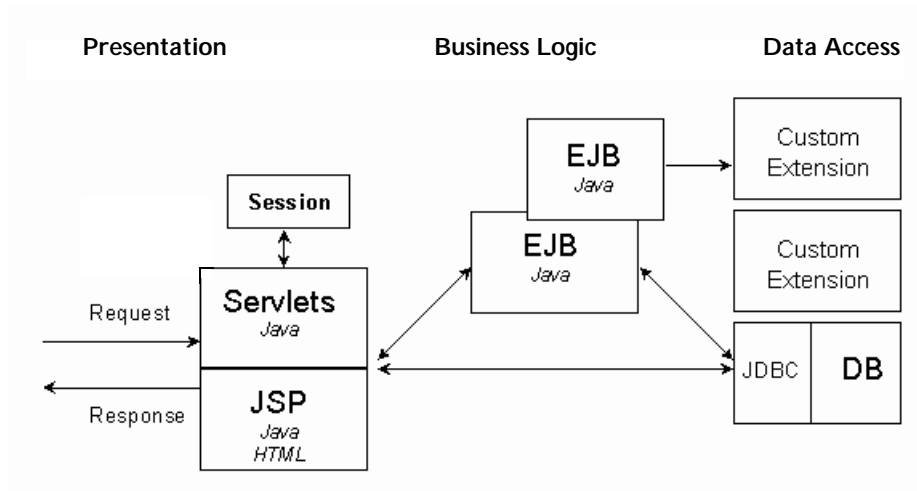
The division of an application is conceptual. It occurs in the design of the application, not necessarily in its deployment. However, the way you design an application can make different deployment options possible.

For example, you can deploy an application to run on one computer, or you can scale the application to run on a cluster of servers. Each server can be dedicated to a separate application task, or the servers can work redundantly to allow for load balancing or failover recovery.

In summary, the application model makes it possible to deploy a fully distributed, network-oriented application.

Functional View of the Application Model

The following figure shows typical interactions between these layers.



The Presentation Layer

The presentation layer offers a user interface to the application's end user. This layer determines the application's "look and feel".

In the NAS application model, presentation is controlled by the presentation layer. This layer separates business logic from presentation on the client.

The presentation layer consists of two main functional areas:

- presentation logic, which is implemented using Java servlets
- presentation layout, which is implemented using JSPs

Presentation logic handles such tasks as page-to-page navigation, session management, simple input validation, and the tying together of business logic.

Presentation layout involves issues typical to web page design, such as creating HTML and images that load efficiently, adhering to corporate design standards, and deciding what buttons or menus will be available on particular pages.

In addition to supporting the standard Java servlets and JavaServer Pages, the presentation layer supports validation of incoming requests, as well as interaction with other application components (for instance, Enterprise JavaBeans).

About Servlets

Java servlets provide an application's presentation logic. A servlet is an application component that runs inside a server. A servlet is an instance of the `Servlet` class. Servlets are similar to applets, which are client-side Java components.

Java servlets handle such tasks as request processing, session management, and the tying together of business logic. Servlets can make calls to JSPs, Enterprise JavaBeans, and JDBC `RowSet` objects.

About JavaServer Pages

JavaServer Pages (JSPs) are used for output processing. A JSP is an HTML file containing a mix of HTML, special JSP tags, and Java code. JSPs act as output templates, separating the presentation of dynamic content from the generation of that content. As a result, developers can code most output to be delivered by HTML files instead of having to issue cumbersome print statements with HTML code in them.

Application Processing

NAS applications consist of a set of Java classes that define the servlets, and a set of JSP files that define output responses. There is an API for expanding templates from a servlet. An application in NAS 4.0 also needs several configuration files for initializing the servlet engine with information about the application.

The application model is based on the following:

- requests containing input information from the user
- objects (servlets) that receive the requests and process the input
- streams for communicating back to the user via the browser (using HTTP)
- sessions for maintaining state between requests

The Business Logic Layer

The business logic layer maintains the application-specific processing and business rules of an application. The application components in the business logic layer connect those in the presentation and data layers. These components define an application's processing logic, processing flow, and connections to back-end data sources.

About EJB Functionality

In the NAS application model, the business layer is implemented using Enterprise JavaBeans. Enterprise JavaBeans support the following functionality:

- transactions
- security
- remote access
- threading
- naming
- resource pooling
- persistence

About Session Beans and Entity Beans

There are two types of Enterprise JavaBeans: session beans and entity beans.

Session beans relate to units of work, such as a request for data. Session beans are short lived—the lifespan of the client request is the same as the lifespan of the session bean. Session beans can be stateless or stateful, and they can be transaction aware.

Entity beans relate to physical data, such as a row in a database. Entity beans are long-lived, because they are tied to persistent data. Entity beans are always transactional and multiuser aware.

The Data Access Layer

The data access layer provides Enterprise JavaBeans with access to back-end data sources. In the NAS application model, Enterprise JavaBeans access relational databases using JDBC APIs.

To access other data sources, your code may need to make a call into an extension. Netscape Application Server includes prebuilt extensions. These extensions allow an application to communicate with transaction-processing systems such as CICS/IMS, IBM MQSeries, and BEA Tuxedo. Additional prebuilt

extensions allow communication with Enterprise Resource Planning (ERP) systems such as SAP R/3. If you need to create a custom extension, use Netscape Extension Builder.

As part of the EJB standard, NAS supports distributed transactions. This support means that NAS applications can ensure data integrity because the server manages transactions across multiple back-end databases.

How You Create Application Components

To create application components, you can use basic tools. For example, you can use a text editor to create Java source code or JSPs. More likely, you will want to use a visual editor or an integrated development environment (IDE) such as Netscape Application Builder.

As you develop NAS applications, you create Java files that use a number of APIs:

- Java Servlet API
- Enterprise JavaBeans API
- JDBC API

These APIs are listed in Appendix A, “Summary of Standard APIs.” Because Sun owns the process of publishing these APIs, the appendix also includes web links to more information.

Along with the standard APIs, applications can use the API that ships with NAS. This API provides:

- added functionality not available through the standard APIs
- functionality that optimizes applications for NAS
- integration with application components that were designed for earlier versions of NAS

The NAS API is more formally known as the Netscape Application Server Foundation Class Library and is summarized in the following sections.

About Class Libraries

A library is a set of predefined interfaces and class declarations that can be used in object-oriented programs. The Netscape Application Server Foundation Class Library contains interfaces and classes for developing server-side code.

The Netscape Application Server Foundation Class Library is designed for building server-side code as part of NAS applications. The class library is stored in several packages in the Netscape Application Server installation directory and is copied to your disk when you install Netscape Application Server.

The classes and interfaces in the Netscape Application Server Foundation Class Library define many types of objects you can include in Netscape Application Server applications. Each object provides a specific type of functionality that is commonly needed. The following list shows some of the major types of objects and functionality you can include in your application by using the Netscape Application Server Foundation Class Library:

- AppLogic objects
- data connections
- queries and other database commands
- dynamic reports
- electronic mailboxes
- user sessions and session-related data
- security

This is just a partial list. For complete information about the Netscape Application Server Foundation Class Library, see the *Netscape Application Server Foundation Class Reference*.

About Interfaces

The Netscape Application Server programming API is based on interfaces. This section provides an overview of the concepts involved in such a programming model. You do not need to read this section if you are already familiar with these concepts.

What Is an Interface?

The objects in Netscape Application Server applications interact through *interfaces*. An interface is a description of the services provided by an object. An interface is like a contract between an object and its user (the code that wants to interact with it). The contract describes a set of expected behavior. Code that wants to use an object need only know what interface the object supports; code does not need to know anything about the internal implementation of the object.

An interface defines a set of functions, called methods. The interface has no implementation code. It describes only the parameters and return types of its methods. The code for the methods is written separately, in a class, which is said to *implement* the interface. Typically, the interfaces and their implementations are written by different groups of people.

How is an interface different from a class? A class is a set of data and functions (member variables and methods) that define the characteristics of one type of object. An object is an instantiation of a class. 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 can be instantiated to form an object, but an interface can not be instantiated, because it has no implementation code to determine what to do when each method is called.

Every interface has a name that serves as an identifier you can refer to in code. By convention, the name of each interface begins with a capital I, for instance, `IBuffer`.

Benefits of Using Interfaces

Interfaces provide a level of abstraction that enables objects to interoperate more easily. The code that will use the object needs to know how to connect to the object and call its methods. The object needs to expose its services to any code that wants to connect. The interface provides the connection that not only allows the code to access the object, but also allows the object to expose its services.

When using an interface-based programming model, you can modify the internal implementation of an object whenever you need to. As long as the object continues to implement the same interface, the code that uses that object does not require rewriting or recompilation.

Designing NAS Applications

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

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

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

Identifying Application Requirements

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

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

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

Defining the Requirements

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

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

Matching Requirements to the Application Model

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

In the presentation layer, the application may need to present one set of pages for anonymous users, and another set for registered users. Also, when an anonymous user tries to access a feature reserved for registered users, you

might design the application to present a page that explains why the attempt failed, and invite the user to become a member. By the same token, premier customers might have access to some pages that are denied to regular customers.

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

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

Assembling the Development Team

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

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

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

The Architect

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

Team Roles for the Presentation Layer

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

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

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

Java Servlet Developers

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

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

JSP Developers

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

JSPs are likely to call EJBs and JDBC RowSet objects.

HTML Designers

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

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

Graphic Artists

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

Client-Side JavaScript Developers

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

Team Roles for the Business Logic Layer

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

- Session bean developers
- Entity bean developers

Session Bean Developers

Session beans encapsulate the logic for business processes and business rules. For example, a session bean might be developed to calculate taxes for a billing invoice. Applications usually handle complex business rules that can change

frequently (for example, due to new business practices or new government regulations). As a result, an application typically uses more session beans than entity beans, and session beans may need continual revision.

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

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

Entity Bean Developers

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

As a result, an entity bean developer typically understands issues such as caching, coherency, indexing, database performance and transactions, and object-relational mapping. An entity bean developer is similar to the schema developer in a relational database environment.

Entity beans are less likely than session beans to require changes, because the organization of enterprise data is usually a mature process, so database schema tend to be stable. Besides, frequent changes to an organization's data model incur a high cost for redevelopment and retesting.

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

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

Team Roles for the Data and Legacy Access Layer

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

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

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

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

Designing the User Interface

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

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

- What is the page flow?
- What commands and buttons are available on each page?
- Will the pages use frames or not?

- Are there any corporate standards that determine headers and footers, logos, menu bars, or banner ads?
- At what point is login required?
- Are there any international issues? For example, are the icons or cartoon images meaningful to all users? Does translation pose a formatting problem?
- Are the commonly used features easy to find?

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

Guidelines for Effective Development

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

The guidelines are grouped into the following goals:

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

Easing Development

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

Maintaining or Reusing Code

- Choose base classes, helper classes, and helper methods in a way that encourages code reusability.
- Use relative paths and URLs, so that links remain valid if you move the code tree later.
- Minimize the Java in your JSPs; instead, put Java in servlets and helper classes. JSP designers can revise JSPs without being Java experts.
- Use property files or global classes to store hardcoded strings such as the names of datasources, tables, columns, JNDI objects, or other application properties.
- Use session beans, rather than servlets and JSPs, to store business rules that are domain-specific or likely to change often, such as input validation.
- Use entity beans for persistent objects; using entity beans allows management of multiple beans per user.
- For maximum flexibility, use Java interfaces rather than Java classes.
- Use extensions to access legacy data.

Improving Performance

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

Planning for Scalability

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

Controlling Applications with Servlets

This chapter describes how to create effective Java servlets to control interactions in Netscape Application Server applications.

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

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

This chapter contains the following sections:

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

Introducing Servlets

NAS servlets are based on the Servlet API specification. All specifications are accessible from `install_dir/nas/docs/index.htm`, where `install_dir` is the location in which you installed NAS.

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

The fundamental characteristics of servlets are as follows:

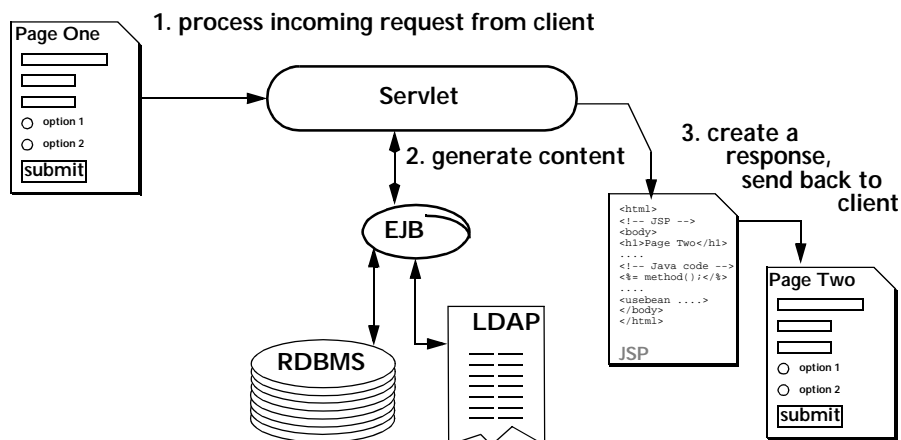
- Servlets are created and managed at run time by the servlet engine, a component of NAS that runs inside the Java server.
- The input data on which servlets operate is encapsulated in an object called the request object. A servlet's response to a query is encapsulated in an object called the response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

NAS offers several features through NAS-specific APIs that enable your applications to take programmatic advantage of specific features of the NAS environment. For more information, see "Accessing Optional NAS Features" on page 68.

Servlets in NAS Applications

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

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



1. Process incoming request from client

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

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

2. **Generate content**

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

3. **Create a response**

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

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

How Servlets Work

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

Types of Servlets

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

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

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

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

Instantiating and Removing Servlets

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

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

How Servlets Handle Requests

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

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

Note The automatic routing in an HTTP servlet is based simply on a call to `request.getMethod()`, which provides the HTTP transfer method. Since request data is already preprocessed into a name-value list in NAS, you could

simply override the `service()` method in an HTTP servlet without losing any functionality. However, this does make the servlet less portable, since it is now dependent on preprocessed request data.

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

How the Servlet Engine Allocates Resources

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

If a servlet is specifically written as single-threaded, the servlet engine creates a pool of ten (10) instances of the servlet to be used for incoming requests. If a request arrives when all instances are busy, it is queued until an instance becomes available.

For tips on threading issues, see “Handling Threading Issues” on page 63.

Run-Time Servlet Modifications

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

How Servlets Are Configured

Servlet configuration refers to the servlet’s metadata, information about the servlet that enables the server to create and use it in the framework of an application. There is currently no standard for servlet configuration.

NAS provides a configuration method whereby an ASCII file stores the metadata needed by the server in a name-type-value (NTV) format. This file is loaded into NAS when the servlet is deployed. The information can also be updated at run-time by the Administration Tool (see the *Administration Guide*).

Servlets can be configured as part of a specific application, or as generic components used by many applications. NAS considers servlets that are not part of a specific application to be members of the generic application, which means simply that components in the generic application are globally available.

For more information about servlet configuration, see Chapter 10, “Creating Configuration Files.”

Important Servlet Files and Locations

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

`AppPath` and `BasePath` are variables held in the NAS registry, a repository for server and application metadata. See “The NAS Registry” in Chapter 10, “Creating Configuration Files” and the *Administration Guide*.

The following table describes important files and locations for servlets:

Location	Description
<code>BasePath</code>	Top of the NAS tree. All files in this directory are part of NAS. Defined by the NAS registry variable <code>BasePath</code> .
<code>AppPath</code>	Top of the application tree. Files in this directory are part of the generic application. Applications reside in subdirectories of this location. Defined by the NAS registry variable <code>AppPath</code> .
<code>AppPath/ntv</code>	Subdirectory that contains application configuration file and servlet configuration files for the generic application.
<code>AppPath/ntv/appInfo.ntv</code>	Application configuration file for the generic application. Contains pointers to generic-application servlet configuration files in the same directory.

Location	Description
<code>AppPath/ntv/ servletInfo.n tv</code>	Servlet configuration file for one or more servlets in the generic application. May be named anything as long as it is referenced in <code>appInfo.ntv</code> in the same directory.
<code>AppPath/ appName/*</code>	Top of the subtree for the application <code>appName</code> . Files in this directory are part of the specific application <code>appName</code> . This corresponds to the URL used to access the application's components; see “Invoking Servlets” on page 68.
<code>AppPath/ appName/ntv/*</code>	Subdirectory that contains application configuration file and servlet configuration files for the specific application <code>appName</code> .
<code>AppPath/ appName/ntv/ appInfo.ntv</code>	Application configuration file for the specific application <code>appName</code> . Contains pointers to application-specific servlet configuration files in the same directory.
<code>AppPath/ appName/ntv/ servletInfo.n tv</code>	Servlet configuration file for one or more servlets in the generic application. May be named anything as long as it is referenced in <code>appInfo.ntv</code> in the same directory.

Deploying Servlets

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

For more information, see Chapter 2, “Deploying and Upgrading Applications” in the *Administration Guide*.

Designing Servlets

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

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

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

Choose a Component: Servlet or JSP

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

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

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

For more information on JSPs, see Chapter 4, “Presenting Application Pages with JavaServer Pages.”

Choose Servlet Type: `HttpServlet` or `GenericServlet`

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

For more information, see “Types of Servlets” on page 48.

Create Standard or Non-Standard Servlets

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

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

For more information on NAS-specific APIs, see “Accessing Optional NAS Features” on page 68.

Planning for Code Re-Use

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

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

For more information, see “How Servlets Are Configured” on page 50.

How Many Servlets for Each Interaction?

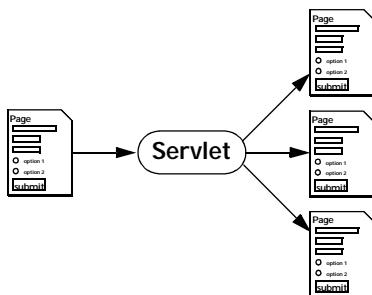
There are several possibilities for programming a given user interaction with a servlet. The method you choose for a given interaction determines how you write the servlet that handles the interaction.

One servlet handles one request

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

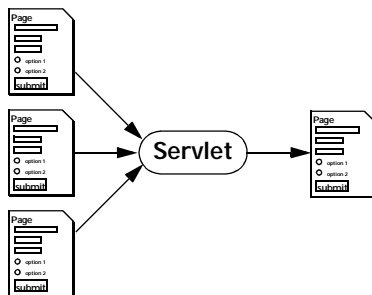


A servlet can tailor the output by setting conditions. For example, a login page could lead to a welcome page for users, an administrative page for site managers, and an error page for incorrect login, as in the following diagram:



One servlet handles multiple requests

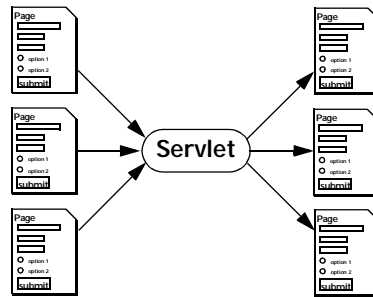
You can write a single servlet to handle many similar (or even dissimilar) requests by setting a conditional flag for each request, as in the following diagram:



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

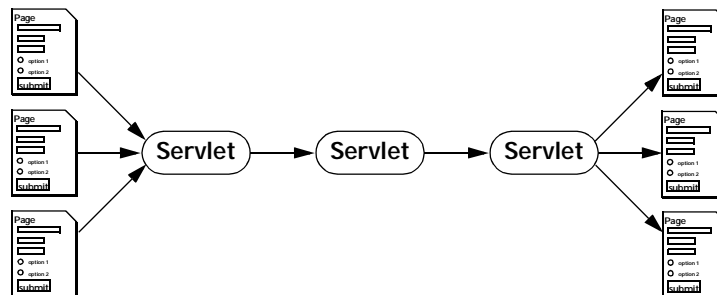
```
flag = request.getParameter("flag");
if ( flag.equals("step1")) {
    process_step1;
}
else if ( flag.equals("step2")){
    process_step2;
}
```

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



Several servlets handle one request

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



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

Creating Servlets

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

- Design the servlet into your application, or, if you want it to be accessed in a generic way, design it so that it accesses no application data.
- Create a class that extends either `GenericServlet` or `HttpServlet`, overriding the appropriate methods so that they handle requests.
- Write a servlet configuration file, or assign your servlet to an existing configuration file.
- If your servlet is part of an application, write an application configuration file that includes your servlet configuration file. If your servlet stands alone (i.e., is part of the generic application), add its configuration file to the application configuration file for the generic application.

For a description of design considerations, see “Designing Servlets” on page 52. For information about creating the files that make up a servlet, see the following sections:

- Writing the Servlet’s Class File
- Creating the Servlet’s Configuration File
- Accessing Optional NAS Features

Writing the Servlet’s Class File

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

Creating the Class Declaration

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

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

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

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

Overriding Methods

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

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

Business logic may involve authenticating the application's user name and password, accessing a database to perform a transaction, or passing the request to an EJB.

Overriding `init`

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

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

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

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

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

Now other methods in the servlet can access this variable.

Overriding destroy

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

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

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

Overriding service, doGet, and doPost

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

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

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

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

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

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

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

Note If you have a servlet that you intend to also call using a `RequestDispatcher` method `include()` or `forward()` (see “Calling a Servlet Programmatically” on page 70), be aware that the request information is no longer sent as HTTP POST, GET, etc. `RequestDispatcher` methods always call `service()`. In other words, if a servlet overrides `doPost()`, it may not process anything if another servlet calls it, if the calling servlet happens to have received its data via HTTP GET. For this reason, be sure to implement routines for all possible types of input, as explained above.

Note Arbitrary binary data, like uploaded files or images, can be problematic, since the web connector translates incoming data into name-value pairs by default. You can program the web connector to properly handle this kind of data and package it correctly in the request object. See Appendix C, “Programming the Web Connector” for information about handling this type of data in the web connector.

Accessing Parameters and Storing Data

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

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

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

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

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

This reveals one way to transfer data to a JSP, since JSPs have access to the request object as an implicit bean. For more information, see “Using Java Beans” in Chapter 4, “Presenting Application Pages with JavaServer Pages.”

Handling Sessions and Security

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

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

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

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

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

Accessing Business Logic Components

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

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

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

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

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

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

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

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

```
String jndiNm = "Bookstore/cart/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 5, “Introducing Enterprise JavaBeans.”

Handling Threading Issues

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

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

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

- Synchronize access to all instance variables, as in the following example: `public synchronized void method()` (whole method) or `synchronized(this) {...}` (block only). Because synchronizing slows response time considerably, synchronize only blocks, or write your blocks so that they do not need to be synchronized.

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

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

public class myServlet extends HttpServlet {

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

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

- Create a single-threaded servlet by implementing `SingleThreadModel`. In this case, when you register a single-threaded servlet with NAS, the servlet engine creates a pool of 10 servlet instances (i.e., 10 copies of the same servlet in memory) that can be used for incoming requests. A single-threaded servlet can be slower under load because new requests must wait for a free instance in order to proceed, but this is less of a problem with distributed, load-balanced applications since the load automatically shifts to a less busy `kjs` process.

For example, this servlet is completely single-threaded:

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

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

Delivering Results to the Client

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

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

Creating a Response Page in a Servlet

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

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

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

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

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

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

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

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

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

Creating a Response Page in a JSP

Servlets can invoke JSPs in two ways:

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

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

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

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

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

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

Note You identify which JSP to call by specifying a URI, or Universal Resource Identifier. This is normally a path relative to `AppPath`, with a leading slash / if the JSP is part of the generic application. For instance, if your JSP is part of an application called `OnlineOffice`, you would refer to a JSP called `ShowSupplies.jsp` as `OnlineOffice/ShowSupplies.jsp`. On the other hand, if `ShowSupplies.jsp` is part of the generic application and is in a subdirectory, specify the subdirectory with a leading slash, as in `/GenericJSPs/ShowSupplies.jsp`.

For more information about JSPs, see Chapter 4, “Presenting Application Pages with JavaServer Pages.”

Creating the Servlet’s Configuration File

Servlet configuration files contain metadata, information about the servlet that identifies it and establishes its role in the application. Some optional NAS features are controlled by configuration entries.

Each servlet requires a configuration file to identify it to NAS. A single configuration file can configure multiple servlets. Additionally, each application must have a configuration file that contains references to all servlet configuration files. A configuration file is a hierarchical list of name-value pairs with ordinal types. The format is called NTV (name-type-value). Each NTV file must contain certain fields in order to properly register the servlet with NAS.

For more information about creating servlet configuration files, see “Creating Servlet and Application Configuration Files” in Chapter 10, “Creating Configuration Files.”

Dynamically Reloading Servlets

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

If you make any post-deployment changes to the `ServletRegistryInfo` section in a servlet’s configuration file, you must stop the server and load the new configuration into the NAS registry in order to make the new configuration visible to NAS. This is because the configuration is read only when servlets are first instantiated.

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

1. Stop the server.
2. Deploy the new configuration file.

3. Execute `servletReg configFile.ntv` on the server to register the new configuration file.
4. Restart the server.

Accessing Optional NAS Features

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

For more details on NAS features, see Chapter 13, “Taking Advantage of NAS Features.”

NAS also provides support for more robust sessions, based on a model from a previous version of NAS. This model uses the same API as the session model described in the servlet 2.1 specification, which is also supported. For more details on distributable sessions, see Chapter 11, “Creating and Managing User Sessions.”

Finally, the NAS implementation of security in servlets is a feature, since the servlet 2.1 specification makes no provision for security. The security model described in the forthcoming servlet 2.2 specification mirrors the NAS model for servlets. For more information about security in servlets, see Chapter 12, “Writing Secure Applications”

Invoking Servlets

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

Calling a Servlet With a URL

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

Invoking Specific Application Servlets

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

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

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

URL element	Description
<i>server:port</i>	Address and optional port number for the web server handling the request.
NASApp	Indicates to the web server that this URL is for a NAS application, so the request is routed to the NAS executive server. ^a
<i>appName</i>	The application's name, as configured in <code>appInfo.ntv</code> .
<i>servletName</i>	The servlet's name, as configured in the servlet configuration file.
<i>?name=value...</i>	Optional name-value parameters to the servlet.

a. This element is configurable by editing the following registry entry:
 Software\Netscape\Application Server\4.0\CCSO\HTTPAPI\SSPL_APP_PREFIX.
 See "The NAS Registry" in Chapter 10, "Creating Configuration Files."

The application name *appName* must correspond to a directory under *AppPath* (see "Important Servlet Files and Locations" on page 51). In this directory, the servlet engine expects to find a subdirectory *ntv*, in which there needs to be a file `appInfo.ntv`, and one or more configuration files defining each servlet used by the application. For example:

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

Invoking Generic Application Servlets

Address servlets that part of the generic application as follows:

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

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

URL element	Description
<i>server:port</i>	Address and optional port number for the web server handling the request.
<i>servlet</i>	Indicates to the web server that this URL is for a generic servlet object.
<i>servletName</i>	The servlet's name, as configured in the servlet configuration file.
<i>?name=value...</i>	Optional name-value parameters to the servlet.

For example:

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

Calling a Servlet Programmatically

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

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

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

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

For example:

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

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

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

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

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


Presenting Application Pages with JavaServer Pages

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

This chapter contains the following sections:

- Introducing JavaServer Pages
- How JSPs Work
- Designing JSPs
- Creating JSPs
- Invoking JSPs

Introducing JavaServer Pages

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

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

How JSPs Work

NAS compiles JSPs into servlets the first time they are called. This makes them available to the application environment as standard objects, and it also enables them to be called from a client using a URL. Think of servlets and JSPs as opposite sides of the same coin: each can perform the tasks of the other, but JSPs, as browser files, are best suited for layout tasks, while servlets are best suited for tasks that require Java code.

You can create JSPs that are not a part of any particular application. These JSPs are considered to be part of a generic application. JSPs can also run in the Netscape Enterprise Server (NES) and other web servers, but these JSPs have access to any application data, which makes their use limited.

JSPs that are included using a server-side include are compiled into the JSP when it is first compiled into a servlet.

A JSP and some other application components can be updated at run time without restarting the server, making it easy to change the look and feel of your application without stopping service. For more information, see Appendix B, “Dynamic Reloading.”

Responding to the Client

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

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

Designing JSPs

This section describes some of the decisions you must consider when you write JSPs. Since JSPs are compiled into servlets, the design considerations for servlets are also relevant to JSPs. See “Designing JSPs” in Chapter 4, “Presenting Application Pages with JavaServer Pages.”

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

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

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

The following sections describe specific design issues:

- Designing for Ease of Maintenance: How Many JSPs?
- Designing for Portability: Generic JSPs

Designing for Ease of Maintenance: How Many JSPs?

Each of your JSPs can call or include any number of other JSPs. For example, you could have a generic corporate banner, a standard navigation bar, and left-side column table of contents, each of which resides in a separate JSP that you include for each page you build. This situation is also a good candidate for using HTML frames.

A JSP can be included using a server-side include (SSI) or by including it using the `include()` method in the `RequestDispatcher` interface. You can also hand control of an interaction to a separate JSP using `RequestDispatcher.forward()`.

Designing for Portability: Generic JSPs

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

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

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

Creating JSPs

You create JSPs in much the same way that you create static browser pages (HTML or XML files), with the exception that you can add JSP-specific tags in order to provide powerful, conditional routines to dynamically create the content in your pages.

Also, there are a set of JSP tags specifically for embedding Java scripts in your JSPs. These scripts are called scriptlets, and can contain any Java code relevant to the environment in which the JSPs run. This means that JSPs have access to all of an application's components, including EJBs.

There are two types of JSP tags, described in the following sections.

- Bean tags, which provide support for using beans.
- Script tags, which provide access to a scripting language.

Using Java Beans

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

Beans themselves are separate classes. For more information about Java beans, see “Creating Java Beans” on page 84, and also visit the site <http://java.sun.com/beans>.

Note Note that this support is for standard Java beans, not EJBs. To access EJBs from your JSP, see “Accessing Business Objects” on page 96.

There are two implicit beans in all JSPs, the request bean, which encapsulates the same data as the request object in a servlet, and the exception bean, which contains information about the last error encountered by the JSP. For more information, see “Implicit Beans” on page 87.

JSP Tags that Support Beans

JSPs support beans with seven tags, described in the following sections:

- `<USEBEAN>`: Bean initialization.
- `<SETONCREATE>`: Bean initialization.
- `<SETFROMREQUEST>`: Bean initialization.
- `<DISPLAY>`: Display a property of the bean.
- `<LOOP>`: Loop to display multiple values in a bean property.
- `<INCLUDEIF>`: Conditionally include a block, based on a bean property.
- `<EXCLUDEIF>`: Conditionally exclude a block, based on a bean property.

Note In JSPs, a hierarchical naming scheme is used to describe bean properties. The use of colons : rather than periods . as separators helps distinguish this naming convention from Java syntax. (Note that a property may, itself, be a Bean containing further properties, as in `superman:costume:cape:color`.) Hierarchical naming is *not* used in the `<USEBEAN>` tag, where the attributes `NAME` and `TYPE` both use the Java variable naming convention.

<USEBEAN>

This set of tags makes a bean available for use later in the JSP. Note that you do not have to write <USEBEAN> to access implicit beans.

The general form of <USEBEAN> is:

```
<USEBEAN NAME="nameofbeaninstance"
          TYPE="nameofbeanclass"
          LIFESPAN="page|session|application" >
  <SETONCREATE>
</USEBEAN>
```

The attributes are all required and are defined as follows:

Attribute	Definition
NAME	Identifies the name of that specific instance of the Bean. The name must adhere to Java variable naming conventions and should not appear more than once per file. This value is case-sensitive.
TYPE	The value of the TYPE attribute is the name of the class that defines the bean. The value of this attribute is case-sensitive.
LIFESPAN	<p>This attribute defines the bean's scope. One of the following:</p> <ul style="list-style-type: none"> • page: This bean is valid for the life of this page request. If not already present, the bean is created and stored as part of the page request. • session: If the bean is present in the current session, it is reused. If not, it is created and stored as part of the session. • application: If the bean is present in the current application, it is reused. If it is not present, it is created and stored in the server context, and remains active for the life of the application.

Optionally, the <USEBEAN> tags may enclose one or more <SETFROMREQUEST> or <SETONCREATE> tags, or a combination of these, but no HTML or JSP tags are allowed between <USEBEAN> and </USEBEAN>.

You must declare `<USEBEAN>` tags for a bean before accessing any property of that bean, except for implicit beans.

For example:

```
<USEBEAN NAME="clock"
        TYPE="calendar.JspCalendar"
        LIFESPAN="page">
</USEBEAN>
```

Once a bean is declared, it can be accessed later in the file. For example:

```
<p>The name of the current month is <%= clock.getAttribute("month") %>.
```

<SETONCREATE>

This tag enables you to set initial properties in a bean if the JSP file creates the bean. You can only use this tag between `<USEBEAN>` tags.

The general form of `<SETONCREATE>` is:

```
<SETONCREATE BEANPROPERTY="propertyname" VALUE="propertyvalue">
```

The attributes are all required and are defined as follows:

Attribute	Definition
BEANPROPERTY	The name of the bean property whose value you wish to set. The property name must follow Java variable-naming conventions rather than hierarchal naming conventions.
VALUE	A value to be assigned to the named bean property.

For example:

```
<USEBEAN NAME="clock"
        TYPE="calendar.JspCalendar"
        LIFESPAN="page">
<SETONCREATE BEANPROPERTY="year" VALUE="1999">
</USEBEAN>
```

<SETFROMREQUEST>

This tag enables you to specify request parameters that are examined by the bean's property setter methods. In other words, you can use this tag to copy parameters from the incoming request into a newly-created bean. You can only use this tag between `<USEBEAN>` tags.

The general form of `<SETFROMREQUEST>` is:

```
<SETONCREATE BEANPROPERTY="*"|propertyname" PARAMNAME="propertyvalue">
```

The attributes are defined as follows:

Attribute	Definition
BEANPROPERTY	Required. Either an asterisk * that acts as a wildcard, or a property name for the bean. The property name must follow Java variable-naming conventions rather than hierarchal naming conventions.
PARAMNAME	Optional. A value to be assigned to the named bean property.

There are three behaviors with this tag:

- When the wildcard character * is used, BEANPROPERTY matches any property name. In this case, all request parameters are copied into the bean, matching parameter names in the request to property names in the bean. PARAMNAME is ignored.
- When a specific BEANPROPERTY value is provided but a PARAMNAME value is not, PARAMNAME is assumed to be equivalent to BEANPROPERTY. For example, `<SETFROMREQUEST BEANPROPERTY="foo">` is equivalent to `<SETFROMREQUEST BEANPROPERTY="foo" PARAMNAME="foo">`.
- If you specify both BEANPROPERTY and PARAMNAME, a bean property is named BEANPROPERTY is created. This property is assigned the value contained in the request parameter PARAMNAME. For example, if the request contains a parameter `dogName` with the value `Fido`, the following line creates an identical bean property:

```
<SETFROMREQUEST BEANPROPERTY="dogName" PARAMNAME="dogName">
```

The bean now has a property `dogName` with the value `Fido`.

<DISPLAY>

This tag displays a specific bean property.

The general form of `<DISPLAY>` is:

```
<DISPLAY PROPERTY="[beanname:+]property"
    PLACEHOLDER="substitutevalue">
```


The plus sign + indicates that more than one *property* can exist.

The attributes are all required and are defined as follows:

Attribute	Definition
PROPERTY	Required. Bean property to display, expressed using a hierarchical naming scheme. This attribute is case-sensitive.
PLACEHOLDER	Optional. A value to be displayed if the requested property value is missing or invalid.

Two examples of <DISPLAY>:

```
">
<DISPLAY PROPERTY="superman:favoriteDrink"
    PLACEHOLDER="<H3>Unknown!</H3>">
```

<LOOP>

This tag provides a mechanism to display a list of values with <DISPLAY>. A <LOOP> tag can contain a <DISPLAY> tag or another (nested) <LOOP> tag.

The general form of <LOOP> is:

```
<LOOP PROPERTY="beaname:indexedproperty"
    PROPTYELEMENT="x" >
<DISPLAY PROPERTY="x" PLACEHOLDER="substitutevalue">
</LOOP>
```

The attributes are all required and are defined as follows:

Attribute	Definition
PROPERTY	<p>The name of the multiple-valued bean property you wish to display. The bean should return either an array or a value and index. This property takes one of the following forms:</p> <ul style="list-style-type: none">• <code>type[] getFoo()</code>: array size is known, the loop displays each value in the array• <code>type getFoo (int index)</code>: if the component has a method <code>int getFooSize()</code>, the return value is taken to be the array's size, otherwise the loop starts at <code>index=0</code> and continues until <code>getFoo()</code> returns a null value or <code>ArrayOutOfBoundsException</code>.
PROPERTYELEMENT	<p>Place-holder for the property name, to be used as the value for <code>PROPERTY</code> in the enclosed <code><DISPLAY></code> tag. This attribute enables nested loops to display different properties, as in the examples below.</p>

This example shows a loop through the property `superman:allergies`:

```
<h2>Superman's Allergies:</h2>
<LOOP PROPERTY="superman:allergies"
    PROPERTYELEMENT="x" >
<DISPLAY PROPERTY= "x" PLACEHOLDER="not found">
</LOOP>
```

This example shows a two-dimensional loop through two related properties:

```
<h2>Phone numbers for grocery store deli departments: </h2>
<LOOP PROPERTY="store:grocery"
    PROPERTYELEMENT="i">
    <LOOP PROPERTY="i:phoneNumbers"
        PROPERTYELEMENT="j">
        <DISPLAY PROPERTY= "j:storeName">: <DISPLAY PROPERTY= "j:deliDept">
    </LOOP>
</LOOP>
```

<INCLUDEIF>

Use this tag to conditionally include a block of HTML code. The condition is based on the value of a bean property; if the condition evaluates to true, the enclosed HTML code is displayed in the final page, otherwise it is not.

The general form of `<INCLUDEIF>` is:

```
<INCLUDEIF PROPERTY="bean[:property]+"
        VALUE="valuetomatch"
        CASE="sensitive|insensitive
        MATCH="null|exact|contains|startswith|endswith">
```

The attributes are defined as follows:

Attribute	Definition
PROPERTY	The name of the multiple-valued bean property you wish to compare with <code>VALUE</code> .
VALUE	Value to match against <code>PROPERTY</code> .
CASE	Determines whether to match case, if the property is a String. Specify either <code>sensitive</code> or <code>insensitive</code> . The default is <code>insensitive</code> . If <code>MATCH</code> is specified, the comparison is automatically case-insensitive.
MATCH	Determines matching characteristics if the property is a String. The default is <code>exact</code> . If <code>MATCH</code> is specified, the comparison is automatically case-insensitive. Values are: <ul style="list-style-type: none"> <code>null</code>: <code>VALUE</code> is ignored, condition returns true if <code>PROPERTY</code> is null <code>exact</code>: <code>PROPERTY</code> matches <code>VALUE</code> exactly <code>contains</code>: <code>PROPERTY</code> contains <code>VALUE</code> <code>startswith</code>: <code>PROPERTY</code> starts with <code>VALUE</code> <code>endswith</code>: <code>PROPERTY</code> ends with <code>VALUE</code>

For example:

```
<INCLUDEIF PROPERTY="budget:validItems" VALUE="lunch"
        CASE="insensitive MATCH="contains">
<b>Reminder:</b> Take team to lunch. </INCLUDEIF>
```

<EXCLUDEIF>

Use this tag to conditionally exclude a block of HTML code. The condition is based on the value of a bean property; if the condition evaluates to true, the enclosed HTML code is not displayed in the final page, otherwise it is.

The general form of `<EXCLUDEIF>` is:

```
<EXCLUDEIF PROPERTY="bean[:property]+"
    VALUE="valuetomatch"
    CASE="sensitive|insensitive
    MATCH="null|exact|contains|startswith|endswith">
```

The attributes are defined as follows:

Attribute	Definition
PROPERTY	The name of the multiple-valued bean property you wish to compare with <code>VALUE</code> .
VALUE	Value to match against <code>PROPERTY</code> .
CASE	Determines whether to match case, if the property is a <code>String</code> . Specify either <code>sensitive</code> or <code>insensitive</code> . The default is <code>insensitive</code> . If <code>MATCH</code> is specified, the comparison is automatically case-insensitive.
MATCH	Determines matching characteristics if the property is a <code>String</code> . The default is <code>exact</code> . If <code>MATCH</code> is specified, the comparison is automatically case-insensitive. Values are: <ul style="list-style-type: none"> <code>null</code>: <code>VALUE</code> is ignored, condition returns true if <code>PROPERTY</code> is null <code>exact</code>: <code>PROPERTY</code> matches <code>VALUE</code> exactly <code>contains</code>: <code>PROPERTY</code> contains <code>VALUE</code> <code>startswith</code>: <code>PROPERTY</code> starts with <code>VALUE</code> <code>endswith</code>: <code>PROPERTY</code> ends with <code>VALUE</code>

For example:

```
<EXCLUDEIF PROPERTY="budget:invalidItems" VALUE="lunch"
    CASE="insensitive MATCH="contains">
<b>Reminder:</b> Take team to lunch. </INCLUDEIF>
```

Creating Java Beans

It is beyond the scope of this manual to describe how to create standard Java beans, since beans are described in their specification. There are many tutorials that describe how to create beans, some of which are accessible from the official site, <http://java.sun.com/beans>.

These are a few general tips regarding beans used in JSPs:

- When you invoke a bean with `<USEBEAN>`, a call is made to the bean's `processRequest()` method. This enables you to initialize any resources or variables that other bean methods might need, or perform any other constructor-type tasks. In the example below, a counter is incremented in `processRequest()`, which enables the JSP to display how many times the bean has been accessed since its instantiation.
- It is common in beans to have “getter” and “setter” methods to retrieve and set bean properties. Getter methods are named `getXxx()`, where `Xxx` is a property called `xxx` (the first letter is capitalized for the method name). If you have a corresponding setter called `setXxx()`, the setter must take a parameter of the same type as the return value from the getter. In the example below, `getFontSize()` returns an integer, and `setFontSize()` takes an integer as a parameter. These methods correspond to the variable `fontSize`.

Example Java Bean Access in a JSP

The following example shows a JSP called `ColorSize.jsp` that accesses a Java bean called `ColorSizeBean`. This example shows how to use the JSP tags described in this section.

ColorSize.jsp:

```

<USEBEAN NAME="csBean"
    TYPE="com.netscape.server.servlet.test.ColorSizeBean"
    LIFESPAN="page">
    <SETONCREATE beanproperty="fontSize" value="3">
    <SETFROMREQUEST beanproperty="fontSize" paramname="size">
</USEBEAN>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

<HTML>
<HEAD> <TITLE>Untitled</TITLE> </HEAD>

<BODY>
<!-- import the bean -->
<%@ import="com.netscape.server.servlet.test.ColorSizeBean" %>

<INCLUDEIF property="csBean:fontSize" value="6">
Hey, your font size is 6!<p>
</INCLUDEIF>

<EXCLUDEIF property="csBean:fontSize" value="4">
Hey, your font size is not 4!<p>
</EXCLUDEIF>

<table border=1>
<tr>
    <td align="center" colspan=2>
        <FONT SIZE=<DISPLAY property="csBean:fontSize">>
        Hello World
        </font>
    </td>
</tr>
<tr>
    <td align="center">
<a href="/servlet/ColorSize.jsp?size=<%= csBean.add(-1) %>">SMALLER</a>
    </td>
    <td align="center">
<a href="/servlet/ColorSize.jsp?size=<%= csBean.add(1) %>">BIGGER</a>
    </td>
</tr>
</table>

<p>Your current font size is: <DISPLAY property="csBean:fontSize">

<p>Your current counter is: <DISPLAY property="csBean:counter">

</BODY>
</HTML>

```

ColorSizeBean.java

```

package com.netscape.server.servlet.test;

import com.netscape.server.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorSizeBean implements java.io.Serializable {
    int counter = 0;
    private int size = 1;

    public int getFontSize() {
        return size;
    }

    public void setFontSize(int i) {
        size = i;
    }

    public int add(int x) {
        return size+x;
    }

    public int getCounter()
    {
        return counter;
    }

    public void processRequest(HttpServletRequest req) {
        counter++;
    }
}

```

Implicit Beans

There are two “implicit” beans that exist for each JSP:

- The exception bean `exception` (of type `java.lang.Throwable`), which maintains state on the most recent errors generated by the JSP
- The request bean `request`, which provides information about the incoming client request. The request bean provides access to environment variables, request parameters, and request headers. Many servlets encapsulate their business data in request parameters before passing this object to a JSP for formatting, so this mechanism provides an easy way to access data stored in the request object.

The request bean contains two nested beans:

- `request:headers`, which contains the request's HTTP headers
- `request:params`, which contains user-provided parameters and attributes such as form field data or, in the case of a request made from a NAS servlet, generated business data stored in the request object.

For more information about implicit beans, consult the JSP specification. All specifications are accessible from *install_dir/nas/docs/index.htm*, where *install_dir* is the location in which you installed NAS.

Embedded Java

You can embed Java code in your JSP in the form of directives, declarations, scriptlets (small Java scripts), and expressions. Attribute values do not require double quotes around them unless they contain spaces, but double quotes are recommended.

The components you can use to embed Java code in your JSP are described in the following table. In all cases, make sure to put a space character between the tag opening the component and any attributes it contains. For example, `<%= myBean.someVariable %>` is valid, but `<%=myBean.someVariable %>` is not.

Java Directives

```
<%@ variable="value" %>
```

Use directives to set preferences within the JSP. Possible directives to set are:

- **language:** the scripting language for this JSP. The only language supported with JSP 0.92 is Java, hence the only valid value is `language=java`. This is also the default.
- **errorpage:** a `.jsp` or `.html` file to be returned to the client if the current JSP results in an error.
- **import:** a comma-separated list of packages and/or class names imported for use within the compiled-page object.

For example:

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

Java Declarations

```
<SCRIPT RUNAT="..."> ...
</SCRIPT>
```

Use declarations to define variables that are valid throughout the JSP. The attribute `RUNAT` describes where the script runs; valid values are `client` and `server`. Set to `server` for all JSPs, otherwise the script is simply passed to the client.

For example:

```
<SCRIPT RUNAT="server">
int i=0;
String scriptname="myScript";
private void myMethod () { ... }
</SCRIPT>
```

Java Scriptlets

```
<% ... %>
```

Use scriptlets to define blocks of code to be executed. For example:

```
<% C balance = request.getAttribute("balance");
printValueAsHTML(balance); %>
```

There are four pre-defined implicit variables that you can use within a scriptlet (that is, between `<%` and `%>`):

Implicit variable	Description
request	The request object, as defined by <code>javax.servlet.http.HttpServletRequest</code> .
response	The response object, as defined by <code>javax.servlet.http.HttpServletResponse</code> .

Implicit variable	Description
<code>in</code>	The servlet input reader class, as defined by <code>java.io.BufferedReader</code> . This variable represents input from the client.
<code>out</code>	The servlet output writer class, as defined by <code>java.io.PrintWriter</code> . This variable represents output to the client.

Java Expressions

```
<%= ... %>
```

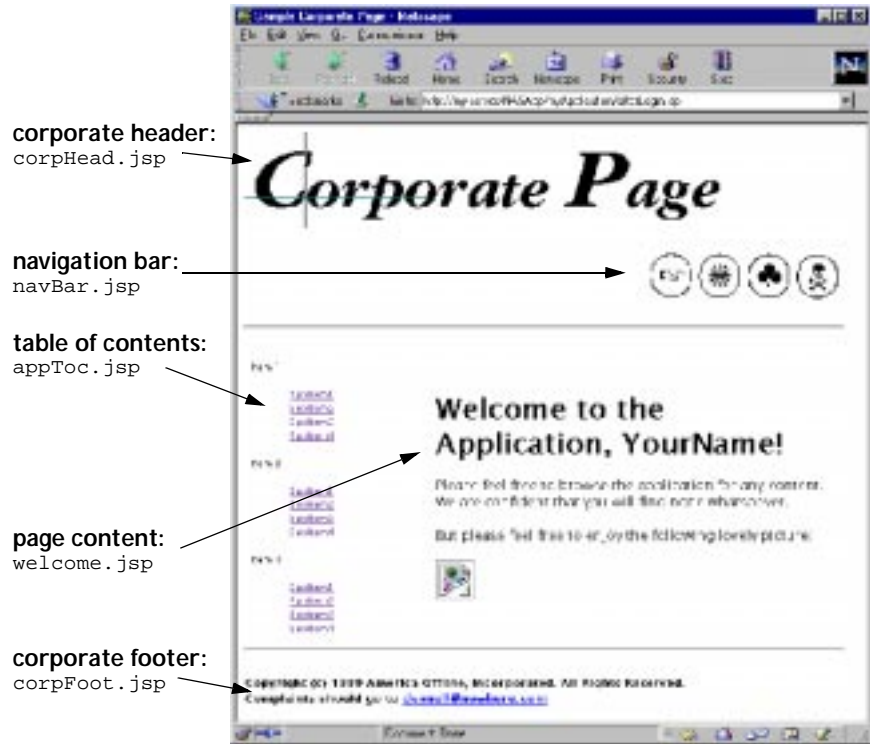
Use expressions as variables to be evaluated. The value of the expression is substituted where the expression occurs. For example:

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

Including Other JSPs

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

The following example shows how each portion of a page can be provided by a separate JSP. The example page is followed by the code that constructs it.



There are three ways to include a JSP in your JSP. These methods are discussed in the following sections.

Note You identify which JSP to call by specifying a path relative to `AppPath`, with a leading slash / if the JSP is part of the generic application. For instance, if your JSP is part of an application called `OnlineOffice`, you would refer to a JSP called `ShowSupplies.jsp` as `OnlineOffice/ShowSupplies.jsp`. On the other hand, if `ShowSupplies.jsp` is part of the generic application and is in a subdirectory, specify the subdirectory with a leading slash, as in `/GenericJSPs/ShowSupplies.jsp`.

Warning If included JSPs are self-contained with HTML tags, some browsers may not display properly. The included JSP should only contain parts of the head and body.

Including JSPs Using SSI

A JSP can be included from another JSP using a server-side include (SSI). This is similar to a macro call; the called JSP is inserted inline into the calling JSP when it is compiled, resulting in a single callable object.

Also, variable names that are common to both the including JSP and the included JSP, which can cause compile-time errors in the single generated servlet. To avoid this, use separate variable names in all servlets that include or are included in other servlets.

A tutorial that discusses server-side includes (SSI) can be found at:
<http://hoohoo.ncsa.uiuc.edu/docs/tutorials/includes.html>

Usage

You specify an included file as follows:

```
<!--#include virtual="directory" file="filename" -->
```

The `file` attribute instructs the server to search for the listed `filename` beginning in the directory listed in the `virtual` attribute, or in the current directory if `virtual` is not specified. `virtual` is a directory path relative to `AppPath`, a NAS registry variable that indicates the top of the application tree. `file` can contain slashes to indicate subdirectories of `virtual`.

The JSP specification introduces another use for these attributes, namely:

```
<!--#include virtual="filename" -->
```

In this case, `filename` is the full sub-path to the included file, starting at `AppPath`.

NAS does not support the `exec=" "` SSI command.

Example

This example code shows a JSP that generates the page sample page shown on page 91 by including the output from other JSPs using server-side includes:

```
<html>
<head><title>Sample Corporate Page</title></head>
<body>
<!--#include virtual="" file="corpHead.jsp" -->
<!--#include file="navBar.jsp" --> <!-- in current directory -->
<hr>
<table border=0><tr>
<td width="30%"><!--#include file="appToc.jsp" --></td>
<td width="70%"><!--#include file="welcome.jsp" --></td>
</tr></table>
<!--#include file="corpFoot.jsp" -->
</body></html>
```

Calling JSPs Using include

The `include()` method in the `RequestDispatcher` interface calls a JSP and waits for it to return before continuing to process the interaction. Since this is a Java method rather than a server-side include, the included JSP is compiled separately and exists as a separate object that can be reused.

Usage

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

```
<% RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("JSP_URI.jsp");
dispatcher.include(request, response);
%>
```

You identify which JSP to call by specifying a URI, or Universal Resource Identifier. This is normally a path relative to `AppPath`, with a leading slash / if the JSP is part of the generic application. For instance, if your JSP is part of an application called `OnlineOffice`, you would refer to a JSP called `ShowSupplies.jsp` as `OnlineOffice/ShowSupplies.jsp`. On the other hand, if `ShowSupplies.jsp` is part of the generic application and is in a subdirectory, specify the subdirectory with a leading slash, as in `/GenericJSPs/ShowSupplies.jsp`.

Example

This example creates the sample page shown on page 91 using `include()`:

```
<html>
<head><title>Sample Corporate Page</title></head>
<body>
<%
(RequestDispatcher)getRequestDispatcher("corpHead.jsp").include(request
, response);
(RequestDispatcher)getRequestDispatcher("navBar.jsp").include(request,
response); %>

<hr>
<table border=0>
<tr>
    <td width="30%">
        <%
(RequestDispatcher)getRequestDispatcher("appToc.jsp").include(request,
response) %>
        </td>
        <td width="70%">
            <%
(RequestDispatcher)getRequestDispatcher("welcomePage").include(request,
response) %>
            </td>
</tr>
</table>

<%
(RequestDispatcher)getRequestDispatcher("corpFooter").include(request,
response) %>
</body></html>
```

Forwarding JSPs Using forward

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

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

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

You identify which JSP to call by specifying a URI, or Universal Resource Identifier. This is normally a path relative to `AppPath`, with a leading slash `/` if the JSP is part of the generic application. For instance, if your JSP is part of an application called `OnlineOffice`, you would refer to a JSP called `ShowSupplies.jsp` as `OnlineOffice/ShowSupplies.jsp`. On the other hand, if `ShowSupplies.jsp` is part of the generic application and is in a subdirectory, specify the subdirectory with a leading slash, as in `/GenericJSPs/ShowSupplies.jsp`.

Page Content Elements

At development time, a JSP contains placeholders for page content. The content itself is generated at run time and accessed by the JSP either through attributes in the `request` object that contain the required information, or directly from the business objects that control the content (EJBs, via a JNDI proxy). JSPs have access to the same public variables controlled by the servlet which called it.

Of course, since JSPs are compiled into servlets when they are first accessed, you can perform presentation logic tasks with JSPs as well. However, the programming model discourages this in favor of using servlets for presentation logic and EJBs for business logic. The exception is in cases where the logic is simple, such as name lookups.

Dynamic content can be accessed by a JSP in three ways:

- Using Java Beans (generating content on the fly)
- Accessing the Request Object
- Accessing Business Objects (content accessed directly from EJBs)

Accessing the Request Object

You can access parameters and attributes of the request object as a standard system variable using the `<% . . . %>` JSP tag. Parameters are name-value pairs sent from the client, including form field data, HTTP header information, etc. Attributes are name-value pairs that can be set by servlets, and so are not present if the JSP is called directly from via a URL.

For example, in a servlet, you can request an account balance and store the result as an attribute of the request object using `setAttribute()`, as in this example from a servlet:

```
request.setAttribute("balance", balance);
```

You can then access that variable from the JSP using `getAttribute()`:

```
<% C balance = request.getAttribute("balance"); %>
```

Similarly, you can access a parameter from the request using `getParameter()` in your JSP:

```
<% String name = request.getParameter("username"); %>
```

This example shows how a short Java loop can be written to display arrays:

```
<table border=1>
<% for ( i=0 ; i < request.getAttribute("totalRecords") ; i++ ) { %>
    <td align=left><%= request.getAttribute(attribute[i]) %></td>
</table>
```

Accessing Business Objects

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

The method described here for contacting EJBs is identical to the method used from servlets. For more information, see “Accessing Business Logic Components” in Chapter 3, “Controlling Applications with Servlets.”

We recommend you use servlets rather than JSPs to contact EJBs in order to properly partition the presentation components in your application: servlets are best suited for presentation logic, JSPs are best suited for presentation layout.

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

```
<%@ import cart.ShoppingCart %>;
...
<% // Get the user's session and shopping cart
    HttpSession session = request.getSession(true);
    ShoppingCart cart = (ShoppingCart)session.getValue(session.getId());

    // If the user has no cart, create a new one
```



```

        if (cart == null) {
            cart = new ShoppingCart();
            session.putValue(session.getId(), cart);
        } %>
...
<%= cart.getDataAsHTML() %>

```

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

```

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

```

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

For more information on contacting EJBs, see “Accessing Business Logic Components” in Chapter 3, “Controlling Applications with Servlets.”

Invoking JSPs

You can invoke a JSP programmatically from a servlet, or you can address it directly from a client using a URL. You can also include JSPs as well as invoke them; see “Including Other JSPs” on page 90.

Calling a JSP With a URL

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

Invoking JSPs in a Specific Application

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

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

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

URL element	Description
<i>server:port</i>	Address and optional port number for the web server handling the request.
<i>NASApp</i>	Indicates to the web server that this URL is for a NAS application, so the request is routed to the NAS executive server. This element is configurable by editing the following registry entry: Software\Netscape\Application Server\4.0\CCS0\HTTPAPI\SSPL_APP_PREFIX.
<i>appName</i>	The application’s name, which must be identical to the name of the application’s subdirectory under AppPath.
<i>jspName</i>	The JSP’s filename, including the .jsp extension.
<i>?name=value...</i>	Optional name-value parameters to the JSP. These are accessible from the request object.

For example:

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

Invoking JSPs in the Generic Application

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

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

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

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

For example:

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

Invoking a JSP Programmatically

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

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

For more information on these methods, see “Delivering Results to the Client” in Chapter 3, “Controlling Applications with Servlets.”

A JSP can also invoke another JSP. For more information, see “Including Other JSPs” on page 90.

Introducing Enterprise JavaBeans

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

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

This chapter includes the following sections:

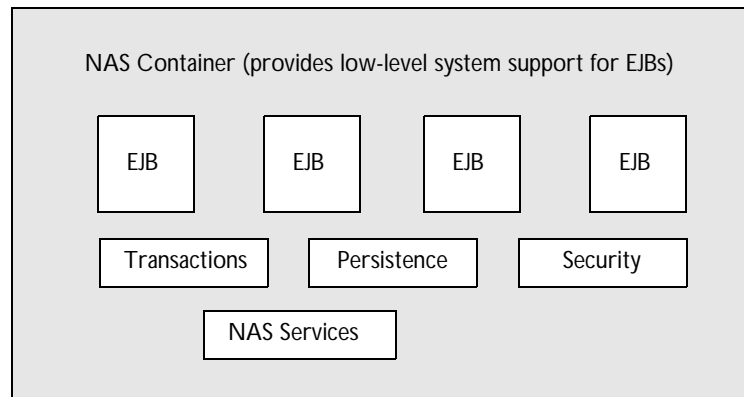
- What Enterprise JavaBeans Do
- What Is an Enterprise JavaBean?
- Session Beans and Entity Beans
- The Role of EJBs in NAS Applications
- Designing an Object-Oriented Application

Note If you already know about EJBs and how they are used in NAS, you may want to jump ahead to the chapter containing specific instructions and guidelines for developing EJBs for use with NAS: Chapter 7, “Building Business Entity EJBs” and Chapter 6, “Using Session EJBs to Manage Business Rules.”

What Enterprise JavaBeans Do

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

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



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

What Is an Enterprise JavaBean?

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

<http://java.sun.com:8081/products/ejb/docs.html>.

The fundamental characteristics of EJBs are as follows:

- Creation and management of beans is handled at runtime by a container. NAS itself provides the container for enterprise beans you write as part of your server application.
- Customization of beans is handled at deployment time by editing the beans' environment properties.
- Manipulation of metadata, such as transaction mode and security attributes, are separated from the enterprise Bean class, and handled instead by the container's tools at design and deployment time.
- Mediation of client access is handled by the container and the server where the bean is deployed, freeing the bean designer from having to process it.
- Restricting a bean to use standard container services defined by the EJB specification guarantees that the bean is portable and can be deployed in any EJB-compliant container.
- Including a bean in, or adding a bean to an application made up of other, separate bean elements—a “composite” application—does not require source code changes or recompiling of the bean.
- Definition of a client's view of a bean is controlled entirely by the bean developer. The view is not affected by the container in which the bean runs or the server where the bean is deployed.

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

Understanding Client Contracts

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

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

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

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

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

Understanding Component Contracts

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

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

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

Understanding Jar File Contracts

The jar file contract specifies what information must be in the jar file a developer uses to package an EJB for deployment. With NAS, you can create a jar file containing EJBs using the NAS Deployment Tool. For more information, see the *Administration Guide*.

A jar file for an EJB includes:

- File manifest entries that describe the contents of the file.
- Java class files for the bean(s) in the file.
- Deployment descriptors that include the declarative attributes associated with a bean. These attributes tell the container how to manage the bean.
- Environment properties that the bean(s) require at runtime.

Session Beans and Entity Beans

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

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

Understanding Session Beans

A session EJB typically has the following characteristics. It:

- Executes in relation to a single client.
- Optionally handles transaction management according to property settings.
- Optionally updates shared data in an underlying database.
- Is relatively short-lived.
- Is not guaranteed to survive a server crash.

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

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

Understanding Entity Beans

An entity EJB has the following characteristics. It:

- Represents data in the database.
- Is always transactional.
- Allows shared access for any number of users.
- Can exist as long as its data in a database.
- Can transparently survive crashes of the EJB server.

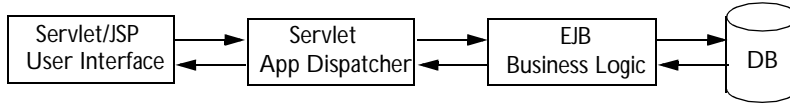
The server that hosts EJBs and an EJB container provide a scalable runtime environment for many concurrently active entity EJBs. Entity EJBs represent either persistent data (such as a database or document), or an object that is used by an existing enterprise application.

In the NAS model, an EJB encapsulates a business object, such as a database. The life duration of an entity EJB can span the entire life of the application that instantiates it, or it can span some portion of it. For example, suppose you are in the automotive parts industry. You might create an inventory control entity bean that interfaces with your parts database everywhere throughout your application.

The Role of EJBs in NAS Applications

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

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



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

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

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

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

The greatest challenge you face when creating an application that uses EJBs is determining how to break your application into a meaningful set of non-enterprise JavaBeans, and enterprise entity and session EJBs.

Designing an Object-Oriented Application

Partitioning an NAS application's business logic and data processing into the most effective set of EJBs is a large part of your job as a developer. There are no hard and fast rules for object-oriented design with EJBs, other than that

instances of entity beans tend to be long-lived, persistent, and shared among clients, while instances of session beans tend to be short-lived, and used only by a single client. Therefore, what follows in these sections is mostly high-level, NAS-specific advice for improving application speed, making your EJBs modular and shareable, and easing maintenance.

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

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

Planning Guidelines

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

- Keep the central and application-specific parts of your application in as few, larger, entity EJBs as possible.
- Restrict most of your long-term database connectivity to your entity EJBs.
- Ask your administrator to co-locate EJBs with your presentation logic (servlets and JSPs) on the same server to reduce the number of remote procedure calls (RPCs) when you run your application.
- Create stateless session beans instead of stateful session beans as much as possible. If you must create stateful session beans, have your administrator turn on sticky load balancing for better performance.

- Create session EJBs that are small, generic, and narrowly task-focused. Ideally, these EJBs encapsulate behavior that can be used in many of your applications.

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

Using Session Beans

Session beans are intended to represent transient objects and processes, such as updating a single database record, a copy of a document for editing, or specialized business objects for individual clients, such as a shopping cart. These objects are available only to a single client. When the client is done with them, the objects are released. When you design an application, designate each such temporary, single-client object as a potential session bean. For example, in an online shopping application, each customer's shopping cart is a temporary object. The cart lasts only as long as the customer is selecting items for purchase. Once the customer is done and the order is processed, the cart object is no longer needed and is released.

Like an entity bean, a session bean may access a database through JDBC calls. A session bean, too, can provide transaction settings. These transaction settings and JDBC calls are refereed by the session bean's container, which is transparent to you as the developer. The container provided with NAS 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 NAS application, see Chapter 6, "Using Session EJBs to Manage Business Rules."

Using Entity Beans

Entity beans are intended to represent persistent objects, such as data, documents, and specific business objects that may be accessed by multiple clients. When you design an application designate each business object as a potential entity bean. For example, an inventory control system combines a

database with specialized methods for search, retrieval, removal, restocking, and reporting. The inventory control system is a perfect candidate for an entity bean.

An entity bean accesses a database through a combination of bean settings for transaction control and JDBC calls. These transaction settings and JDBC calls are refereed by the entity bean's container, which is transparent to you as the developer. The container provided with NAS handles the JDBC calls and result sets.

The high-level ability to support transactions through EJB properties is especially useful for entity beans because the bean can encapsulate control for distributed transactions that span multiple datasources without requiring you to handle the details.

While an entity bean may represent a business object to one or many clients, often there are specific client database events that are unique to the client. In these cases, you may want to use a session bean to control a client's interaction with the entity bean's database representation.

For a complete discussion of using entity beans to define persistent objects and business logic in an NAS application, see Chapter 7, "Building Business Entity EJBs".

Planning for Failover Recovery

Failover recovery is a process whereby a bean can reinstantiate itself after a server crash. Stateless session beans support failover recover, but stateful session beans do not.

Entity beans support failover recovery with the caveat that the reference to the bean is lost after a server crash. To recover an entity bean, you must create a new reference to it with a finder (see "Using Finder Methods" in Chapter 7, "Building Business Entity EJBs").

Working with Databases

In NAS, the preferred method for working with databases is through the Java Database Connectivity (JDBC) API in conjunction with the transaction controls available through the EJB interface. You can also use the Java Naming and

Directory Interface (JNDI) to obtain a database connection. JNDI provides a standard way for application to find and access database services independent of JDBC drivers.

NAS fully supports JDBC 1.0 and also supports most of the JDBC 2.0 specification, including connection pools, distributed transaction support, and rowsets.

For a complete discussion of using entity beans to define persistent objects and business logic in an NAS application, see Chapter 9, “Using JDBC for Database Access.”

For a complete description of transaction controls available through session and entity beans, see Chapter 8, “Handling Transactions with EJBs.”

Deploying EJBs

You normally deploy EJBs with the rest of an application using the NAS Deployment Manager. You can also choose to deploy manually for the purpose of testing your EJB, or upgrading before restarting the server.

For more information, see Chapter 2, “Deploying and Upgrading Applications” in the *Administration Guide*.

Using Session EJBs to Manage Business Rules

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

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

This chapter includes the following sections:

- Introducing Session EJBs
- Session Bean Components
- Additional Session Bean Guidelines
- Determining the Business Rules in Your Applications

Introducing Session EJBs

Much of a standard, distributed application consists of logical units of code that perform repetitive, time-bound, and user-dependent tasks. These tasks can be simple or complex, and they are often needed in different applications. For example, banking applications must verify a user's account ID and balances before performing a fund transfer of any kind. These kinds of tasks define the

business rules and business logic that you use to run your business. Such discrete tasks, transient by nature, are perfect candidates for session Enterprise JavaBeans (EJBs).

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

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

Session beans can either be “stateless” or “stateful.” 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. A stateful session bean is also transient, uses a “conversational state” to preserve information about its contents and values between client calls. The conversational state enables the container to maintain information about the state of the session bean and to recreate that state at a later point in program execution when needed.

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

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

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

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

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

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

Session Bean Components

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

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

Creating the Remote Interface

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

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

public interface MySessionBean extends EJBObject {
    // define business method methods here....
}
```

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

- The same name.
- The same number and type of arguments, and the same return type.
- The same exceptions defined in the `throws` clause, except that in the remote interface definition, the `throws` clause must also include `java.rmi.RemoteException`.

Besides the business methods you define in the remote interface, the `EJBObject` interface defines several abstract methods that enable you to retrieve the home interface for the bean, to retrieve the bean's handle, which is its unique identifier, to compare the bean to another bean to see if it is identical, and to free, or remove the bean when it is no longer needed. When you implement beans, do not override these built-in methods.

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

Declaring vs. Implementing the Remote Interface

A bean class definition must include one matching method definition, including matching method names, arguments, and return types, for each method defined in the bean's remote interface. The EJB specification also permits the bean class

to implement the remote interface directly, but recommends against this practice to avoid inadvertently passing a direct reference (via `this`) to a client in violation of the client-container-EJB protocol intended by the specification.

Creating the Class Definition

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

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

public class MySessionBean implements SessionBean {

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

    public void ejbActivate() throws RemoteException {
    }

    public void ejbPassivate() throws RemoteException {
    }

    public void ejbRemove() throws RemoteException{
    }

    public void setSessionContext(SessionContext ctx) throws
    RemoteException {
    }

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

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

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

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

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

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

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

Finally, there are a couple of optional interface implementations permitted in a session bean class definition, particularly `javax.ejb.SessionSynchronization`, which 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/nas/docs/index.htm`, where `install_dir` is the location in which you installed NAS.

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 property file. For more information, see "Specifying Session Timeout" in Chapter 10, "Creating Configuration Files."

Passivation and Activation

You can "passivate" a bean rather than destroying it. Passivation returns the bean to a pool of active beans so that it can be later reactivated rather than reinstantiating a new bean.

You can control passivation in the `ejbPassivate()` and `ejbActivate()` methods. The container calls these methods to passivate or activate a bean.

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 "Specifying Passivation Timeout" in Chapter 10, "Creating Configuration Files."

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

Creating the Home Interface

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

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

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

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

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

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

Additional Session Bean Guidelines

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

The NAS container supports both state management modes, so you may choose the state most appropriate for your session beans. Use stateless session beans as much as possible for performance reasons. In general, stateless session beans also scale more efficiently in a highly distributed environment. All session beans must be serializable. Because stateful session beans contain much state information that must be both serializable and capable of being recreated, they tend to perform less efficiently and scale less well in a highly distributed environment.

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

Accessing NAS Functionality

You can develop session beans that adhere strictly to the EJB Specification, you can develop session beans that take advantage both of the specification and additional, value-added NAS features, and you can develop session beans that

adhere to the specification in non-NAS environments, but that take advantage of NAS features if they are available. Make the choice that is best for your intended deployment scenario.

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

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

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

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

Serializing Handles and References

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

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

Managing Transactions

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

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

Committing a Transaction

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

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

Note that transactions begun in the container are implicitly committed. Also, any participant can roll back a transaction. For more information about transactions, see Chapter 8, "Handling Transactions with EJBs."

Accessing Databases

Many session beans access and even update data. Because of the transient nature of session beans, however, be careful about how that access takes place. In general, use the JDBC API to make your calls, and always use the transaction and security management methods described in Chapter 8, “Handling Transactions with EJBs” to manage transaction isolation level and transaction requirements at the bean level.

For more information see Chapter 9, “Using JDBC for Database Access.”

Determining the Business Rules in Your Applications

When you adopt the NAS application programming model defined in this book, you split your application into general parts. For example, you separated your presentation logic from your business logic and business entities, such as databases. You also determined which part(s) of your application were best handled as servlets, the central dispatcher for your application. What’s left is the heart of your application, its business—logical flow, access rights, calculating, processing, and data access. The next step is to separate what are business entities from what are business rules and business logic.

Choosing a Coarse Bean Granularity

In many cases business rules may well involve database access. For example, if you are building an enterprise-wide employee database access program, there may be discrete business rules about what employees can access which parts of the database. For example, you will want to grant access to names, telephone extension, and email address information to all of your employees so they can contact one another. On the other hand, perhaps only managers and payroll employees will have access to employee compensation information, and perhaps only certain payroll employees will actually be able to update the employee compensation information. In these cases, there are different database access rules for each category of employee. Is each of these categories a candidate for a separate session bean, or should you write a larger, more sophisticated session bean that can handle all different employee types?

In general, you should create your application out of as few beans as reasonable. An application with fewer, larger beans significantly outperforms an application with many, smaller beans because there is less system overhead, and fewer remote procedure calls to process. Otherwise, there is no hard and fast rule about where the boundary lies.

One option might be to create a session bean that determines employee access rights and then passes that information as an argument to a single database access session bean. The questions here are as follows:

- What level of granularity do you want in your application?
- What level of bean reuse do you want or anticipate among your applications?
- How important is application speed and performance?

The smaller the elements you break your application into, the easier it becomes to isolate functionality and maintain it. In addition, many smaller elements tend to be more generic, and easier to reuse in other applications. On the other hand, the more elements you use to construct a composite enterprise application, the more system overhead there is, and the slower your application may run. In general, unless your site has many applications it uses, it is always best to balance the ease of maintenance and reuse with the speed and performance of your applications.

Performance Considerations

As a bean developer, you can make choices that affect the application's speed and performance at bean design time. For example, to increase performance, you may choose to create fewer, larger beans that are not as easily reusable in other applications. You may choose to group database access and updates to minimize data traffic.

There is no hard and fast rule for affecting application speed, except that fewer, larger, stateless beans tend to produce a faster application than many smaller or stateful beans. You should work with your system administrator to anticipate how your application will be deployed and how many users will access the system at the same time. Armed with that information, you can make wiser choices about how to divide your business rules across session EJBs.

Building Business Entity EJBs

This chapter describes how to use entity Enterprise JavaBeans (EJBs) to encapsulate your business entities.

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

This chapter contains the following sections:

- Introducing Business Entity EJBs
- Entity Bean Components
- Additional Entity Bean Guidelines
- Determining the Business Objects in Your Applications

Note Netscape Application Server (NAS) 4.0 offers full support for entities even though such support is optional in the EJB Specification. The reason for this is because using entities lends flexibility to your applications.

All specifications are accessible from `install_dir/nas/docs/index.htm`, where `install_dir` is the location in which you installed NAS.

Introducing Business Entity EJBs

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

Business EJBs are self-contained, reusable components—with data members, properties, and methods—that represent instances of generic, transactionally aware, persistent data objects that can be shared among clients. *Persistence* refers to the creation and maintenance of a bean throughout the lifetime of the application. In NAS 4.0, beans are responsible for their own persistence, called *bean-managed persistence*.

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

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

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

Clients access an entity bean through the bean's remote interface. The object that implements the remote interface is called the EJB object. Usually, an entity EJB is shared among multiple clients, and represents a single point of entry to a

data resource or business object, such as a database. Regardless of which client access an entity object at a given time, each client's view of the object is both location independent, and transparent to other clients.

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

An entity bean:

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

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

How an Entity Bean Is Accessed

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

A client never accesses instances of an entity bean directly. Instead, a client uses the entity bean's remote interface to access a bean instance. The EJB object class that implements an entity bean's remote interface is provided by the container. At a minimum, an EJB object supports all of the methods of the `java.ejb.EJBObject` interface. This includes methods to obtain the entity bean's home interface, to get the object's handle, to retrieve the entity's primary key, to test if the object is identical to another object, and to remove the object.

These methods are stipulated by the EJB Specification. All specifications are accessible from *installdir/nas/docs/index.htm*, where *installdir* is the location in which you installed NAS.

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

Entity Bean Components

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

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

Creating the Class Definition

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

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

public class MyEntityBean implements EntityBean {

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

    public void ejbActivate() throws RemoteException {
    }

    public void ejbLoad() throws Remote Exception {
    }

    public void ejbPassivate() throws RemoteException {
    }

    public void ejbRemove() throws RemoteException{
    }

    public void ejbStore() throws RemoteException{
    }
}
```



```

public void setEntityContext(EntityContext ctx) throws RemoteException {
}

public void unsetEntityContext() 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 also define one `ejbPostCreate()` method for each `ejbCreate()` method. It may also optionally provide additional, developer-defined finder methods that take the form `ejbFindXXX`, where `XXX` represents a unique continuation of a method name (e.g., `ejbFindApplesAndOranges`) that does not duplicate any other methods names.

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

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

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

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

```

private transient javax.ejb.EntityContext m_ctx = null;

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

```

Using `ejbActivate` and `ejbPassivate`

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

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

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

The container passivates session beans after they are inactive for a specified (or default) time. This timeout value is set in the bean's property file. For more information, see "Specifying Passivation Timeout" in Chapter 10, "Creating Configuration Files."

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

For more information about `ejbCreate()` and `ejbRemove()`, see "Using `ejbCreate` Methods" on page 132.

Using `ejbLoad` and `ejbStore`

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

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

```

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

    itemId = (String) m_ctx.getPrimaryKey();

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

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

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

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

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

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

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

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

    dc = new DatabaseConnection();

```

```

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

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

```

Note For related information about isolation levels in beans that access transactions concurrently with other beans, see “Handling Concurrent Access” on page 138.

Using `setEntityContext` and `unsetEntityContext`

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

```

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

```

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

```

public void unsetEntityContext()
{
    m_ctx = null;
}

```

Using `ejbCreate` Methods

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

```

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

```

Each `ejbCreate(...)` method must be declared as `public`, return either the entity's primary key type or a collection, and be named `ejbCreate`. The return type can be any legal Java RMI type that can be converted to a number for key purposes. Any arguments must be legal types for Java RMI. The `throws` clause, may define application-specific exceptions, may include `java.rmi.RemoteException`, or `java.ejb.CreateException`.

For each `ejbCreate()` method, the entity bean class may optionally define an `ejbPostCreate()` method to handle entity services immediately following creation. Each `ejbPostCreate()` method must be declared as `public`, must return `void`, and be named `ejbPostCreate`. The method arguments, if any, must match in number and type the arguments of its corresponding `ejbCreate` method. The `throws` clause, may define application-specific exceptions, may include `java.rmi.RemoteException`, or may include `java.ejb.CreateException`.

Finally, an entity bean also implements one or more `ejbRemove()` methods to free a bean when it is no longer needed.

Using Finder Methods

Because entity beans are persistent, are shared among clients, and may have more than once instance instantiated at the same time, an entity bean must implement at least one method, `FindByPrimaryKey()`, that enables the client and the bean's container to locate a specific bean instance. All entity beans must provide a unique primary key as an identifying signature. You can implement the `FindByPrimaryKey()` method in the bean's class to enable a bean to return its primary key to the container.

The following example shows a definition for `FindByPrimaryKey()`:

```
public String ejbFindByPrimaryKey(String key)
    throws java.rmi.RemoteException,
           javax.ejb.FinderException
{
    //System.out.println("@@@ myBean.ejbFindByPrimaryKey key = " + key);
    return key;
}
```

In some cases, you may want to find a specific instance of an entity bean based on what the bean does, based on certain values the instance is working with, or based on still other criteria. These names of these implementation-specific

finder methods take the form `ejbFindXXX`, where `XXX` represents a unique continuation of a method name (e.g., `ejbFindApplesAndOranges`) that does not duplicate any other methods names.

Finder methods must be declared as public, and their arguments and return values must be legal types for Java RMI. The return type of each finder method must be the entity bean's primary key type or a collection of objects of the same primary key type. If the return type is a collection, the return type must be `java.util.Enumeration`.

The `throws` clause of a finder method can be an application specific exception, may include `java.rmi.RemoteException` and/or `java.ejb.FinderException`.

Declaring vs. Implementing the Remote Interface

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

Creating the Home Interface

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

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

public interface MyEntityBeanHome extends EJBHome {
    MyEntityBean create() throws CreateException, RemoteException;
}
```

As this example illustrates, an entity bean's home interface defines one or more `create` methods. Usually the home interface also defines one or more `find` methods corresponding to the `finder` methods in the bean class.

Defining Create Methods

Each such method must be named `create`, and must correspond in number and type of arguments to an `ejbCreate` method defined in the entity bean class. The return type for each create method, however, does not match the return type of its corresponding `ejbCreate` method. Instead, it must return the entity bean's remote interface type.

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

Defining Find Methods

A home interface can define one or more `find` methods. Each such method must be named `findXXX` (e.g., `findApplesAndOranges`), where `XXX` is a unique continuation of the method name. Each finder method must correspond to one of the finder methods defined in the entity bean class definition. The number and type of arguments must also correspond to the finder method definitions in the bean class. The return type, however, may be different. The return type for a finder method in the home interface must be the entity bean's remote interface type or a collection of interfaces.

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

Creating the Remote Interface

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

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

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

- The same name.
- The same number and type of arguments, and the same return type.
- The same exceptions defined in the `throws` clause, except that in the remote interface definition, the `throws` clause must also include `java.rmi.RemoteException`.

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

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

Additional Entity Bean Guidelines

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

Accessing NAS Functionality

You can develop entity beans that adhere strictly to the EJB Specification, you can develop entity beans that take advantage both of the specification and additional, value-added NAS features, and you can develop entity beans that

adhere to the specification in non-NAS environments, but that take advantage of NAS features if they are available. Make the choice that is best for your intended deployment scenario.

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

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

Managing Transactions

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

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

Committing a Transaction

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

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

Note that transactions begun in the container are implicitly committed. Also, any participant can roll back a transaction. For more information about transactions, see Chapter 8, "Handling Transactions with EJBs."

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

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

You typically perform this synchronization in conjunction with the underlying database or resource. One possible approach would be to acquire the corresponding database locks in the `ejbLoad()` method, for example by

choosing an appropriate isolation level or by using a “select for update” clause. The specifics depend on the database backend that is being used. For more information, see the EJB specification as it relates to concurrent access.

The following example `ejbLoad()` snippet illustrates the use of “select for update” syntax to obtain database locks. This prevents other instances from being loaded at the same time.

```
public void ejbLoad() throws java.rmi.RemoteException
{
    ....
    // Get the lock on the corresponding DB table
    try {
        java.sql.Connection dbConn = ds.getConnection();
        String query = "SELECT accountNum, balance FROM accounts "
            + "WHERE customerId = ? FOR UPDATE";
        prepStmt = dbConn.prepareStatement(query);
        prepStmt.setString(1, m_customerId);
        resultSet = prepStmt.executeQuery();
        if ((resultSet != null) && resultSet.next()) {
            acctNum = resultSet.getInt(1);
            acctBalance = resultSet.getInt(2);
        } else {
            throw new RemoteException("Database error. "
                + "Couldn't find account");
        }
    } catch (java.sql.SQLException e) {
        throw new RemoteException("Database error. "
            + "Couldn't load account");
    } finally {
        try {
            if (resultSet != null)
                resultSet.close();
            if (prepStmt != null)
                prepStmt.close();
            if (dbConn != null)
                dbConn.disconnect();
        } catch (java.sql.SQLException e) {
            System.out.println("Unexpected exception while "
                + "closing resources");
        }
    }
}
```

Accessing Databases

Most entity beans work with databases. Always use the transaction and security management methods in `javax.ejb.deployment.ControlDescriptor` to manage transaction isolation level and transaction requirements. For more information about creating and managing transactions with beans, see Chapter 8, “Handling Transactions with EJBs”.

To work with data in the context of a bean-managed transaction, use JDBC. For more information about using JDBC to work with data, see Chapter 9, “Using JDBC for Database Access”.

Determining the Business Objects in Your Applications

When you adopt the NAS application programming model defined in this book, you split your application into general parts. For example, you separated your presentation logic from your business logic and transient business rules that you placed in session beans. You also determined which part(s) of your application were best handled as servlets, the central dispatcher for your application. What's left are the data and business objects around which your entire application revolves.

In general, business entities correspond to underlying database and document structures. So, for example, all corporations maintain extensive employee databases. An employee database is a perfect candidate for wrapping with a business entity to manage data access.

Choosing a Coarse Bean Granularity

By their very nature as business objects, entity beans tend to be larger and more monolithic than session beans. For developing database intensive applications, this is just as well, because fewer, larger objects improve the speed and performance of your distributed application.

As a bean developer, you can make choices that affect the application's speed and performance at bean design time. For example, to increase performance, we recommend that you create fewer, larger beans. You may also choose to gang database access and updates to minimize data traffic.

You should also work with your system administrator to anticipate how your application will be deployed and how many users will access the system at the same time. Armed with that information, you can make wiser choices about what your entity objects look like and how they will be shared within and across applications.

Handling Transactions with EJBs

This chapter describes the transaction support built into the Enterprise JavaBean programming model.

This chapter begins by introducing the EJB transaction model and explains the notion of container and bean managed transactions. Then it explains the semantics of all transaction attributes and use of the Java Transaction API for bean managed transactions. Finally, it discusses the restrictions on various combinations of EJB types and transaction attributes.

This chapter includes the following sections:

- Understanding the Transaction Model
- Specifying Transaction Attributes in an EJB
- Using Bean Managed Transactions
- Restrictions on Transaction Attributes
- Setting Isolation Levels

Understanding the Transaction Model

One of the primary advantages of using Enterprise JavaBeans (EJBs) is the support they provide for declarative transactions. In the declarative transaction model, attributes are associated with beans at deployment time and it is the container's responsibility, based on the attribute value, to demarcate and transparently propagate transactional context. The container is also responsible, in conjunction with a Transaction Manager, for ensuring that all participants in the transaction see a consistent outcome.

Declarative transactions free the programmer from explicitly demarcating transactions. They facilitate component-based applications where multiple components, potentially distributed and updating heterogeneous resources, can participate in a single transaction. The EJB specification also supports programmer-demarcated transactions using `javax.transaction.UserTransaction()`.

It is necessary to understand the distinction between global and local transactions in order to understand the NAS support for transactions. Global transactions are managed and coordinated by a Transaction Manager and can span multiple databases and processes. The Transaction Manager typically uses the XA protocol to interact with the database backends. Local transactions, on the other hand, are native to a database and are restricted within a single process. Local transactions can work against only a single backend. Local transactions are typically demarcated using JDBC APIs.

In NAS, all transactions started by the container or by using `javax.transaction.UserTransaction()` are global 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, you should configure and enable the corresponding NAS Resource Managers. See the *Administration Guide* for further details.

Specifying Transaction Attributes in an EJB

Transactional attributes can be specified on a bean-wide basis or on a per-method basis for a bean's remote interface. If attributes are specified at both levels, method-specific values take precedence over bean-wide values. These two should be mixed with care since some combinations are invalid as documented in the restrictions section.

Transactional attributes are specified as part of the bean's property file (see "Specifying Transaction Requirements" in Chapter 10, "Creating Configuration Files"). The following table lists all possible values of the attributes and their associated semantics.

Transaction Management Attribute	Purpose
<code>TX_BEAN_MANAGED</code>	The EJB or method explicitly controls transaction boundaries using the <code>javax.transaction.UserTransaction</code> interface. The container does not manage transactions for this bean.
<code>TX_MANDATORY</code>	The EJB container invokes the EJB method using the transaction context associated with the calling client. If the client has not started a transaction when the bean is invoked, the container throws the <code>TransactionRequired</code> exception.
<code>TX_NOT_SUPPORTED</code>	The container invokes the EJB without a transaction context. If a client calls with a transaction scope, the container suspends the association of the transaction with the current thread before delegating the method call to the EJB. When the bean's work is completed, the container resumes the transaction.
<code>TX_REQUIRED</code>	The container invokes the EJB using the client's transactional context, unless the client does not have one. If the client does not have a transaction in progress, the container automatically starts a transaction before delegating the method call to the bean. The container attempts to commit the transaction when the method call on the bean is completed, unless the transaction was marked for rollback or an exception was raised.
<code>TX_REQUIRES_NEW</code>	The EJB requires that the container start a new transaction when a method is invoked. The container attempts to commit the transaction when the bean's method call completes. When a client calls into a bean with this attribute, the container suspends the client's transaction before starting the bean's new transaction. Upon completing the bean's transaction, the container resumes the client's suspended transaction.
<code>TX_SUPPORTS</code>	The EJB mirrors the client's transaction scope. If the client has a transaction context, the EJB is invoked in that context. If not, the EJB is invoked without a transaction context.

Using Bean Managed Transactions

While it is preferable to use container-managed transactions, you may find that your application requirements necessitate the use of bean-managed transactions. To manage transactions programmatically, set the bean's transactional attribute to `TX_BEAN_MANAGED`. The application can use `javax.transaction.UserTransaction()`, which is made available by the container via `EJBContext`.

The following code snippet illustrates the usage pattern.

```
import javax.transaction.UserTransaction;
...
EJBContext ejbContext = ...;
...
UserTransaction tx = ejbContext.getUserTransaction();
tx.begin();
... // do work
tx.commit();
```

EJBs with attributes other than `TX_BEAN_MANAGED` are forbidden from using `javax.transaction.UserTransaction()`. For more information, see the Java specification for this interface at the following URL:

<http://java.sun.com/products/ejb/javadoc-1.1/javax/ejb/EJBContext.html>

Restrictions on Transaction Attributes

NAS prohibits certain combination of bean types and transaction attributes. The following combinations are not permitted.

- If you use the `TX_BEAN_MANAGED` transaction attribute anywhere in a bean, whether at the bean-wide level or for a method, all other methods in that bean must also employ the `TX_BEAN_MANAGED` transaction attribute. You cannot mix this attribute with other attributes at the method level.
- `TX_BEAN_MANAGED` and `TX_NOT_SUPPORTED` attributes are not supported for entity beans.

Setting Isolation Levels

NAS does not support declarative isolation levels. You should set your desired isolation level programmatically using the `setTransactionIsolation()` method in the JDBC API. For more information about isolation levels, see “Specifying Transaction Isolation Level” in Chapter 9, “Using JDBC for Database Access.”

Using JDBC for Database Access

This chapter describes how to use the Java Database Connectivity (JDBC) API for database access with Netscape Application Server.

This chapter provides high-level instructions for using the NAS implementation of JDBC in EJBs, and it indicates specific NAS resources affected by JDBC statements when those resources have clear programming ramifications. In NAS, Enterprise JavaBeans (EJBs) support database access primarily through the JDBC API. NAS supports all of JDBC 1.0 API as well as many of the emerging JDBC 2.0 extensions to JDBC, including result set enhancements, batch updates, distributed transactions, row sets, and JNDI support for datasource name lookups.

While this chapter assumes familiarity with JDBC 1.0, it also describes specific implementation issues that may have programming ramifications. For example, the JDBC specifications do not make it clear what constitute JDBC resources. In the specifications, some JDBC statements—such as any of the `Connection` class methods that close database connections—release resources without specifying exactly what those resources are.

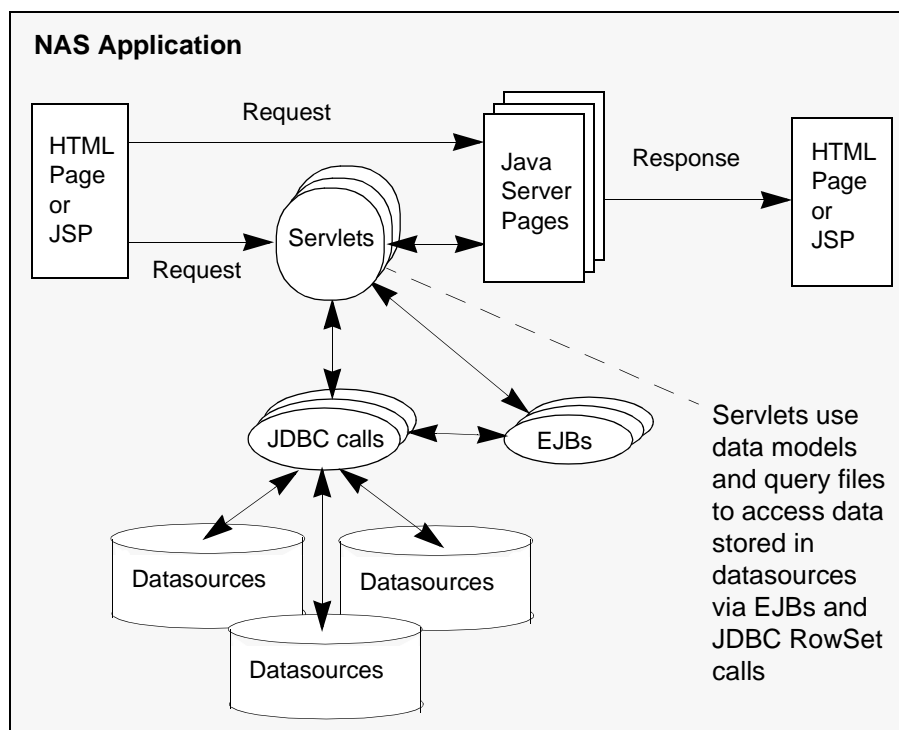
This chapter contains the following sections:

- Introducing JDBC
- Using JDBC in Server Applications
- Handling Connections
- Working with JDBC Features

Introducing JDBC

From a programming perspective, JDBC is a set of Java classes and methods that let you embed database calls in your server applications. That's all you need to know in order to start using JDBC in your server applications.

More specifically, JDBC is a set of interfaces that every server vendor, such as Netscape, must implement according to the JDBC specifications. NAS provides a JDBC type 2 driver which supports a variety of database backends. This driver processes the JDBC statements in your applications and routes the SQL arguments they contain to your database engines.



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

Supported Functionality

The JDBC specification is a broad, database-vendor independent set of guidelines that try to encompass the broadest range of database functionality possible in a simple framework.

At a minimum, JDBC assumes that all underlying database vendors support the SQL-2 database access language. Since its original inception, the JDBC specification has grown. It now has three parts:

- JDBC 1.0 describes the core set of database access and functionality that server vendors must implement in order to be JDBC compliant. The Netscape Application Server (NAS) fully meets this compliance standard. From the database vendor's perspective, JDBC 1.0 describes a database access model that permits full access to the standard SQL-2 language, those portions of the standard language each vendor supports, and those extensions to the language that each vendor implements.
- JDBC 2.0 describes additional database access and functionality. Much of this functionality involves support for newly-defined SQL-3 features, data types, and mappings. Other parts of JDBC 2.0 extend JDBC 1.0 features. The NAS implementation of JDBC supports most of the JDBC feature enhancements, but omits support for the new SQL-3 data types, such as blobs, clobs, and arrays, which many database vendors do not, as yet, fully support in their relational database management systems. The NAS JDBC implementation also omits support for SQL-3 data type mapping
- JDBC 2.0 Standard Extension API describes advanced support features, many of which offer the promise of improved database performance. The NAS implementation of JDBC currently supports JNDI and rowsets.

Understanding Database Limitations

When you start using JDBC in your server applications, you may encounter situations when you do not get the results you desire or expect. You may think the problem lies in JDBC or in the NAS implementation of the JDBC driver. In fact, the vast majority of the time, the problems actually lie in the limitations of your database engine.

Because JDBC covers the broadest possible spectrum of database support, it enables you to attempt operations not every database supports. For example, most database vendors support most of the SQL-2 language, but no vendor provides fully unqualified support for all of the SQL-2 standard. Most vendors built SQL-2 support on top of their already existing proprietary relational database management systems, and either those proprietary systems offer features not in SQL-2, or SQL-2 offers features not available in those systems. Most vendors, too, have added non-standard SQL-2 extensions to their implementations of SQL in order to support their proprietary features. JDBC provides ways for you to access vendor-specific features, but it's important to realize that these features may not be available for all the databases you use. This is especially true when you build an application that uses databases from two or more different vendors.

As a result, not all vendors fully support all aspects of every available JDBC class, method, and method arguments. More importantly, a set of SQL statements embedded as an argument in a JDBC method call may or may not be supported by the database or databases your server application uses. In order to make maximum use of JDBC, you must consult your database vendors' documentation about which aspects of SQL and JDBC they support. Before you call NAS technical support for a database problem, make sure you first eliminate the possibility that your database is the cause of the problem.

Understanding NAS Limitations

Like JDBC, NAS attempts to support the broadest possible spectrum of database engines and features. In some cases, NAS itself or the NAS JDBC driver may not fully support a particular feature of a database you use, or they may report incorrect information. If you can't access a database feature from your NAS application and you have eliminated the database as the source of the problem, check this section of the documentation and the product release notes to see if the problem you encounter is a documented NAS limitation. If not, document the problem fully and contact NAS 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 NAS JDBC driver. Almost all of these feature limitations apply to JDBC 2.0.

The following table lists JDBC features that are either partially or completely unsupported in NAS 4.0:

Feature	Limitation
<code>Connection.setTransactionIsolation</code>	Works only with isolation levels supported by your database vendors.
<code>Connection.getTypeMap</code>	Type maps are not supported.
<code>Connection.setTypeMap</code>	Type maps are not supported.
<code>Connection.cancel</code>	Works only with databases that support it.
<code>PreparedStatement.setObject</code>	Works only with simple data types.
<code>PreparedStatement.addBatch</code>	Works only with supported data manipulation statements that return a count of records changed.
<code>PreparedStatement.setRef</code>	References are not supported.
<code>PreparedStatement.setBlob</code>	Blobs are not supported. Use <code>setBinaryStream()</code> instead.
<code>PreparedStatement.setClob</code>	Clobs are not supported. Use <code>setBinaryStream()</code> instead.
<code>PreparedStatement.setArray</code>	Arrays are not supported. Use <code>setBinaryStream()</code> instead.
<code>PreparedStatement.getMetaData</code>	Not all databases return complete information.
<code>CallableStatement.getObject</code>	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.
<code>CallableStatement.getRef</code>	References are not supported.
<code>CallableStatement.getBlob</code>	SQL3-style blobs are not supported.
<code>CallableStatement.getClob</code>	SQL3-style clobs are not supported.
<code>CallableStatement.getArray</code>	Arrays are not supported.
<code>CallableStatement</code>	Updatable <code>ResultSet</code> is not supported.
<code>ResultSet.setCursorName</code>	Behavior differs depending on database: <ul style="list-style-type: none"> For Oracle, if user does not specify a cursor name with <code>SetCursorName</code>, an empty string is returned. For Sybase, if the result set is not updatable, a cursor name is automatically generated by NAS. Otherwise an empty string is returned. For ODBC, Informix, and DB2, the driver returns a cursor name if none is specified.
<code>ResultSet.getObject</code>	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.

Feature	Limitation
<code>ResultSet.updateObject</code>	Works only with scalar types.
<code>ResultSet.getRef</code>	References are not supported.
<code>ResultSet.getBlob</code>	SQL3-style blobs are not supported.
<code>ResultSet.getClob</code>	SQL-style clobs are not supported.
<code>ResultSet.getArray</code>	Arrays are not supported.
<code>ResultSetMetaData.getTableName</code>	Returns an empty string for non-ODBC database access.
<code>DatabaseMetaData.getUDTs</code>	Not supported.

For more information about working with `ResultSet`, `ResultSetMetaData`, and `PreparedStatement`, see the appropriate sections later in this chapter.

Supported Databases

NAS currently connects to many different relational databases. The following tables lists the most commonly used databases that are supported.

Database	Notes
Oracle	Support is offered through the Oracle OCI interface. Both Oracle 7 and Oracle 8 database instances are supported in a fully multi-threaded environment. NAS also coexists with all Oracle RDBMS tools and utilities, such as SQL*Plus, Server Manager, and Oracle Backup.
Informix	Support is offered through Informix CLI interface. Both Informix Online Dynamic Server and Informix Universal Server are supported.
Sybase	Support is offered through Sybase CTLIB. Sybase versions 10 and 11 are supported.
Microsoft SQLServer	Support is offered through the Microsoft ODBC interface. Microsoft SQLServer on Windows NT only is supported.
DB2	Support is offered through the DB2 CLI client interface. DB2 versions 2.1 and 5.2 are supported.
ODBC	NAS does not specifically support or certify any ODBC 2.0 or 3.0 compliant driver set, though they may work.

Both because the databases supported by NAS are constantly updated, and because database vendors consistently upgrade their products, you should always check with NAS technical support for the latest database support information.

Using JDBC in Server Applications

JDBC is part of the NAS run-time environment. In theory, this means that JDBC is always available to you any time you use Java to program an application. In a typical multi-tiered server application, it is theoretically possible to use JDBC to access a database backend from the client, from the presentation layer, in servlets, and in Enterprise Java Beans (EJBs).

In practice, however, it usually makes sense—for security and portability reasons—to restrict database access to the middle layers of a multi-tiered server application. In the NAS programming model, this means placing all JDBC calls in servlets and EJBs, with a strong preference toward EJBs.

There are two reasons for this programming preference. One is that placing all JDBC calls inside EJBs makes your application more modular and more portable. Another is that EJBs provide built-in mechanisms for transaction control. Placing JDBC calls in well-designed EJBs frees you from programming explicit transaction control using JDBC or `java.transaction.UserTransaction` that provide low-level transaction support under JDBC.

Note Make sure to use a globally available datasource to create a global (bean-wide) connection so that the EJB transaction manager can control the transaction.

Using JDBC in EJBs

Placing all your JDBC calls in EJBs ensures a high degree of server application portability. It also frees you from having to manage transaction control with explicit JDBC calls unless you so desire. Because EJBs are components, you can use them as building blocks for many applications with little or no recoding, and maintain a common interface to your database backends.

Managing Transactions with JDBC or `java.transaction.UserTransaction`

Using the EJB transaction attribute property to manage transactions is recommended, but not mandatory. There may be times when explicit coding of transaction management using JDBC or `java.transaction.UserTransaction` is appropriate for your application. In these cases, you code transaction management in the bean yourself. Using an explicit transaction in an EJB is called a bean-managed transactions.

Transactions can be local to a specific method (method-specific) or they can encompass the entire bean (bean-wide).

There are two steps for creating a bean managed transaction:

1. Set the transaction attribute property of the EJB to `TX_BEAN_MANAGED` in the bean's deployment descriptor.
2. Code the appropriate JDBC or transaction management statements in the bean, including statements to start the transaction, and to commit it or roll it back.

It is an error to code explicit transaction handling in EJBs for which the transaction attribute property is not `TX_BEAN_MANAGED`. For more information about handling transactions with JDBC, see the JDBC 1.0 API specification.

Specifying Transaction Isolation Level

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

Transaction isolation levels are defined as follows:

Transaction Isolation Level	Description
<code>TRANSACTION_NONE</code>	Transactions are not supported. Only used with <code>Connection.getTransactionIsolation()</code>
<code>TRANSACTION_READ_COMMITTED</code>	Dirty reads are prevented; non-repeatable reads and phantom reads can occur.

Transaction Isolation Level	Description
TRANSACTION_READ_UNCOMMITTED	Dirty reads, non-repeatable reads and phantom reads can occur.
TRANSACTION_REPEATABLE_READ	Dirty reads and non-repeatable reads are prevented; phantom reads can occur.
TRANSACTION_SERIALIZABLE	Dirty reads, non-repeatable reads and phantom reads are prevented.

Before you specify a transaction isolation level for a bean, make sure the level is supported by your relational database management system. Not all databases support all isolation levels. You can test your database programmatically by using the method `supportsTransactionIsolationLevel()` in `java.sql.DatabaseMetaData`, as in the following example:

```
java.sql.DatabaseMetaData db;
if (db.supportsTransactionIsolationLevel(TRANSACTION_SERIALIZABLE) {
    Connection.setTransactionIsolation(TRANSACTION_SERIALIZABLE);
}
```

For more information about these isolation levels and what they mean, see the Java Database Connectivity (JDBC) 1.0 API Specification. All specifications are accessible from `installdir/nas/docs/index.htm`, where `installdir` is the location in which you installed NAS.

Using JDBC in Servlets

Servlets are at the heart of your NAS server applications. They stand between a client interface, such as an HTML page on a browser, the JSP that generated the HTML, and the EJBs that do the bulk of your application's work.

NAS applications use JDBC embedded in EJBs for most database access. This is the preferred arrangement for database access using NAS because it enables you to take advantage of the transaction control built into EJBs and their containers. Servlets, however, can also provide database access through embedded JDBC.

In some situations, accessing a database directly from a servlet can offer a speed advantage over accessing databases from EJBs. There is less call overhead, especially if your application is spread across servers so that your EJBs are accessible only through the Java Remote Method Interface (RMI). Use

direct database service through servlets sparingly. If you do provide database access from servlets, restrict access to those situations where access is very short duration, the transaction is read-only, and you can take advantage of the new JDBC 2.0 rowset class.

If you choose to access a database from a servlet, use the new JDBC 2.0 rowset interface to interact with a database. A rowset is a Java object that encapsulates a set of rows that have been retrieved from a database or other tabular datasource, such as a spreadsheet. The rowset interface provides JavaBean properties that allow a rowset instance to be configured to connect to a datasource and retrieve a set of rows. For more information about working with rowsets, see “Working with Rowsets” on page 169.

Handling Connections

NAS implements the JDBC 2.0 compliant interface `java.sql.Connection`. The behavior of the connection depends on whether it is local or global.

Local Connections

A `Connection` object is called a local connection if its transaction context is not managed by an EJB container. The transaction context in a local connection can not propagate across processes or across datasources; it is local to the current process and to the current datasource.

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

Registering a Local Datasource

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

You can register the datasource by first creating a properties file for the datasource. Next, register the properties file with NAS using the Administration Tool or the provided utility `dsReg`. `dsReg` takes as arguments the datasource name and the name of a properties file describing the datasource. `dsReg` does not overwrite existing entries unless you specify the `-force` option.

For example, to register a datasource called `SampleDS` which connects to Oracle database using the user name `kdemo`, password `kdemo`, database `ksample` and server `ksample`, create a properties file like the following, and name it `SampleDS.props`:

```
DataBase=ksample
DataSource=ksample
UserName=kdemo
PassWord=kdemo
DriverType=ORACLE_OCI
```

You can then use this properties file to register the datasource with the following command:

```
dsReg 'jdbc/SampleDS' SampleDS.props
```

For more information about datasource properties files, see “Creating Datasource Property Files” in Chapter 10, “Creating Configuration Files.” For more information about the NAS Administration Tool, see the *Administration Guide*.

Global Connections

A `Connection` object is called a global connection if its transaction context is managed by EJB container. The transaction context in a global connection can be propagated across datasources. The transaction context is managed implicitly by the EJB container for container-managed transactions, or explicitly in case of bean-managed transactions. For more information about transactions, see Chapter 8, “Handling Transactions with EJBs.”

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

Using Resource Managers

The collection of datasources in which a global transaction participates is known as a resource manager. All resources managers needs to be registered with NAS and be enabled so that they participate in global transactions. Resource managers can be set up at install time or they can also be set up using the NAS Administration Tool (see the *Administration Guide*). A global connection must be associated with a resource manager.

Registering a Global Datasource

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

You can register the datasource by creating a properties file for the datasource, including a resource manager's name. Next, register the properties file with NAS using the Administration Tool or the provided utility `dsReg`. `dsReg` takes as arguments the datasource name and the name of a properties file describing the datasource. `dsReg` does not overwrite existing entries unless you specify the `-force` option.

For example, to register a datasource called `GlobalSampleDS` which connects to Oracle database using the username `kdemo`, password `kdemo`, database `ksample` and server `ksample`, create a properties file like the following, and name it `GlobalSampleDS.props`:

```
DataBase=ksample
DataSource=ksample
UserName=kdemo
PassWord=kdemo
DriverType=ORACLE_OCI
ResourceMgr=ksample_rm
```

You can then use this properties file to register the datasource with the following command:

```
dsReg 'jdbc/GlobalSampleDS' GlobalSampleDS.props
```

For more information about datasource properties files, see “Creating Datasource Property Files” in Chapter 10, “Creating Configuration Files.” For more information about the NAS Administration Tool, see the *Administration Guide*.

Creating a Global Connection

The following code demonstrates how a global connection can be obtained from a datasource, and how that global connection can be used in a bean managed transaction.

```
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);
}
```

Working with JDBC Features

While this chapter is not a JDBC primer, it does introduce you to using JDBC in EJBs with NAS 4.0. The following sections describe various JDBC interfaces and classes that have either have special requirements in the NAS environment, or that are new JDBC 2.0 features that you are especially encouraged to use when developing NAS server applications.

For example, “Working With Connections” describes what resources NAS releases when a connection is closed because that information differs among different JDBC implementations. On the other hand, “Pooling Connections” and “Working with Rowsets” offer more extensive coverage because these are new JDBC 2.0 features that offer increased power, flexibility, and speed for your server applications.

Working With Connections

When you open a connection in JDBC, NAS allocates resources for the connection. If you call `Connection.close()` when a connection is no longer needed, the connection resources are freed. Always reestablish connections before continuing database operations after you call `Connection.close()`.

You can use `Connection.isClosed()` to test whether the connection is closed. This method returns `false` if the connection is open, and returns `true` only after `Connection.close()` is called.

You can determine if a database connection is invalid by catching the exception that is thrown when a JDBC operation is attempted on a closed connection.

Finally, opening and closing connections is an expensive operation. If your application uses several connections, and if connections are frequently opened and closed, NAS automatically provides connection pooling. Connection pooling provides a cache of connections that are automatically closed when necessary.

Note Connection pooling is an automatic feature of NAS; the API is not exposed.

setTransactionIsolation

Not all database vendors support all levels of transaction isolation available in JDBC. NAS permits you to specify any isolation level your database supports, but throws an exception against values your database does not support. For more information, see “Specifying Transaction Isolation Level” on page 156.

getTypeMap, setTypeMap

The NAS implementation of the JDBC driver does not support type mapping, a new SQL-3 feature that most database vendors also do not support. The methods exist, but currently do nothing.

cancel

`cancel()` is supported for all databases.

Pooling Connections

Two of the costlier database operations you can execute in JDBC are for creating and destroying database connections. Connection pooling permits a single connection cache to be used for connection requests. When you use connection pooling, a connection is returned to the pool for later reuse without actually destroying it. A later call to create a connection merely retrieves an available connection from the pool instead of actually creating a new one.

NAS automatically provides JDBC connection pooling wherever you make JDBC calls.

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

NAS 4.0 supports concurrency for `FORWARD-ONLY READ-ONLY` result sets. On callable statements, NAS 4.0 also supports concurrency for `FORWARD-ONLY UPDATABLE` result sets.

NAS also supports concurrency for `SCROLL-INSENSITIVE READ-ONLY` result sets.

`SCROLL-SENSITIVE` concurrency is not supported.

Updatable Result Set Support

In NAS 4.0, creation of updatable result sets is restricted to queries on a single table. The `SELECT` query for an updatable result set must include the `FOR UPDATE` clause:

```
SELECT...FOR UPDATE [OF column_name_list]
```

Note You can use join clauses to create read-only result sets against multiple tables; these result sets are not updateable.

For Sybase, the select list must include a unique index column. Sybase also permits you to call `execute()` or `executeQuery()` to create an updatable result set, but the statement must be then be closed before you can execute any other SQL statements.

To use an updatable result set with Oracle 8, you must wrap the result set query in a transaction:

```
conn.setAutoCommit(false);
ResultSet rs =
    stmt.executeQuery("SELECT...FOR UPDATE...");
...
rs.updateRows();
...
conn.commit();
```

For Microsoft SQL Server, if concurrency for a result set is `CONCUR_UPDATABLE`, the `SELECT` statement in the `execute()` or `executeQuery()` methods must not include the `ORDER BY` clause.

getCursorName

One method of `ResultSet`, `getCursorName()`, enables you to determine the name of the cursor used to fetch a result set. If a cursor name is not specified by the query itself, different database vendors return different information. NAS attempts to handle these differences as transparently as possible. The following table indicates the name of a cursor returned by different database vendors if no cursor name is specified in the initial query

Database Vendor	getCursorName Value Returned
Oracle	If a cursor name is not specified with <code>setCursorName()</code> , an empty string is returned.
Sybase	If a cursor name is not specified with <code>setCursorName()</code> , and the result set is not updatable, a unique cursor name is automatically generated by NAS. Otherwise an empty string is returned.
Informix, DB2, ODBC	If a cursor name is not specified with <code>setCursorName()</code> , the driver automatically generates a unique cursor name.

getObject

NAS implements this JDBC method in a manner that only works with scalar data types. JDBC 2.0 adds additional versions of this method that include a map argument. NAS does not implement maps, and ignores the map argument if it is supplied.

getRef, getBlob, getClob, and getArray

References, blobs, clobs, and arrays are new SQL-3 data types. NAS does not implement these data objects or the methods that work with them. You can, however, work with references, blobs, clobs, and arrays using `getBinaryStream()` and `setBinaryStream()`.

Working with ResultSetMetaData

The `getTableName()` method only returns meaningful information for ODBC-compliant databases. For all other databases, this method returns an empty string.

Working with PreparedStatement

`PreparedStatement` is a class that encapsulates a query, update, or insert statement that is used repeatedly to fetch data. Be aware of the following behaviors or limitations associated with this class.

Note You can use the NAS feature `SqlUtil.loadQuery()` to load a `NASRowSet` with a prepared statement. For more information, see the entry for the `SqlUtil` class in the *NAS Foundation Class Reference*.

setObject

This method may only be used with scalar data types.

addBatch

This method enables you to gang a set of data manipulation statements together to pass to the database as if it were a single statement. `addBatch()` only works with SQL data manipulation statements that return a count of the

number of rows updated or inserted. Contrary to the claims of the JDBC 2.0 specification, `addBatch()` does not work with any SQL data definition statements such as `CREATE TABLE`.

setRef, setBlob, setClob, setArray

References, blobs, clobs, and arrays are new SQL-3 data types. NAS does not implement these data objects or the methods that work with them. You can, however, work with references, blobs, clobs, and arrays using `getBinaryStream()` and `setBinaryStream()`.

getMetaData

Not all database systems return complete metadata information. See your database vendor's documentation to determine what kind of metadata your database provides to clients.

Working with CallableStatement

`CallableStatement` is a class that encapsulates a database procedure or function call for databases that support returning result sets from stored procedures. Be aware of the following limitation associated with this class. The JDBC 2.0 specification states that callable statements can return an updatable result set. This feature is not supported in NAS.

getRef, getBlob, getClob, getArray

References, blobs, clobs, and arrays are new SQL-3 data types. NAS does not implement these data objects or the methods that work with them. You can, however, work with references, blobs, clobs, and arrays using `getBinaryStream()` and `setBinaryStream()`.

Handling Batch Updates

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

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

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

In order to use batch updates, your application must disable auto commit options:

```
...
// turn off autocommit to prevent each statement from committing
separately
con.setAutoCommit(false);

Statement stmt = con.createStatement();

stmt.addBatch("INSERT INTO employees VALUES(4671, 'James Williams')");
stmt.addBatch("INSERT INTO departments VALUES(560, 'Produce')");
stmt.addBatch("INSERT INTO emp_dept VALUES( 4671, 560)");

//submit the batch of updates for execution
int[] updateCounts = stmt.executeBatch();
con.commit();
```

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

Note The JDBC 2.0 specification erroneously implies that batch updates can include data definition language (DDL) statements such as `CREATE TABLE`. DDL statements do not return a simple update count, and so cannot be grouped for a batch operation. Also, some databases do not allow data 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 into the NAS 4.0 EJB container, so if you use EJBs that do not specify the `TX_BEAN_MANAGED` transaction attribute, you get automatic support for distributed transactions in your application.

In servlets and in EJBs that specify the `TX_BEAN_MANAGED` transaction attribute, you can still use distributed transactions, but you must manage transactions using the `JTS UserTransaction` class. For example:

```
InitialContext ctx = null;
String dsName1 = "jdbc/SampleDS1";
String dsName2 = "jdbc/SampleDS2";
DataSource ds1 = null;
DataSource ds2 = null;

    try {
        ctx = new InitialContext();
        ds1 = (DataSource)ctx.lookup(dsName1);
        ds2 = (DataSource)ctx.lookup(dsName2);

    } catch(Exception e) {
        e.printStackTrace(System.out);
    }

UserTransaction tx = ejbContext.getUserTransaction();

tx.begin();

Connection conn1 = ds1.getConnection();
Connection conn2 = ds2.getConnection();

// do some work here

tx.commit();
```

In this example, `ds1` and `ds2` must be registered with NAS as global datasources. In other words, their `datasource` properties files must include a `ResourceMgr` entry whose value must be configured at install time.

For example a global database properties file looks like:

```
DataBase=ksample
DataSource=ksample
UserName=kdemo
Password=kdemo
DriverType=ORACLE_OCI
ResourceMgr=orarm
```


In this example, `orarm` must be a valid `ResourceMgr` entry and must be enabled to get a global connection successfully. In order to be a valid `ResourceMgr` entry, an resource manager must be listed the registry in `CCS0\RESOURCEMGR`, and the entry itself must have the following properties.

```
DatabaseType (string key)
IsEnabled (integer type)
Openstring ( string type key)
ThreadMode ( string type key)
```

Working with Rowsets

A rowset is an object that encapsulates a set of rows retrieved from a database or other tabular data store, such as a spreadsheet. To implement a rowset, your code must import `javax.sql`, and implement the `RowSet` interface. `RowSet` extends the `java.sql.ResultSet` interface, permitting it to act as a JavaBeans component.

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

You fill a rowset by calling the `RowSet.execute()` method. The `execute()` method uses property values to determine the datasource and retrieve data. The actual properties you must set and examine depends upon the implementation of `RowSet` you invoke.

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

Using NASRowSet

NAS 4.0 provides a rowset class called `NASRowSet`. `NASRowSet` extends `ResultSet`, so call methods are inherited from the `ResultSet` object. `NASRowSet` overrides the `getMetaData()` and `close()` methods of `ResultSet`.

The `RowSet` interface is fully supported except as noted in the following table.

Method	Argument	Exception Thrown	Reason
<code>setReadOnly()</code>	<code>false</code>	<code>SQLException</code>	<code>NASRowSet</code> is already read-only.
<code>setType()</code>	<code>TYPE_SCROLL_INSENSITIVE</code>	<code>SQLException</code>	<code>SCROLL_INSENSITIVE</code> is not supported.
<code>setConcurrency()</code>	<code>CONCUR_UPDATABLE</code>	<code>SQLException</code>	<code>NASRowSet</code> is read-only.
<code>addRowSetListener()</code>	any	None	Not supported.
<code>removeRowSetListener()</code>	any	None	Not supported.
<code>setNull()</code>	any type name	Arguments ignored	Not supported.
<code>setTypeMap()</code>	<code>java.util.Map</code>	None	Map is a JDBC 2.0 feature that is not currently supported.

RowSetReader

`NASRowSet` provides a full implementation of the `RowSetReader` class.

RowSetWriter

`NASRowSet` is read-only, but an interface for this class is provided for future expansion. At present, its only method, `writeData()` throws `SQLException`.

RowSetInternal

This internal class is used by `RowSetReader` to retrieve information about the `RowSet`. It has a single method, `getOriginalRow()`, which returns the original result set instead of a single row.

Using CachedRowSet

The JDBC specification provides a rowset class called `CachedRowSet`. `CachedRowSet` permits you to retrieve data from a datasource, then detach from the datasource while you examine, and modify the data. A cached rowset

keeps track both of the original data retrieved, and any changes made to the data by your application. If the application attempts to update the original datasource, the rowset is reconnected to the datasource, and only those rows that have changed are merged back into the database.

Creating a RowSet

There are two ways to create a rowset in a NAS server application:

To create a NAS-dependent rowset:

```
NASRowSet rs = new NASRowSet();
```

To create a NAS-independent rowset:

Look up the class that implements the RowSet interface using
InitialContext():

```
InitialContext ctx = new InitialContext();
RowSet rs = (javax.sql.RowSet) ctx.lookup("javax.sql.RowSet");
```

Using JNDI

JDBC 2.0 specifies that you can use the Java Naming and Directory Interface (JNDI) to provide a uniform, platform and JDBC vendor-independent way for your applications to find and access remote services over the network. For example, all JDBC driver managers, such as the JDBC driver manager implemented in NAS 4.0, must find and access a JDBC driver by looking up the driver and a JDBC URL for connecting to the database. For example:

```
Class.forName("SomeJDBCDriverClassName");

Connection con =
    DriverManager.getConnection("jdbc:NAS_subprotocol:machineY:portZ");
```

This code illustrates how a JDBC URL may not only be specific to a particular vendor's JDBC implementation, but also to a specific machine and port number. Such hard-coded dependencies make it hard to write portable applications that can easily be shifted to different JDBC implementations and machines at a later time.

In place of this hard-coded information, JNDI permits you to assign a logical name to a particular datasource. Once you establish the logical name, you need only modify it a single time to change the deployment and location of your application.

JDBC 2.0 specifies that all JDBC datasources should be registered in the `jdbc` naming subcontext of a JNDI namespace, or in one of its child subcontexts. The JNDI namespace is hierarchical, like a file system's directory structure, so it is easy to find and nest references. A datasource is bound to a logical JNDI name. The name identifies a subcontext, `"jdbc"`, of the root context, and a logical name. In order to change the datasource, all you need to do is change its entry in the JNDI namespace without having to modify a line of code in your application.

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

Creating Configuration Files

This chapter describes how to create, edit, and maintain configuration files for the application as a whole and for individual components (EJBs and servlets).

The following topics are presented in this chapter:

- Introducing Configuration Files
- Creating Servlet and Application Configuration Files
- Creating EJB Property Files
- Creating Datasource Property Files

Introducing Configuration Files

You use configuration files in NAS to describe metadata about the individual components (servlets and EJBs) that make up your application. The information in each configuration file is stored in a registry internal to NAS.

Each application component must have a configuration file associated with it. Servlets use a plain-text format called NTV. Additionally, each application component must be associated with a globally unique identifier, or GUID.

The NAS Registry

The NAS 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 NAS gains access to servlets, EJBs, and other application resources is called registration, because it involves placing entries in the NAS registry for each item.

You can change some of the information in the registry at run-time using the NAS Administration Tool. For more information about the registry, and the Administration Tool, see the *Administration Guide*.

In particular, see “Deploying Application Files Manually” in Chapter 10, “Creating Configuration Files” in the *Administration Guide*. This section describes how to register servlets, EJBs, and JDBC datasources manually using provided utilities.

GUIDs

A GUID is a 128-bit hexadecimal number in the following format:

```
{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}
```

GUIDs are guaranteed to be globally unique, which makes them ideal for identifying components in a large-scale heterogeneous system such as a NAS application.

GUIDs are normally assigned automatically by the deployment tool, or by Netscape Application Builder (NAB). You can generate a GUID manually by using a utility named `kguidgen`. `kguidgen` is installed by default into the directory `BasePath/bin`. That directory must be listed in your search path (your `PATH` environment variable) in order to generate a GUID.

To generate a new GUID, simply run `kguidgen` from a command line or window.

Creating Servlet and Application Configuration Files

You must provide a configuration for each servlet in order to identify it to NAS and to set up some. A single configuration file can configure multiple servlets. Additionally, each application must have a configuration file that references all servlet configuration files, identifies the application's name, and controls how sessions work in the application.

Servlet and application configuration files are hierarchical lists of name-value pairs with ordinal types. The format is called NTV (name-type-value). Each NTV file must contain certain fields in order to properly register the servlet with NAS.

NTV Syntax

The NTV format consists of predefined variable names, their ordinal types, and their values, which depend on the type. In general, an NTV list is surrounded in curly braces { } and delimited with commas. Each entry in the list consists of a name, a type, a value, and a trailing comma. Non-keywords, including strings and numerical values, must be surrounded in double-quotes. String arrays are surrounded with square brackets [].

Valid types and syntax are shown below:

Type	Description
Bool	Boolean value: true or false
Date	Date (various standard formats are accepted)
Double	Double-precision floating point value
Int	Integer
NTV	A Name-Type-Value list. Use variables of this type to nest lists.
Str	String. Surrounded in double-quotes.
StrArr	An array of strings, each of which is surrounded in double-quotes. The array is delimited with commas. The entire array is surrounded by square brackets [].

Note Do not use `:` (the colon character) in NTV attribute values.

Manually Registering Servlets to NAS

Normally, servlets are registered to the server registry file upon deployment. In some cases (such as for testing), you may want to register your servlets manually. NAS provides a registration script called `servletReg` for this purpose.

You can use `servletReg.sh` as follows on Solaris:

```
servletReg.sh -i appInfo.ntv
```

This example shows `servletReg.bat` under Windows NT:

```
servletReg.bat /i appInfo.ntv
```

For more information, see “Manually Deploying Servlets and JSPs” in Chapter 10, “Creating Configuration Files”

Uploading Configuration File Changes

If you change either of the following fields, you must stop the server and re-register your configuration files in order to make the changes active:

- Any changes to the `ServletRegistryInfo` section of a servlet configuration file
- Any change to the `SessionInfo` section of an application configuration file `appInfo.ntv`

Changes to other fields in application or servlet configuration files do not require a server restart.

To load a new servlet configuration file into NAS, stop the server and redeploy the configuration file, then execute `servletReg appInfo.ntv` on the server to register the new configuration file. Finally, restart the server.

Application Configuration Information

Create an application information file named `appInfo.ntv` for each application. Only one application can be configured in a given application configuration file. `appInfo.ntv` normally resides in `AppPath/appName/ntv`. For more information, see “Important Servlet Files and Locations” in Chapter 3, “Controlling Applications with Servlets.”

Application Configuration File Layout

This pseudocode example shows all the available fields in an application configuration file. Items in **bold** are required. Items in *italics* indicate defaults. Ellipses (. . .) indicate data that you must supply. These fields are described in detail in the following sections.

```
NTV-ASCII {
    "SessionInfo"      NTV      {
        "timeout"      Int       "1800",
        "flags"         StrArr   [ "SESSION_DISTRIB", "SESSION_TIMEOUT_CREATE" ],
        "sessionManagment" Int    "0",
    },

    "ServletFiles"      StrArr   [...],

    "AppName"           Str       "...",
}
```

Application Configuration File Fields

Each `appInfo.ntv` file can contain the following fields:

Name	Type	Value
<code>SessionInfo</code>	NTV	Optional session settings for this application. The default is a distributable session with default behavior. This section is required for a local session (in order to specify the session type) and for any non-default behavior in a distributable session.

Name	Type	Value
AppName	Str	Semi-optional. Name of the application, required if the configuration file is associated with a particular application. Do not configure this field for the generic application. This name can not be hierarchical (a/b/c).
ServletFiles	StrArr	Required. Names of all the servlet configuration files associated with this application.

Specifying Session Information

SessionInfo

The `SessionInfo` section consists of several variables, which are applied to all servlets in the same application. None of these fields is required; if they are not provided, default values are used. All of these fields except `sessionManagement` are valid only for distributable sessions.

For more information about sessions, see Chapter 11, “Creating and Managing User Sessions.”

The `SessionInfo` section consists of the following attributes:

Name	Type	Value
secure	Bool	Optional. If set to <code>true</code> , the session cookie is only sent over secure connections (e.g. HTTPS).
timeout	Int	Optional. Duration (in seconds) after which the session expires. The default timeout is 1800 seconds (30 minutes). Note: for local sessions, set the timeout value at run time using <code>setMaxInactiveInterval()</code> .
flags	StrArr	Optional. Configures the scope of the distributable session to be created (see below for details).
sessionManagement	Int	Optional. Specifies session type for this application. 0 (the default) indicates a distributable session, 1 indicates a local session.

The `flags` field can contain up to two flags, one indicating the session's scope and the other indicating expiration behavior. The following values are valid:

Scope Flag	Description
<code>SESSION_LOCAL</code>	Makes the session visible to objects in the local process only.
<code>SESSION_CLUSTER</code>	Makes the session visible to objects within the cluster.
<code>SESSION_DISTRIB</code>	Makes the session visible to objects on all Netscape Application Servers (i.e., including those outside the cluster).

Expiration Flag	Description
<code>SESSION_TIMEOUT_ABSOLUTE</code>	Specifies that the session expires at a specific date and time. NOTE: this flag is not supported in this release.
<code>SESSION_TIMEOUT_CREATE</code>	Specifies that the session expires a certain duration after the session was created. Specify the duration using the <code>timeout</code> field.

If you specify nothing in the `SessionInfo` section, the default is a distributable session with `SESSION_DISTRIB` and `SESSION_TIMEOUT_CREATE` with a default of 1800 seconds (30 minutes), as explicitly described in the following example:

```
"SessionInfo"      NTV      {
    "secure"        Bool      "false",
    "timeout"        Int       "1800",
    "flags"          StrArr    ["SESSION_DISTRIB", "SESSION_TIMEOUT_CREATE"],
    "sessionManagment" Int      "0",
},
...
```

Specifying the Application's Name

AppName

The `AppName` entry contains the name of the application, which must conform to URL standards for naming as this name is specified when invoking a servlet that is part of this application. For example:

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

This field is required only for applications that have servlets which are part of an application. For the generic application, this field should *not* be specified.

For more information about invoking servlets, see “Invoking Servlets” in Chapter 3, “Controlling Applications with Servlets.”

Specifying Servlet Configuration Files

ServletFiles

The `ServletFiles` entry consists of an array of strings, where each string is the name of a servlet configuration file without the `.ntv` extension. At least one servlet configuration file must be configured. This is a required field.

Since they are identified by name in `appInfo.ntv`, and since they reside in the same directory as `appInfo.ntv`, there are no restrictions on how a servlet configuration file is named. By convention, servlet configuration files are named for the servlets they describe. For example:

```
"ServletFiles"      StrArr  ["myServlet", "myOtherServlet"],
```

For more information about servlet configuration files, see “Servlet Configuration Information” on page 181.

Example Application Configuration File

The following is an example `appInfo.ntv` that creates a distributable session and configures two servlet info files, `servInfo1.ntv` and `servInfo2.ntv`:

```
NTV-ASCII {
  "SessionInfo"      NTV      {
    "timeout"         Int      "400",
    "flags"           StrArr   ["SESSION_DISTRIB"],
    "sessionID"       Str      "123456789",
    "sessionManagement" Int     "0",
  },
  "ServletFiles"     StrArr   ["servInfo1", "servInfo2"],
  "AppName"          Str      "BookStoreDemo",
}
```

(For examples of `servInfo1.ntv` and `servInfo2.ntv`, see “Example Servlet Configuration Files” on page 188.)

Servlet Configuration Information

Create a servlet configuration for each servlet. A servlet configuration resides in a servlet configuration file, which can contain configurations for any number of servlets. Servlet configuration files can be named anything, so long as the name is listed in the `ServletFiles` attribute in `appInfo.ntv` (see “Application Configuration Information” on page 177).

Servlet configuration files for servlets that are part of an application reside in `AppPath/appName/ntv` alongside the application’s configuration file `AppPath/appName/ntv/appInfo.ntv`. `AppPath` refers to the value for the registry variable `AppPath`, the base file location for NAS. `appName` refers to the application’s name, as configured in `appInfo.ntv`.

Servlet configuration files for generic servlets that do not belong to a particular application reside in `AppPath/ntv` alongside the generic application configuration file `AppPath/ntv/appInfo.ntv`. Generic servlet configuration files must be listed in `AppPath/ntv/appInfo.ntv`.

Dynamically Reloading Servlets

If you change either of the following fields, you must stop the server and re-register your configuration files in order to make the changes active:

- Any changes to the `ServletRegistryInfo` section of a servlet configuration file
- Any change to the `SessionInfo` section of an application configuration file `appInfo.ntv`

Changes to other fields in application or servlet configuration files do not require a server restart.

To load a new servlet configuration file into NAS, stop the server and redeploy the configuration file, then execute `servletReg appInfo.ntv` on the server to register the new configuration file. Finally, restart the server.

See “Manually Registering Servlets to NAS” on page 176.

Servlet Configuration File Layout

This pseudocode example shows all of the available fields in a servlet configuration file. Items in **bold** are required. Items in *italics* indicate defaults. Ellipses (. . .) indicate data that you must supply. These fields are described in detail in the following sections.

```
NTV-ASCII {
    "... "      NTV      { (indicate servlet filename without the .class extension)

        "ServletRegistryInfo" NTV      {
            "type"                Str      "j",
            "enable"              Str      "y",
            "encrypt"             Str      "n",
            "lb"                  Str      "y",
            "guid"                Str      "{...}",
            "serverInfo"          StrArr   ["..."],
            "acl"                  StrArr   ["..."],
        },

        "ServletRunnerInfo"      NTV      {
            "ServletClassPath"    Str      "...",
        },

        "ServletData"            NTV      {
            "CreateDate"          Str      "...",
            "Description"          Str      "...",
            "CacheTimeOut"        Int      "...",
            "CacheSize"           Int      "...",
            "CacheCriteria"       Str      "...",
            "RequiredSessionVars" StrArr   ["..."],
            "Parameters"          NTV      {
                "... "            NTV      {
                    "inputRequired" Str      "n",
                    "inputRule"     Str      "...",
                },
            },
        },

        "FormActionHandlers"     NTV      {
            "... "                Str      "...",
        },
    },

    "InitArgs"                    NTV      {
        "... "                    Str      "...",
    },
}
```

Servlet Configuration File Fields

A servlet configuration file has several sections nested under each servlet name.

Name	Type	Value
<code>ServletRegistryInfo</code>	NTV	Required. Information needed to register a servlet with NAS.
<code>ServletRunnerInfo</code>	NTV	Required. CLASSPATH information for the servlet engine.
<code>ServletData</code>	NTV	Optional. Information supporting NAS features.
<code>InitArgs</code>	NTV	Optional. Supports NAB-generated code.

Specifying NAS Registry and ACL Information

`ServletRegistryInfo`

The `ServletRegistryInfo` section contains information that enables the servlet to be registered into NAS. The only required attribute is `guid`, which specifies a unique identifier used internally by the server. For more information on the NAS registry, see “The NAS Registry” on page 174.

The `ServletRegistryInfo` section is made up of the following attributes:

Name	Type	Value
<code>enable</code>	Str	Optional. Indicates whether NAS instantiates and activates the server for use immediately after registration. Valid values are <code>y</code> or <code>n</code> , the default is <code>y</code> .
<code>encrypt</code>	Str	Optional. Indicates whether internal server communications to this servlet are encrypted. Valid values are <code>y</code> or <code>n</code> .
<code>lb</code>	Str	Optional. Indicates whether sticky load balancing is set for this servlet. Valid values are <code>y</code> or <code>n</code> . There is no default; if this field empty, load balancing is determined by the server. <code>lb</code> is automatically set to <code>y</code> for servlets in applications that use local sessions, since those servlets can not be distributed (see Chapter 11, “Creating and Managing User Sessions”).
<code>descr</code>	Str	Optional. Description of the servlet that appears in the NAS Administration Tool.

Name	Type	Value
<code>guid</code>	Str	Required. Globally unique identifier generated by <code>kguidgen</code> . For more information, see “GUIDs” on page 174.
<code>serverInfo</code>	StrArr	Optional array of server descriptions. Each server description uses the syntax: <code>SRVR_IP_ADDR:SRVR_IP_PORT[=SRVR_FLAGS]</code> <code>SRVR_IP_ADDR</code> indicates the server’s IP address. <code>SRVR_IP_PORT</code> indicates the Executive Server port number. <code>SRVR_FLAGS</code> is optional: <code>0x80000000</code> sets sticky load balancing, <code>0x00000001</code> sets the <code>enable</code> flag (settings for each server override the default settings given in the <code>lb</code> and <code>enable</code> descriptions above).
<code>acl</code>	StrArr	Optional. Array of users or user groups and their permissions with respect to servlet execution. For more information, see below.

The `acl` field defines an array of users and/or groups that can access this servlet. User and group names are defined in the directory server. For information about adding directory server users, see “Storing and Managing Users and Groups” in Chapter 6, “Securing Applications,” in the *Administration Guide*.

For example, the following entry enables the user `fred` with permission to read and the group `managers` permission to read and write.

```
"acl"      StrArr      ["fred=read", "managers=readwrite"]
```

Your servlet tests for authorization and conditionally executes based on user permissions. For more information about component security in general, see Chapter 12, “Writing Secure Applications.”

Specifying Servlet Engine Information

ServletRunnerInfo

The `ServletRunnerInfo` section contains information relating to the NAS servlet engine, and consists of the following attribute:

Name	Type	Value
<code>ServletClassPath</code>	Str	The class name of the servlet (without the <code>.class</code> extension), which resides either in a path in the <code>CLASSPATH</code> environment variable or in a path in <code>GX_CLASSPATH</code> (configured in the registry). This enables the servlet engine to know which class file to load. For example, a servlet <code>Foo</code> in package <code>com.mysite.package</code> is defined as: "ServletClassPath" Str "com.mysite.pkg.Foo"

Note If the NAS class loader is unable to find a class file through the `SYSTEM_JAVA` parameter, the registry parameter that contains `GX_CLASSPATH`, NAS hands the request to the Java class loader, which in turn reads the `CLASSPATH` environment variable to find the class file. This can be potentially confusing behavior, as classes can load even if they are not referenced in `GX_CLASSPATH`, as long as they are in `CLASSPATH`.

Specifying Servlet Metadata, Result Cache, Required Session Variables, and NAS Features

ServletData

The `ServletData` section consists of information about the servlet itself, information about optional result caching, and particular information about two NAS features: form field validation and named action handlers. For more information about these and other NAS features, see Chapter 13, "Taking Advantage of NAS Features."

The `ServletData` section consists of the following attributes:

Name	Type	Value
<code>CreateDate</code>	Str	Optional. Date the servlet was created.
<code>Description</code>	Str	Optional. Description of the servlet. The NAS Administration Tool uses this field as a servlet's description.
<code>CacheTimeout</code>	Int	Optional. Elapsed time (in seconds) before the memory cache for this servlet is released.
<code>CacheSize</code>	Int	Optional. Size (in kilobytes) of the memory cache for this servlet.
<code>CacheCriteria</code>	Str	Optional. A string of comma-delimited descriptors; if a descriptor matches the input, the results of the servlet are cached. See below for syntax.
<code>RequiredSessionVars</code>	StrArr	Optional. Variable list indicating which variables are required in the session object. Validates that the session is created and the configured values are in the session when the user calls <code>FIXME</code> .
<code>Parameters</code>	NTV	Optional. Describes validation input rules for each parameter expected by the servlet. Typically used only in code generated by NAB.
<code>FormActionHandlers</code>	NTV	Optional. Form button names expected by the servlet, specifying the corresponding handler routine name for each button. Typically used only in code generated by NAB.

Use the follow syntax for the `CacheCriteria` field. For more information, see “Caching Servlet Results” in Chapter 13, “Taking Advantage of NAS Features.”

Syntax	Description
<i>arg</i>	Test succeeds for any value of <i>arg</i> in the input parameter list. For example: "EmployeeCode"
<i>arg=v</i>	Test whether <i>arg</i> matches <i>v</i> (a string or numeric expression). For example: "stock=NSCP". Assign an asterisk (*) to the argument to cache a new set of results every time the servlet runs with a different value. For example: "EmployeeCode=*"

Syntax	Description
<code>arg=v1 v2</code>	Test whether <i>arg</i> matches any values in the list (<i>v1</i> , <i>v2</i> , and so on). For example: "dept=sales marketing support"
<code>arg=n1-n2</code>	Test whether <i>arg</i> is a number that falls within the given range. For example: "salary=40000-60000"

The `Parameters` field is a subsection that contains all the parameter names in a form and their corresponding validation input rules. This attribute is normally only used in code generated by Netscape Application Builder. For more information about validation, see “Validating Form Field Data” in Chapter 13, “Taking Advantage of NAS Features.”.

Each parameter is listed with a name and the following attributes:

Name	Type	Value
<code>inputRequired</code>	Str	Optional, indicates whether an input value must be supplied for that parameter. Valid values are <i>y</i> or <i>n</i> , the default is <i>n</i> .
<code>inputRule</code>	Str	Optional, specifies how the named parameter should be validated. <code>inputRule</code> indicates the type of data the value should be validated against. For more information, see “Validation Rules” in Chapter 13, “Taking Advantage of NAS Features.”

The `FormActionHandlers` field is a subsection that contains all form SUBMIT buttons in a form and a corresponding form action handler method for each. This attribute is normally only used in code generated by Netscape Application Builder (NAB).

Each button name is listed with a the name of an action handler defined in the servlet. The type is `Str`. For example:

```
"FormActionHandlers" NTV {
    "Button1"           Str    "Handler1",
    "Button2"           Str    "Handler2",
}
```

When the form is submitted, the servlet determines which button was clicked and routes the request and response objects to the appropriate action handler. For more information about named event handlers, see “Creating Named Form Action Handlers” in Chapter 13, “Taking Advantage of NAS Features.”

InitArgs

The `InitArgs` section consists of initial parameters to the servlet and their corresponding initial values. This attribute is normally only used in code generated by Netscape Application Builder (NAB).

These parameters are available to the servlet by calling the `getInitParameterNames()` method in the `ServletConfig` interface, and the parameter values are available by calling the `getParameter()` method in the `ServletConfig` interface. These are usually used by the servlet during the initialization phase.

Example Servlet Configuration Files

This example shows an NTV file that configures two servlets:

```

NTV-ASCII {
    "HelloWorldServlet" NTV {
        "ServletRegistryInfo" NTV {
            "guid" Str "{E1E5C6BB-A677-161A-8B60-0800209C42F2}",
        },
        "ServletRunnerInfo" NTV {
            "ServletClassPath" Str "HelloWorldServlet",
        },
        "ServletData" NTV {
            "CreateDate" Str "12-03-1998",
            "Description" Str "Sample with cache criteria",
            "CacheTimeOut" Int "300",
            "CacheSize" Int "5",
            "CacheCriteria" Str "cache",
        },
        "InitArgs" NTV {
            "name1" Str "value1",
            "name2" Str "value2",
        },
    },
    "MyServlet" NTV {
        "ServletRegistryInfo" NTV {
            "type" Str "j",
            "enable" Str "y",
            "encrypt" Str "n",
            "lb" Str "y",
            "descr" Str "Description",
            "guid" Str "{F2920FDD-A6A0-161A-F4EB-0800209C42F2}",
        },
        "ServletRunnerInfo" NTV {
            "ServletClassPath" Str "MyServlet",
        },
        "ServletData" NTV {
            "FormActionHandlers" NTV {
                "submit" Str "OnSubmit",
                "clear" Str "OnClear",
            },
        },
    },
}

```

This example shows a single servlet configured with a number of parameters:

```
NTV-ASCII {
  "ReceiptServlet"      NTV    {

    "ServletRegistryInfo" NTV    {
      "type"              Str    "j",
      "enable"            Str    "y",
      "encrypt"           Str    "n",
      "lb"                Str    "y",
      "group"             StrArr ["BookStore"],
      "guid"              Str    "{9D3CFF79-A81B-161A-CAEF-0800209C42F2}",
    },

    "ServletRunnerInfo"  NTV    {
      "ServletClassPath" Str    "ReceiptServlet",
    },

    "ServletData"        NTV    {
      "Description"      Str    "Receipt for books purchased",

      "Parameters"      NTV    {
        "zip"            NTV    {
          "inputRequired" Str    "y",
          "inputRule"     Str    "VALIDATE_US_ZIPCODE",
        },
        "ssn"            NTV    {
          "inputRequired" Str    "y",
          "inputRule"     Str    "VALIDATE_SSN",
        },
        "email"          NTV    {
          "inputRequired" Str    "y",
          "inputRule"     Str    "VALIDATE_EMAIL",
        },
        "phone"          NTV    {
          "inputRequired" Str    "n",
          "inputRule"     Str    "VALIDATE_US_PHONE",
        },
      },
    },
  },
},
}
```

Creating EJB Property Files

EJB property files (also known as bean property files) are simple text files that determine how an Enterprise JavaBeans (EJB) container sets up and maintains the beans that are part of your NAS applications.

The following topics are presented in this chapter:

- Introducing Bean Property Files
- Declaring a Deployment Descriptor
- Declaring a Session Bean Descriptor
- Declaring an Entity Bean Descriptor
- Declaring Control Descriptors
- Creating Access Control Entries
- Manually Editing Property Files

Introducing Bean Property Files

A bean property file is a text file containing EJB property information that might otherwise be encapsulated programmatically in the EJB's `javax.ejb.DeploymentDescriptor` file. The format of the contents in the file corresponds exactly with a standard JavaBeans property file, except that an EJB file may contain entries specific to EJBs, such as access control entries. There are two advantages to providing property information in a simple text file:

- Using a text file instead of a binary serialized class makes it easy to manipulate bean meta data without requiring a specialized development tool.
- Storing text in source control systems is easier with text files.

Although bean property files are easy to edit in conventional text and code editors, you may prefer to use the NAS deployment tool or the Netscape Application Builder (NAB) to create and manipulate property files graphically.

A bean property file contains entries for an EJB. It must include a base bean deployment descriptor that lists the bean type, globally unique identifier (GUID), version number, bean home name, and the fully-qualified classfile names that comprise the bean. A bean property file also contains either a session bean descriptor (for session EJBs), or an entity bean descriptor (for entity EJBs). Optionally, a bean property file may also contain control descriptor entries that specify transaction and security management for the bean.

Bean property files are standard text files and can be edited with any text editor. They can be named anything, but by convention have a filename suffix of `.properties`.

Declaring a Deployment Descriptor

A base bean deployment descriptor provides general information about an EJB, including its type, its GUID, version number, and classfiles that comprise the EJB.

Specifying the EJB Type

All EJBS are either session beans or entity beans. You must specify an EJB's type information using the `BeanType` property. This property is required for all EJBS.

For example, the following code declares that an EJB is a session bean:

```
# BeanType
# Must be one of {SESSION, ENTITY}
BeanType=SESSION
```

Specifying a GUID

A GUID provides a unique identifying number for an EJB, servlet, extension, or old-style NAS AppLogic. The GUID property is mandatory for all EJBS. GUIDs are normally assigned automatically by the deployment tool, or the Netscape Application Builder (NAB). You can also generate GUIDs directly by running the `kguidgen` utility provided with NAS ("GUIDs" on page 174).

The following code illustrates how a GUID is declared in a bean property file.

```
# GUID (type:STRING)
GUID={B6B9FE90-7E86-11D2-94D6-0060083A5082}
```

Specifying a Version Number

The EJB property file format provides a mechanism to support bean versioning, using the `VersionNumber` property. Currently, however, EJB versioning is not supported in NAS 4.0. If you provide this value, always set it to 1:

```
# VersionNumber (type:Integer)
VersionNumber=1
```

Specifying the EJB Location

The Java Naming and Directory Interface (JNDI), registers each EJB using the EJBs unique storage location. The `BeanHomeName` property enables you to specify the relative path name to the EJB. This entry is required for each EJB.

For example, the following code snippet sets the relative path for the `ShoppingCart` EJB:

```
# BeanHomeName
BeanHomeName=Bookstore/cart/ShoppingCart
```

Indicating the Classfiles That Make up an EJB

A deployment descriptor must identify the class names, or classfiles, that compose the bean. There are three class files that are part of every bean:

- `EnterpriseBeanClassName`, the fully qualified class name of the EJB's implementation
- `HomeInterfaceClassName`, the fully qualified class name of EJB's home interface
- `RemoteInterfaceClassName`, the fully qualified class name of bean's remote interface

You must supply all three classfile names for each EJB. Failure to do so prevents you from using the EJB. Use a fully qualified Java class name format for each classfile.

For example, the following code snippet establishes the classfiles for the `ShoppingCart` EJB:

```
EnterpriseBeanClassName = GXApp.OnlineBookstore.cart.ShoppingCartBean
HomeInterfaceClassName = GXApp.OnlineBookstore.cart.IShoppingCartHome
RemoteInterfaceClassName = GXApp.OnlineBookstore.cart.IShoppingCart
```

Declaring a Session Bean Descriptor

Some EJB property file information is specific to session beans. This information can be thought of as a session bean descriptor. The session bean descriptor, however, is actually part of the EJB property file. There are three properties that are specific to session beans:

- `SessionTimeout`, which specifies how long a session bean can remain inactive before it is removed by its container.
- `PassivationTimeout`, which specifies how long a session bean can remain inactive before it is passivated (temporarily stored out of memory) by the system.
- `StateManagementType`, which specifies if a session bean is stateless or stateful.

If the `BeanType` property is set to `SESSION`, then two of these three properties must be set. `SessionTimeout` and `StateManagementType` are mandatory for all session beans. `PassivationTimeout` is optional.

If you set `SessionTimeout` or `PassivationTimeout` to zero, NAS uses the system default setting for these values. The system default setting can be adjusted using the NAS Administration Tool. For more information, see the *Administration Guide*.

Specifying Session Timeout

`SessionTimeout` is a property that permits you to set the total time in seconds that a stateful session bean can exist without being accessed before it is automatically removed by the system. A value of zero indicates that the bean should use the system default. The `SessionTimeout` property is mandatory for all session beans.

For example, the following code snippet tells the container to use the system default setting for determining when to remove an inactive session bean:

```
# SessionTimeout (type:Integer, units: seconds)
SessionTimeout=0
```

Specifying Passivation Timeout

`PassivationTimeout` is a property that permits you to specify the total time in seconds that a stateful session bean can remain active without being accessed before it is removed from memory and stored in a temporary location. A value of zero indicates that the bean should use the system default. If specified, this value should always be less than the time specified for `SessionTimeout`, or it is effectively be ignored.

Note This entry is optional. If it is omitted, the container uses a default passivation timeout to passivate a session bean.

For example, the following code snippet tells the container to use the system default to passivate an inactive session bean:

```
# PassivationTimeout (type:Integer, units: seconds)
PassivationTimeout=0
```

Specifying Session Bean State

The `StateManagementType` property indicates whether a session bean is stateful or stateless. This property is mandatory for all session beans.

For example, the following code indicates that a session bean is a stateful session bean:

```
# StateManagementType
# {STATEFUL_SESSION, STATELESS_SESSION}
StateManagementType=STATEFUL_SESSION
```

Note `StateManagementType` must match the mode that was supplied to `ejbc` when compiling stubs and skeletons for the session bean.

Declaring an Entity Bean Descriptor

Some EJB property file information is specific to entity beans. This information can be thought of as an entity bean descriptor. The entity bean descriptor, however, is actually part of the EJB property file. There are two properties that are specific to entity beans:

- `PrimaryKeyClassName`, which specifies the fully qualified class name of the entity bean's primary key.
- `IsReentrant`, which indicates whether an entity bean's code can be reentered.

These properties must be present if `BeanType=ENTITY`. Otherwise, they must be omitted.

Specifying the Primary Key Class Name

Use the `PrimaryKeyClassName` property to specify the fully qualified class name of the entity bean's primary key. This can be either a user-defined class or a simple class (such as `java.lang.String`), but it must be serializable.

For example, the following code snippet indicates that the primary key corresponds to a Java string type:

```
# PrimaryKeyClassName
PrimaryKeyClassName=java.lang.String
```

Indicating Reentry Support

Entity beans can be optionally reentrant. If you implement an entity bean with the necessary thread synchronization and constructs necessary to support reentrancy, you can set the `IsReentrant` property to `true`. Otherwise, always set this property to `false`.

For example, the following code snippet indicates that an entity bean is not reentrant:

```
# IsReentrant {true | false}
IsReentrant=false
```

Declaring Control Descriptors

You can use additional properties to specify EJB transaction and security using control descriptors. Control descriptors also enable you to specify optional, individual property overrides for bean methods.

As with session bean and entity bean descriptors, control descriptors are actually part of the bean property file. Control descriptors, however, have an indexing syntax that enables you to associate an index number with a particular method, and then use that index number to indicate specific property overrides for transaction and security control. For example, you must always provide at least one `MethodName` control descriptor for all beans, and it takes the form:

```
ControlDescriptor.0.MethodName=DEFAULT
```

The `ControlDescriptor` preface indicates that what follows is a descriptor, rather than a simple bean property. The 0 value is a default index value that indicates a default behavior to apply to all methods that do not have overrides. Each subsequent control descriptor for specific method names must increment the index by 1 without skipping any numbers. `MethodName` is used to indicate a specific method to override. When `MethodName=DEFAULT`, the container knows to apply the settings for all control descriptors with an index of 0 to any bean method that does not have overrides. Otherwise, `MethodName` must always specify the actual name of a bean method.

There are four control descriptor properties you can set for a bean and its methods:

- `MethodName`, which specifies both a default index value for methods that are not overridden, and index values for specifically named EJB methods. The index is used in subsequent control descriptors and access control identifiers to specify which method is affected by the property setting.
- `TransactionAttribute`, which enables you to specify transaction requirements for an EJB or specific methods of an EJB.
- `RunAsMode`, which enables you to specify whether an EJB uses a client's identity to execute methods, or whether it uses a specific user identity to run methods.
- `RunAsIdentity`, which is used to indicate a specific identity to use when running a method whose `RunAsMode` is set to `SPECIFIED_IDENTITY`.

`RunAsMode` and `RunAsIdentity` are used with access control descriptors to provide secure EJB access. For more information about access control descriptors, see “Creating Access Control Entries” on page 202.

Creating Index Values for EJB Methods

You must always specify at least one `MethodName` property for each EJB. This property declaration always takes the form:

```
ControlDescriptor.0.MethodName=DEFAULT
```

This form of the `MethodName` property provides a default index value for all EJB methods for which you do not supply explicit index numbers. In effect, it creates a default control descriptor for all EJB methods that do not have their own, specific index entries.

You can also optionally create explicit index values for any EJB methods using the following `MethodName` syntax:

```
ControlDescriptor.<n>.MethodName=<insertmethodnamehere>
```

`<n>` indicates an index value greater than 0, and `<insertmethodnamehere>` is a place holder for the name of an actual bean method. You must always increment `<n>` by one each time you provide an index value for another method, and you cannot skip numbers.

For example, the following code snippet provides a default method control descriptor, and two additional, explicit method control description indexes:

```
# Method Names
ControlDescriptor.0.MethodName=DEFAULT
ControlDescriptor.1.MethodName=UnloadCart
ControlDescriptor.2.MethodName=TallyCart
```

Providing index entries for method names enables you to set properties and conditions on them, such as transaction attributes, that differ from the setting for the bean as a whole.

Specifying Transaction Requirements

One of the most powerful features of EJBs is that you can stipulate a global level of transaction support for the bean in the bean property file. In all but one case, when the bean container processes the bean property file, it uses these settings to handle and initiate transactions as needed, freeing you from having to code transaction handling in your bean unless you desire to do so.

The `TransactionAttribute` property specifies the level of transaction support for a bean. The following table lists the attributes for transaction management and describes what each attribute means.

Transaction Management Attribute	Purpose
<code>TX_BEAN_MANAGED</code>	The EJB or method explicitly controls transaction boundaries using the <code>javax.transaction.UserTransaction</code> interface. The container does not manage transactions for this bean.
<code>TX_MANDATORY</code>	The EJB container invokes the EJB method using the transaction context associated with the calling client. If the client has not started a transaction when the bean is invoked, the container throws the <code>TransactionRequired</code> exception.
<code>TX_NOT_SUPPORTED</code>	The container invokes the EJB without a transaction context. If a client calls with a transaction scope, the container suspends the association of the transaction with the current thread before delegating the method call to the EJB. When the bean's work is completed, the container resumes the transaction.
<code>TX_REQUIRED</code>	The container invokes the EJB using the client's transactional context, unless the client does not have one. If the client does not have a transaction in progress, the container automatically starts a transaction before delegating the method call to the bean. The container attempts to commit the transaction when the method call on the bean is completed, unless the transaction was marked for rollback or an exception was raised.
<code>TX_REQUIRES_NEW</code>	The EJB requires that the container start a new transaction when a method is invoked. The container attempts to commit the transaction when the bean's method call completes. When a client calls into a bean with this attribute, the container suspends the client's transaction before starting the bean's new transaction. Upon completing the bean's transaction, the container resumes the client's suspended transaction.
<code>TX_SUPPORTS</code>	The EJB mirrors the client's transaction scope. If the client has a transaction context, the EJB is invoked in that context. If not, the EJB is invoked without a transaction context.

Note If you set `TransactionAttribute` to `TX_BEAN_MANAGED`, you must handle transactions in your bean code. The bean container will not handle transactions for a bean with `TransactionAttribute` set to `TX_BEAN_MANAGED`.

The syntax for the `TransactionAttribute` property takes the form:

```
ControlDescriptor.0.TransactionAttribute=<value>
```

where `<value>` is one of the transaction levels detailed in the table above.

For example, the following code snippet sets the `TransactionAttribute` for the EJB to `TX_SUPPORTS`:

```
ControlDescriptor.0.TransactionAttribute=TX_SUPPORTS
```

Specifying Identity Information

By default, when you call an EJB method, it executes under the auspices of the client's identity. The client's identity is something your system administrator uses on your systems to identify users, such as user names. EJB containers read identity information from your system, and translate it into a form usable by EJBs.

You can use the control descriptor's `RunAsMode` property to specify that a bean or a bean method execute under any client's identity (the default), or that a bean or method execute as if it were accessed by a specific user. If you specify that a bean or method executes under the auspices of a specific user other than the actual client, then you must also use the `RunAsIdentity` property to specify the identity to run under.

`RunAsMode` takes the following form:

```
ControlDescriptor.n.RunAsMode={CLIENT_IDENTITY | SPECIFIED_IDENTITY}
```

where `n` is 0 to apply the mode to the entire bean or greater than 0 to apply to a specific bean method.

Note The EJB specification permits a third option for `RunAsMode`, `SYSTEM_IDENTITY`, but support for this option is not available in NAS.

For example, the following code snippet sets the global mode for a bean to use the current client's identity:

```
# RunAsMode
ControlDescriptor.0.RunAsMode=CLIENT_IDENTITY
```


The following code snippet sets the global mode for the bean to a specific user, and then uses `RunAsIdentity` to list that user as `SystemAdministrator`:

```
ControlDescriptor.0.RunAsMode=SPECIFIED_IDENTITY
ControlDescriptor.0.RunAsIdentity=SystemAdministrator
```

To force a method to run using a specified identity, replace the global index value (0) with the method's index value. For example, the following code sets the global mode for a bean to `CLIENT_IDENTITY`, then sets the user ID for a specific method associated with index value 1 to `SystemAdministrator`:

```
# RunAsMode
ControlDescriptor.0.RunAsMode=CLIENT_IDENTITY
ControlDescriptor.1.RunAsMode=SPECIFIED_IDENTITY
ControlDescriptor.1.RunAsIdentity=SystemAdministrator
```

Note Besides forcing a bean or method to run under the auspices of a certain user, you can also restrict the ability to run a bean or its methods to particular users with access control entries. For information about using access control entries, see “Creating Access Control Entries” on page 202

Note Client and group names are defined in the directory server. For information about adding directory server users, see “Storing and Managing Users and Groups” in Chapter 6, “Securing Applications,” in the *Administration Guide*. For more information about component security in general, see Chapter 12, “Writing Secure Applications.”

Specifying Parameter Passing Rules

When a servlet or EJB calls another bean that is co-located within the same process, NAS does not perform marshalling of all call parameters by default. This optimization allows the collocated case to execute far more efficiently than if strict “by-value” semantics were used. In certain cases, however, you may want to ensure that parameters passed to a bean are always passed by value. NAS supports the ability to mark a bean or even a particular method within a bean as optionally requiring pass by value semantics. Because this option can decrease performance by greatly increasing call overhead, the default value is “false” if this entry is unspecified.

The following code snippet forces a bean to pass all parameters by value:

```
ControlDescriptor.0.AlwaysPassByValue=true
```

Creating Access Control Entries

Access control entries in a bean property file declare security properties of a bean. The access control entry for a bean specifies the name or names of clients or groups that can use the bean. The bean's container uses the information in the bean property file to determine who has access to the bean at run time.

Note Client and group names are defined in the directory server. For information about adding directory server users, see “Storing and Managing Users and Groups” in Chapter 6, “Securing Applications,” in the *Administration Guide*.

Specifying Access Control

Access control entries for a bean must include a `MethodName` entry that is set to `DEFAULT`, and one `Identity` entry for each client or group that can access the bean. For example, the following access control entries tell a bean container that only members of the groups “manager” and “is-staff” can access a bean.

```
# default ACL for beans says bean can only be invoked by
# members of the group "managers" or "is-staff"
```

```
AccessControlEntry.0.MethodName=DEFAULT
AccessControlEntry.0.Identity.0=managers
AccessControlEntry.0.Identity.1=is-staff
```

The `MethodName=DEFAULT` entry provides default access control for any method that does not have its own access control entry elsewhere in the property file. For more information about providing access control entry overrides for individual bean methods, see “Overriding Access Control on Bean Methods” on page 203.

You can provide as many or as few access control entries for a bean as desired. As this example illustrates, `Identity` entries are indexed. The first entry is always 0, and subsequent entries are incremented in value by 1.

Note Specified `Identity` values must already be known to the system. A common error message seen when attempting to register a bean with an access control entry containing an unknown identity is:

```
unable to complete bean meta-data registration
unable to set value for "AllowedIdentity"
```

Client and group names are defined in the directory server. For information about adding directory server users, see “Storing and Managing Users and Groups” in Chapter 6, “Securing Applications,” in the *Administration Guide*. For more information about component security in general, see Chapter 12, “Writing Secure Applications.”

Overriding Access Control on Bean Methods

Besides providing access control for an entire bean, you can specify different access controls for individual bean methods. Frequently, a bean provides a broad set of access control entries, but some of its methods may require a more restrictive set of rights. If you specify access control for individual bean methods, the method entries supersede the access control rights you specified for the entire bean.

As with bean-level access control entries, method-level access control entries consist of two parts:

- `MethodName` specifies the name of the method for which you are providing the access control entries.
- `Identity` specifies the name of the client or group to whom you grant execute privilege.

`MethodName` is indexed. The index for the first `MethodName` entry in a bean property file is 0. It defines the default access for any method that does not have its own access control entry.

All subsequent method-level access control entries increment the index value by 1. For example, the following snippet provides access control for a specific method, `IncrementSalary`, and limits access to those members of the `PayRoll` group:

```
AccessControlEntry.1.MethodName=IncrementSalary
AccessControlEntry.1.Identity.0=Payroll
```

You can provide as many or as few access control entries for bean methods as desired. `Identity` entries for each method, too, are indexed. The first entry is always “0”, and subsequent entries are incremented in value by 1.

Running as Another User

When you access an EJB, you run the bean as a client with a known identity. You can force a bean or one of its methods to run with a specific identity by adding a `RunAsMode` statement to the bean property file.

The EJB specification lists three identities that can be used to execute a bean:

- `CLIENT_IDENTITY`, which means that a bean or its methods are executed using the client's identity. This is the default identity for beans without identity entries in the property file.
- `SPECIFIED_IDENTITY`, which means that a bean or its method(s) execute using a particular identity that is supplied as an argument to `RunAsMode`. A `SPECIFIED_IDENTITY` overrides the client's actual identity.
- `SYSTEM_IDENTITY`, which means that a bean or its method(s) execute using a system resource identity. Currently NAS does not support using the `SYSTEM_IDENTITY` mode.

To specify that a bean run in `CLIENT_IDENTITY` mode, add the following entry to the bean property file:

```
# RunAsMode
ControlDescriptor.0.RunAsMode=CLIENT_IDENTITY
```

To specify that a bean run in `SPECIFIED_IDENTITY` mode, you must add two lines to the bean property file: one that indicates the mode to run in, and a second that indicates the specific identity to use. In the following example, a bean is set up to run using the `Payroll` identity.

```
# RunAsMode
ControlDescriptor.0.RunAsMode=SPECIFIED_IDENTITY

# RunAsIdentity
ControlDescriptor.0.RunAsIdentity=Payroll
```

Example Bean Property File

This example bean property file comes from the Online Bookstore sample application:

```
# Shopping CartBean properties

# BeanType
#
# Specify one of session or entity as well as the relevant
# properties below.  Properties in this file for the "other" type
# of bean will be ignored.
#
# { SESSION, ENTITY }
BeanType=ENTITY

# GUID (type:STRING)
#
GUID={A000E626-0A12-1656-EBD7-0800208055C0}

#####
# DeploymentDescriptors
#

# VersionNumber (type:DWORD)
VersionNumber=1

# BeanHomeName
#
# This is the name used to register the bean in JNDI.  The name is
# always relative; the JNDI root prefix is specified elsewhere.
#
BeanHomeName=nsOnlineBookstore/BookAccount

# EnterpriseBeanClassName
# HomeInterfaceClassName
# RemoteInterfaceClassName
#
# In order to bootstrap from this file, the user must specify these 3
# mandatory classfiles.  Use the java class name format, i.e
# "hello.HelloImpl"
#
EnterpriseBeanClassName=GXApp.nsOnlineBookstore.account.BookAccountBean
HomeInterfaceClassName=GXApp.nsOnlineBookstore.account.IBookAccountHome
RemoteInterfaceClassName=GXApp.nsOnlineBookstore.account.IBookAccount
PrimaryKeyClassName=java.lang.String

# for entity beans, mandatory property
IsReentrant=true

#####
# ControlDescriptor
#

# The MethodName must always be specified, and should be the
```

```

# special string "DEFAULT" to indicate that the control
# descriptor changes apply to all methods.
#
# Multiple control descriptors can be specified by
# increasing the index, i.e. ControlDescriptor.1.MethodName=foobar
# Indices cannot be skipped.
#
ControlDescriptor.0.MethodName=DEFAULT

# RunAsMode
#
# { CLIENT_IDENTITY, SPECIFIED_IDENTITY, SYSTEM_IDENTITY }
#
ControlDescriptor.0.RunAsMode=CLIENT_IDENTITY

# TransactionAttribute
#
# { TX_BEAN_MANAGED, TX_MANDATORY, TX_NOT_SUPPORTED, TX_REQUIRED,
#   TX_REQUIRES_NEW, TX_SUPPORTS }
#
ControlDescriptor.0.TransactionAttribute=TX_REQUIRED

```

Manually Editing Property Files

NAS provides an administrative tool and a visual application tool (NAB) that permit you to build bean property files in a visual manner. Because bean property files are simple text files, however, you can choose to create or modify them by hand, and manually register them to NAS using the provided utility `beanreg` (for more information, see “Manually Deploying EJBs” in Chapter 10, “Creating Configuration Files” in the *Administration Guide*).

If you modify a bean's property file by hand in a text editor, here are a few precautionary items to note.

Although the `beanreg` utility catches most syntactical errors in a property file, it cannot detect all possibly legal semantic variations that you can introduce when editing a file by hand. For example, if you modify a bean's deployment descriptor using the deployment tool, the deployment tool does not permit you to create a bean with a default transaction for a bean of `TX_BEAN_MANAGED` and a per-method override that is anything but `TX_BEAN_MANAGED`. You can, however, create a bean property file by hand that violates this rule, and `beanreg` does not catch it.

To facilitate support for Java and EJB method overloading, NAS internally mangles method names to make them unique. For instance, the following methods, have the same method name, but are uniquely identified internally by a mangled identifier:

```
public void snap()
public void snap(int snaps)
```

Although this mangling is normally hidden from the you, it becomes visible when you edit bean property files by hand and want to supply per-method overrides for either access control entries or control descriptors. For example, to manually set the second `snap()` method to transaction-mode `TX_REQUIRED`, the property file should contain an entry resembling:

```
ControlDescriptor.1.MethodName=snap__void__int
ControlDescriptor.1.TransactionAttribute=TX_REQUIRED
```

You can discover the mangled names for your bean's methods by using the `-p` option to the `beanreg` utility. Given a standard bean property file, run:

```
beanreg -p <your bean.properties file>
```

`beanreg` prints out a list of method names and their corresponding mangled identifiers. Use these identifiers when editing a property file.

Be sure to rerun `beanreg` if you change your the remote or home interfaces for an EJB. Changing these interfaces may invalidate method-level overrides, and if they become invalid, `beanreg` refuses to register the bean. If this occurs, comment out the method-level overrides, rerun `beanreg -p` to identify the new method identifiers, and modify the property file accordingly.

Creating Datasource Property Files

To create a JDBC datasource, you must provide a properties file for the datasource and register it with NAS. This file contains several entries that define the way NAS connects to the datasource.

The file can be named anything, but generally has a suffix of `.props`. Fieldnames are not case-sensitive.

Datasource Property File Entries

A datasource property file contains the following entries:

Property	Example	Description
Database	Database=ksample	Name of the database to connect to.
Datasource	Datasource=ksample	Name to assign to this datasource.
Username	Username=kdemo	Valid user name for this database
Password	Password=kdemo	Valid password for the user name listed.
DriverType	DriverType=ORACLE_OCI	Backend-specific JDBC driver. One of the following: ORACLE_OCI (Oracle) DB2_CLI (DB2) INFORMIX_CLI (Informix) SYBASE_CTLLIB (Sybase) ODBC (ODBC)
ResourceMgr	ResourceMgr=oraksample	Optional. If this attribute is set, the datasource is available for distributed transactions by way of the resource manager listed. If this attribute is not specified, the datasource is only valid for a local database. The value must be a name that you create for a resource manager under the RESOURCEMGR key. .

Manually Registering Datasources to NAS

You generally specify the datasource properties file when you deploy your application, and your datasources are automatically registered.

If you need to register a datasource manually, use the provided utility dsreg. dsreg is a command-line utility that requires the datasource name and properties file name, as in the following example:

```
dsreg "dataSource" dsPropsFile.props
```

For more information, see “Manually Deploying Data Sources” in Chapter 10, “Creating Configuration Files” in the *Administration Guide*.

Creating and Managing User Sessions

This chapter describes how to create and manage a session that allows user and transaction information to persist between interactions.

This chapter contains the following sections:

- Introducing Sessions
- How to Use Sessions

Introducing Sessions

The term *user session* refers to a series of user-application interactions that are tracked by the server. Sessions are used for maintaining user-specific state, including persistent objects (like handles to EJBs or database result sets) and authenticated user identities, among many interactions. For example, a session could be used to track a validated user login followed by a series of directed activities for that particular user.

The session itself resides in the server. For each request, the client transmits the session ID in a cookie or, if the browser does not allow cookies, the server automatically writes the session ID into the URL.

NAS supports the servlet standard session interface `HttpSession` for all session activities. This interface enables you to write portable, secure servlets.

Additionally, NAS provides an additional interface `HttpSession2`, which gives support for a servlet security framework as well as sharing sessions between servlets and older NAS components (AppLogics).

Behind the scenes, there are two session styles, distributable sessions and local sessions. The main difference between them is that distributable sessions, as the name implies, can be distributed among multiple servers in a cluster, while local sessions are sticky (i.e., bound to an individual server). Sticky load balancing is automatically set for servlets of an application that is configured to use the local session model. You determine which session style to use in the application configuration file.

Sessions and Cookies

A cookie is a small collection of information that can be transmitted to a calling browser, and then retrieved on each subsequent call from that browser so that the server can recognize calls from the same client. The cookie is returned with each subsequent call to the site that created it unless it expires.

Sessions are maintained automatically by a session cookie that is sent to the client when the session is first created. The session cookie contains the session ID, which identifies the client to the browser on each successive interaction. If a client does not support or allow cookies, the server rewrites URLs such that the session ID appears in URLs from that client.

Sessions and Security

The NAS security model is based on an authenticated user session. Once a session has been created, the application user can be authenticated and “logged in” to the session. Each step of an interaction, from the servlet that receives the request to the EJB that generates content to the JSP that formats the output, can be aware that the user has been properly authenticated.

Additionally, you can specify that a session cookie only gets passed on a secured connection (like HTTPS), so the session can only remain active on a secure channel.

For more information about security, see Chapter 12, “Writing Secure Applications.”

How to Use Sessions

To use a session, you first create a session using the `HttpServletRequest` method `getSession()`. Once the session is established, you can examine and set its properties using the provided methods. You can set a session to time out after being inactive for a certain time, or you can invalidate it manually.

You can also bind objects to the session, thus storing them for use by other components.

Creating or Accessing a Session

To create a new session, or to gain access to an existing session, use the `HttpServletRequest` method `getSession()`, as in the following example:

```
HttpSession mySession = request.getSession();
```

`getSession()` returns the valid session object associated with this request, identified in the session cookie which is encapsulated in the request object. If you call this method with no arguments, a session is created if there is not already a session associated with the request. If you call this method with a Boolean argument, then the session is created only if the argument is `true`.

This example shows the `doPost()` method from a servlet that only performs the servlet's main functions if the session is present (note that the `false` parameter to `getSession()` prevents the servlet from creating a new session if one does not already exist):

```
public void doPost (HttpServletRequest req,
                   HttpServletResponse res)
    throws ServletException, IOException
{
    if ( HttpSession session = req.getSession(false) )
    {
        // session retrieved, continue with servlet operations
    }
    else
        // no session, return an error page
    }
}
```

For more information about `getSession()`, see the servlet specification.

Examining Session Properties

Once a session ID has been established, you can use methods in the `HttpSession` interface to examine properties in the session, and methods in the `HttpServletRequest` interface to examine properties in the request that relate to the session.

Use the following methods to examine properties in the session:

HttpSession method	Description
<code>getCreationTime()</code>	Returns the time at which this session was created in milliseconds since January 1, 1970, 00:00:00 GMT.
<code>getId()</code>	Returns the identifier assigned to this session. An HTTP session's identifier is a unique string that is created and maintained by the server.
<code>getLastAccessedTime()</code>	Returns the last time the client sent a request carrying the identifier assigned to the session, or -1 if the session is new. Time is expressed as milliseconds since January 1, 1970, 00:00:00 GMT.
<code>isNew()</code>	Returns a Boolean value indicating whether this session is considered to be new. A session is considered to be new if it has been created by the server and not received from the client as part of this request. This means that the client has not "acknowledged" or "joined" the session and may not ever return the appropriate session identification information when it makes its next request.

For example:

```
String mySessionID = mySession.getId();
if ( mySession.isNew() ) {
    log.println(currentDate);
    log.println("client has not yet joined session " + mySessionID);
}
```

Use the following methods to inspect properties in the request object that relate to the session:

HttpServletRequest method	Description
<code>getRemoteUser()</code>	Gets the name of the user making this request. This information may be provided by HTTP authentication. Returns null if there is no user name information in the request.
<code>getRequestIdSessionId()</code>	Returns the session ID specified with this request. This may differ from the session ID in the current session if the session ID given by the client was invalid and a new session was created. Returns null if the request does not have a session associated with it.
<code>isRequestedSessionIdValid()</code>	Checks whether this request is associated with a session that is currently valid. If the session used by the request is not valid, it is not returned via the <code>getSession()</code> method.
<code>isRequestedSessionIdFromCookie()</code>	Returns true if the session ID for this request was provided from the client as a cookie, or false otherwise.
<code>isRequestedSessionIdFromURL()</code>	Returns true if the session ID for this request was provided from the client as part of a URL, or false otherwise.

For example:

```
if ( request.isRequestedSessionIdValid() ) {
    if ( request.isRequestedSessionIdFromCookie() ) {
        // this session is maintained in a session cookie
    }
    // any other tasks that require a valid session
} else {
    // log an application error
}
```

Binding Data to a Session

You can bind objects to sessions in order to make them persistent across multiple user interactions. The following `HttpSession` methods provide support for binding objects to the session object:

HttpSession method	Description
<code>getValue()</code>	Returns the object bound to a given name in the session, or null if there is no such binding.
<code>getValueNames()</code>	Returns an array of the names of all the values bound into this session.
<code>putValue()</code>	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 <code>Serializable</code> interface. Note that <code>NAS RowSets</code> and <code>JDBC ResultSets</code> are not serializable, and can not be distributed.
<code>removeValue()</code>	Unbinds an object in the session with the given name. If there is no object bound to the given name, this method does nothing.

Binding Notification with `HttpSessionBindingListener`

Some objects may require that you know when they are placed into, or removed from, a session. You can obtain this information by implementing the `HttpSessionBindingListener` interface in those objects. When your application stores data in or removes data from the session, the servlet engine checks whether the object being bound or unbound implements `HttpSessionBindingListener`. If it does, methods in the interface automatically notify the object that it has been bound or unbound.

Invalidating a Session

You can specify that the session invalidates itself automatically after being inactive for a certain amount of time. Alternatively, you can invalidate a session manually with the `HttpSession` method `invalidate()`.

Invalidating a Session Manually

To invalidate a session manually, simply call the following method:

```
session.invalidate();
```

All objects bound to the session are also removed.

Setting a Session Timeout

To set a timeout for distributable sessions, set the following line in the `SessionInfo` section of the application configuration file `appInfo.ntv`:

```
"timeout"           Int      "timeout",
```

To set a timeout value for a local session, invoke the following method in a servlet:

```
session.setMaxInactiveInterval(int timeout).
```

In both types of sessions, set *timeout* to the maximum amount of time (in seconds) that the session should be inactive before NAS can invalidate it. The default is 1800 seconds (30 minutes). You can examine the current timeout value with the method `session.getMaxInactiveInterval()`.

Controlling the Type of Session

You can control the type of session by editing the fields in the `SessionInfo` section of the application configuration file `appInfo.ntv`. For more information, see “Application Configuration Information” in Chapter 10, “Creating Configuration Files.”

Note If you change the type of session from distributable to local or vice versa, you must re-register your application and restart your web server for the change to take effect. This is because some load-balancing characteristics (stickiness in particular) are kept in the web server plug-in. The plug-in must be reloaded after any changes, thus necessitating a web server restart.

Sharing Sessions with AppLogics

Servlet programmers can use the NAS feature interface `HttpSession2` to share distributable sessions between AppLogics and servlets. Sharing sessions is useful when you want to migrate an application from NAS 2.x to NAS 4.0. `HttpSession2` interface adds security and direct manipulation of distributable sessions.

Additionally, if you establish a session in an AppLogic using `loginSession()` and you want to be able to access that session from a servlet, you must call the method `setSessionVisibility()` in the AppLogic class in order to instruct the session cookie to be transmitted to servlets as well as AppLogics. It is important to do this before calling `saveSession()`.

For example, in an AppLogic:

```
domain=".mydomain.com";
path="/"; //make entire domain visible
isSecure=true;
if ( setSessionVisibility(domain, path, isSecure) == GXE.SUCCESS )
    { // session is now visible to entire domain }
```

For more information about `setSessionVisibility()`, see the entry for the AppLogic class in the *Foundation Class Reference*. For more information about sharing sessions between AppLogics and servlets, see the *Migration Guide*.

Writing Secure Applications

This chapter describes how to write secure application components that perform authentication to establish and maintain a user's identity.

The security model described in this chapter was developed for EJBs, and is part of the standard for EJBs. As of the 2.1 servlet specification, there is no standard security model for servlets. NAS has created a model for servlets in order to provide security throughout the application, with some collaboration with the developers of Java standards, so that the model presented here for servlets is based heavily on an emerging standard.

This chapter includes the following sections:

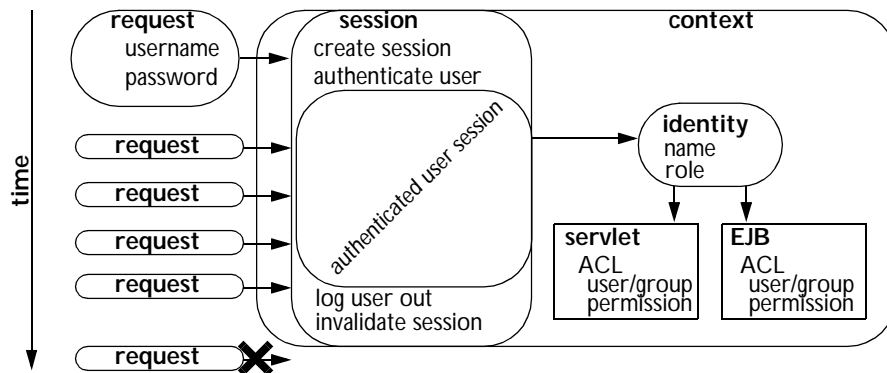
- Understanding the Security Model
- Authenticating a User
- Access Control Lists

Understanding the Security Model

Secure applications depends on two kinds of validation:

- You authenticate users by comparing a user name and password provided by the user with a user name and password stored in the Directory Server using LDAP. Authentication is held in the user's session and remains available until the session expires or the user logs out. Components can retrieve the identity in order to ensure that the caller is authentic.
- You create secure components by setting up access control lists (ACLs) that define permissions granted to specific users and groups. These lists are also stored in the Directory Server using LDAP. Components can test for a user's membership in these groups.

The following diagram shows the basic steps in the security model:



- First, the application establishes a session for the user, and prompts the user to supply a user name and password.
- A servlet authenticates the user name and password by comparing them with the values in the Directory Server, and then “logs the user in” by authenticating the user's session.
- The server's *context*, a programmatic view of the state of the server, recognizes the authenticated session by managing an identity object to components. Components can test authenticity by examining this identity.

- Additionally, components can test the identity to see whether it has permission to perform certain tasks. These permissions are defined in an access control list, which resides in the Directory Server.
- If the user logs out or the session expires, the context can no longer supply the identity, so authentication fails for any requests that require security.

The identity can be used to determine whether a user or group is a member of a certain “role”, so that application flow can be controlled based on a user’s function in the application paradigm. For example, if part of your application is restricted to paying customers only, those customers can be associated with a group called `PayingCustomers`. You can then write your components to perform tasks based on a user’s membership in that group.

Additionally, each component can have an access control list (ACL) that defines the permissions given to various users or groups with respect to that component. You define the permissions and the users/groups to which they apply either from the NAS Administration Tool or in component configuration files. For example, an EJB that represents an employee database can contain an ACL specifying that members of the group `Employee` can only read the data, while members of the group `Manager` can also update the data.

Security Concepts

The concepts that describe how security works are similar for EJBs and servlets, though the implementation is slightly different. This is because the servlet 2.1 specification does not describe a security model. NAS adapts the EJB security model for servlets by providing the `ICallerContext` interface, which allows a servlet to retrieve a caller’s identity as well as determine whether the identity is allowed to perform certain tasks based on its membership in a group (a *role*).

Additionally, both EJBs and servlets benefit from the `IServerContext` interface, which enables the creation of an identity object (an unspecified task for both EJBs and servlets) as well as the retrieval of that identity from the component’s context object (`EJBContext` or `ServletContext`). Note that components have read-only access to the identity, and cannot add, change, or delete identity data.

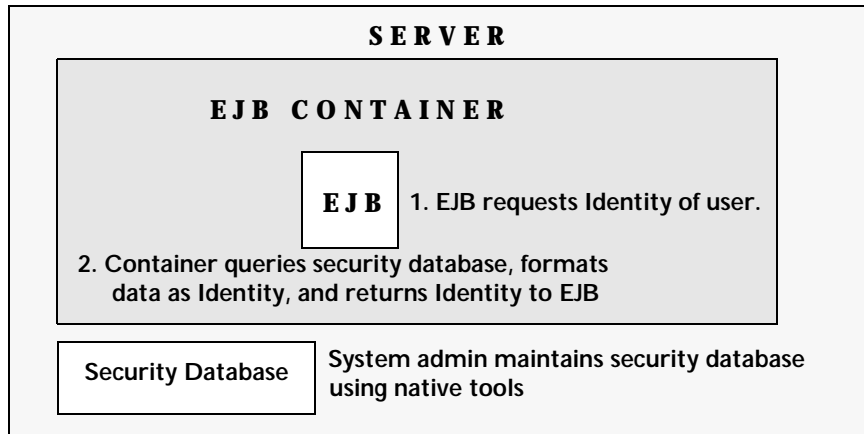
Security for servlets is a representation of the security model for EJBs, described below.

The Security Model for EJBs

The design behind EJBs pushes most of the responsibility for low-level systems management, such as security, from the bean itself to its container and the server. In the EJB security model, a server contains platform-specific user account information that is administered by the system administrator. (In NAS, this information is stored in the directory server.)

The EJB container, which is invisible to you as a developer, is responsible for the following:

- authenticating the user using LDAP
- maintaining the association between an authenticated user and subsequent client calls using sessions and the propagated server context
- providing declarative security for components (ACLs)
- providing an API for programmatic security



As this figure illustrates, the EJB security model separates platform-specific administrative tasks, such as security database maintenance, from EJB run-time activities, such as identity verification. An EJB has read-only access to identity information. Only a system administrator can add, change, or delete records in the security database.

The management and construction of this database is typically vendor/solution specific and outside the scope of the EJB specification.

Note In NAS, the security database resides in the Directory Server (LDAP).

Also, the EJB security model provides a generic, high-level, and standard interface that permits the EJB writer to share EJBs across platforms without having to know about security models on individual platforms. An EJB that uses the standard security interfaces can be deployed on any platform and in any application without recoding.

The Security Model for Servlets

The servlet specification does not supply a security model, though it will in future versions. For this reason, NAS supplies a declarative and programmatic security model based on the model for EJBs described above, except that servlets do not have a container to manage security issues for them.

Servlets are responsible for the following:

- authenticating the user using LDAP (with `loginSession()`)
- maintaining the association between an authenticated user and subsequent client calls using sessions and the propagated server context
- providing declarative security for components (ACLs)
- providing an API for programmatic security

These responsibilities are described in “Authenticating a User” on page 228.

Programmatic Security

You perform the following security tasks programmatically:

- Authorize users by authenticating them and logging them into their respective sessions. To do this, you first prompt for a user name and password, and then supply these to the method `loginSession()`. `loginSession()` verifies the user with the appropriate entry in the Directory Server and then establishes an identity in the user’s session.
- Retrieve the user’s identity from the server context via `getCallerIdentity()`.

- Verify the user's membership in a role using `isCallerInRole()`. Roles are conceptually slightly different from user groups; see “Groups and Roles” on page 225.
- Verify whether a user or group is authorized to perform tasks by using `isAuthorized()` to check an access control list for permissions. For more information, see “Creating and Using Programmatic ACLs” on page 234.

For more information on how to use programmatic security, see “Authenticating a User” on page 228.

Declarative Security

Components can declare a level of security for themselves by establishing access control lists (ACLs). ACLs are named lists in the Directory Server that relate a user or group with one or more permissions. In the case of NAS servlets and EJBs, ACLs determine which users or groups have permission to execute the resource. EJBs can provide bean-level and method-level security this way.

You set up ACLs for specific components in their respective configuration files. If an ACL is present for a given servlet or EJB method, the list is checked to see whether the current user has permission to execute the resource.

You can also set up arbitrary ACLs using the Administration Tool and then inspect permissions programmatically. For more information, see “Creating and Using Declarative ACLs” on page 233.

A client may also be defined in terms of a security role. For example, a company might use its employee database to generate both a company-wide phone book application, and to generate payroll information. Obviously, while all employees might have access to phone numbers and email addresses, only some employees would have access to salary information. Employees with the right to view or change salaries might be defined as having a special security role.

Groups and Roles

In addition to identities, a client may also be defined in terms of a security role. For example, a company might use its employee database to generate both a company-wide phone book application, and to generate payroll information. Obviously, while all employees might have access to phone numbers and email addresses, only some employees would have access to salary information. Employees with the right to view or change salaries might be defined as having a special security role.

A role is different from a user group in that a role defines a function in an application, while a group is a set of users that are related in some way. For example, members of the groups `astronauts`, `scientists`, and (occasionally) `politicians` all fit into the role of `SpaceShuttlePassenger`.

The EJB security model describes roles (as distinguished from user groups) as being described by an application developer and independent of any particular domain. Groups, by contrast, are specific to a deployment domain. The role of the deployer is to map roles into one or more groups.

In NAS, roles correspond to user groups configured in the Directory Server. LDAP groups can contain both users and other groups. Future versions of NAS will provide specific support for roles.

Security and Cookies

Since security is based on sessions, and sessions themselves are partially maintained in a session cookie, some aspects of application security are controlled by settings in the session cookie.

In particular, a cookie has a security attribute controlled by the `setSecure()` method in the `Cookie` class. If you set this attribute with `mySessionCookie.setSecure(true)`, the session cookie is only transmitted if the connection is secure (for example, with HTTPS). On an unsecure connection, the cookie never appears, and so the session can not be validated.

You can require all session activities to adhere to a secure connection by setting the `secure` attribute in the `SessionInfo` structure in the application configuration file `appInfo.ntv`, as in the following example:

```
"SessionInfo" NTV {  
    "secure" Bool "true",  
    "timeout" Int "100",  
    ...  
}
```

For more information about configuration files, see Chapter 10, “Creating Configuration Files.”

Guide to Security Information

Each of the following types of information is shown with a short description, the location where the information resides, how to create the information, how to access the information, and where to look for further information.

- User Information
- Security Roles
- Component ACLs
- General ACLs

User Information

User name, password, etc.

Location Directory Server

How to Create Create using Administration Tool, or programmatically using the LDAP SDK.

How To Access Provide user name and password to `loginSession()` to authenticate session.

See Also “Authenticating a User” on page 228, “Storing and Managing Users and Groups” in Chapter 6, “Securing Applications,” in the *Administration Guide*

Security Roles

Role that defines a function in an application, made up of a number of users and/or groups. LDAP groups function as roles in NAS.

Location Directory Server

How to Create Since roles correspond to user groups in this version of NAS, you create them the same way you create users and groups using Administration Tool, or programmatically using the LDAP SDK.

How To Access Test for a user's membership in a role using `isCallerInRole()`.

See Also “Checking a Client's Security Role” on page 230

Component ACLs

Optional ACL that governs user permissions for a specific component.

Location NAS registry, via the component's configuration file

How to Create Edit the appropriate field in the component's configuration file:

- **servlets:** `acl` field in `ServletRegistryInfo`
- **EJBs:** `AccessControlEntry` entries

How To Access NAS uses these lists internally to determine whether the current user can execute the resource. This type of security is declarative, meaning that there is no programmatic impact. You do not have to write any code in order to use ACLs in this way.

See Also “Creating and Using Declarative ACLs” on page 233, “Specifying Access Control” and “Specifying NAS Registry and ACL Information” in Chapter 10, “Creating Configuration Files.”

General ACLs

Arbitrary named ACLs.

Location NAS registry

How to Create NAS Administration Tool

How To Access Use `isAuthorized()` from the session to test whether the current user has the listed permission, as in the following example:

```
if (sess2.isAuthorized("myAcl",  
                       "read"))  
{...}
```

See Also “Creating and Using Programmatic ACLs” on page 234, “Setting Access Control List Authorization” in Chapter 6, “Securing Applications,” in the *Administration Guide*

Authenticating a User

In order to authenticate a user, create a session for the user, and then validate the user’s user name and password and “log into” the session. This creates an identity, an instance of `java.security.Identity`, that is managed by the server context (a view of the server’s state). Components can now access the identity for the user from the context by calling `getCallerIdentity()`, and can use this identity to test whether a user is authentic or to create an audit trail.

Note that the user’s identifier and password must be validated against a user name and password that are already known to NAS. NAS looks for user and group entries, as well as the access control lists that determine their respective permissions, in the directory server.

Note You can use the NAS Administration Tool to create new users, or you can create entries programmatically within an application using the LDAP JDK included with each installation of NAS. For more information about setting up users in the directory server, see “Storing and Managing Users and Groups” in Chapter 6, “Securing Applications,” in the *Administration Guide*.

Logging In to the Session

You can authenticate a user name and password and log into the session using the method `loginSession()` from the NAS feature `HttpSession2` interface. For example, the following code (in a servlet) authenticates the user and logs them into a session:

```
String name = req.getParameter("username");
String pass = req.getParameter("password");

HttpSession session = req.getSession(true); // create user session
HttpSession2 sess2 = (HttpSession2) session;
if (sess2.loginSession(name,pass))           // authenticate user
{
    // now we have an authenticated session for this user
}
else
{
    // authentication failed
}
```

Once the user is logged into the session, the session can be trusted. Any servlets or EJBs that depend on the session being active can also trust that the user is authentic. If the session expires or the user logs out, the identity is no longer available, and attempts to retrieve the identity fail.

At run time, the container (EJBs) or servlet engine (servlets) throws an exception if an unauthorized user attempts to access the component.

Checking Identity

The identity that is created in the server when the client logs in is an instance of `java.security.Identity`. This object contains the client's name and other information pertinent to identification. You can determine which client has invoked a component through the caller's context using the method `getCallerIdentity()`. See "Retrieving a Client's Identity," below.

Note `getCallerIdentity()` returns `null` if the session does not exist (i.e. has expired) or if the session is not authenticated.

Identity objects can also identify roles (groups of users). For example, you create an identity for a role in order to test whether the current user is in that role using `isCallerInRole()`. See "Checking a Client's Security Role".

Retrieving a Client's Identity

`getCallerIdentity()` operates on the standard `EJBContext` object for EJBs. This example shows an EJB retrieving a caller's identity:

```
/* Get the security identity of a client. */
Identity caller = EJBContext.getCallerIdentity();

/* Put client's name into human readable form. */
String clientName = caller.getName();
```

Since the servlet specification does not provide for this type of context, NAS provides two interfaces, `ICallerContext` and `IServerContext`, that enable you to derive the caller's identity from the standard `ServletContext` object. This example shows how to retrieve a caller's identity in a servlet:

```
ServletContext ctx = getServletContext();
nCtx = (com.netscape.server.IServerContext) ctx;
com.netscape.server.ICallerContext callerCtx = nCtx.getCallerContext();

/* Get the security identity of a client. */
Identity caller = callerCtx.getCallerIdentity();

/* Put client's name into human readable form. */
String clientName = caller.getName();
```

One possible use for `getName()` is to create an audit trail by logging the user's name and the current time. For more information, see "Controlling Execution Flow" on page 231. For other methods available in `java.security.Identity`, see the documentation at:

<http://java.sun.com/products/jdk/1.1/docs/api/java.security.Identity.html>

Checking a Client's Security Role

Once you create the identity, you can then pass it to a subsequent test for equality or use it as the role parameter of the `isCallerInRole()` method. The `isCallerInRole()` method is available in both `javax.ejb.EJBContext` (for EJBs) and `com.netscape.server.ICallerContext` (for servlets).

`isCallerInRole()` requires a `java.security.Identity` object as a parameter, so to use this method you need an identity that corresponds to the role with which you are concerned. You can create an identity for this role using `createIdentityByString("role")`, in the `IServerContext` interface. *role* must correspond to a user group in the Directory Server.

`isCallerInRole()` returns a boolean value indicating true or false. For example, in the following code fragment, an EJB checks to see if the client's role is `PayrollDept`:

```
/* Check if the client is in the PayrollDept role */
Identity PayRoll = EJBContext.createIdentityByString("PayrollDept");
boolean B = EJBContext.isCallerInRole(PayRoll);
```

This example shows the same logic in a servlet:

```
ServletContext ctx = getServletContext();
nCtx = (com.netscape.server.IServerContext) ctx;
com.netscape.server.ICallerContext callerCtx = nCtx.getCallerContext();

/* Check if the client is in the PayrollDept role */
Identity PayRoll = nCtx.createIdentityByString("PayrollDept");
boolean B = callerCtx.isCallerInRole(PayRoll);
```

Controlling Execution Flow

You can test a client's membership in a role in order to control execution flow. For example, in the following EJB code snippet, if a client is a member of the Payroll department, one set of statements is executed, while if the client is not a member of Payroll, a different set of statements is executed.

```
/* Check if the client is in the PayrollDept role */
Identity PayRoll = EJBContext.createIdentityByString("PayrollDept");

if (callerCtx.isCallerInRole(PayRoll))
{
    // client can edit payroll entries
}
else
{
    // client can only view payroll entries
}
```

This example shows the same logic in a servlet:

```
ServletContext ctx = getServletContext();
nCtx = (com.netscape.server.IServerContext) ctx;
com.netscape.server.ICallerContext callerCtx = nCtx.getCallerContext();

/* Check if the client is in the PayrollDept role */
Identity PayRoll = nCtx.createIdentityByString("PayrollDept");

if (callerCtx.isCallerInRole(PayRoll))
{
    // client can edit payroll entries
```

```

    }
    else
    {
        // client can only view payroll entries
    }

```

Checking Permission

You can check whether a user or group has permission to execute a given resource by using `isAuthorized()` to access a programmatic ACL. For more information about ACLs, see “Access Control Lists” on page 233.

Note This is different from declarative security, described in “Declarative Security” on page 224, as this ACL is an arbitrary list and is not connected to any specific resource. You can define any permissions you want in a named ACL.

For example, assume you have an ACL named `HR_Database` that like this:

```

john_smith      write
fred_jones      readwrite
bill_white      read

```

In a servlet, you can conditionally execute a block of code based on the user’s stated permissions, as in the following example:

```

HttpSession session = req.getSession(); // assume an authorized session
HttpSession2 sess2 = (HttpSession2) session;
if ( sess2.isAuthorized(HR_Database, "read") )
{ ... }

```

In the above example, the conditional block of code executes if the user is `john_smith`.

Logging Out of the Session

When authentication is no longer necessary, you can log out of the session using `logoutSession()`. The session still exists, but it is no longer authenticated. Any subsequent calls to `getCallerIdentity()` return null.

For example, the following statement nullifies the authenticated identity, though the session remains:

```
sess2.logoutSession();
```

Authentication also ends if the session expires or is deleted.

Access Control Lists

You can control access to a resource by creating an access control list (ACL). There are two types of ACLs that serve different purposes:

- declarative, or component-specific, ACLs that are created in the component's configuration file
- programmatic, or arbitrary, ACLs that are created in the NAS registry directly using the NAS Administration Tool

These ACLs are described in the following sections.

Creating and Using Declarative ACLs

Components can declare a level of security for themselves by establishing component-specific ACLs in their configuration files. These ACLs determine which users or groups have permission to execute the resource.

If an ACL is present for a given servlet or EJB method, NAS checks the list to see whether the current user has permission to execute the resource. You do not have to write any code in the resource. In fact, the resource is not executed at all if the user is not permitted to execute the resource.

You can create ACLs specifically for a given servlet by specifying the `acl` field in the `ServletRegistryInfo` section of the servlet's configuration file. For more information, see "Specifying NAS Registry and ACL Information" in Chapter 10, "Creating Configuration Files."

You can create ACLs specifically for a given EJB by specifying values in the bean's property file. For more information, see "Specifying Access Control" in Chapter 10, "Creating Configuration Files." You can also create an ACL for a bean using the NAS Deployment Manager. For more information, see the *Administration Guide*.

Creating and Using Programmatic ACLs

Programmatic ACLs are named lists in the Directory Server that relate a user or group with a permission, as in the following example ACL:

```
john_smith    write
managers      readwrite
employees     read
```

In this example, the user `john_smith` and the group `managers` can both write to this resource, but `john_smith` (possibly a data-entry person) can not read from it. Members of the group `employees` can only read from the resource. You must then provide code in the resource to verify these permissions.

Once a client is authenticated and logged in, you can use the method `isAuthorized()` (in `HttpSession2`) to determine whether the current client is permitted to perform certain tasks. You provide the name of the ACL and the required permission.

The following example tests whether the current user is permitted to update a payroll record, and assume that there is an ACL called `PayrollACL` with permissions called `update` and `view`. This example is from a servlet. The same logic works in an EJB, though a client normally logs into a session from a servlet rather than from an EJB.

```
HttpSession session = req.getSession();
HttpSession2 sess2 = (HttpSession2) session;
if ( sess2.loginSession(username,password) ) //log into session
{
    if ( sess2.isAuthorized("PayrollACL", "update") )
    { updatePayrollEntries(); }
    else if ( sess2.isAuthorized("PayrollACL", "view") )
    { viewPayrollEntries(); }
    else
    { errorNotAllowed(); // return error page, not permitted }
}
```

ACLs and their permissions can be named anything. You create programmatic ACLs using the NAS Administration Tool. For more information, see “Setting Access Control List Authorization” in Chapter 6, “Securing Applications,” in the *Administration Guide*.

Taking Advantage of NAS Features

This chapter describes how to implement NAS features in your application.

NAS provides many additional features to augment your servlets for use in a NAS environment. These features are not a part of the official servlet specification, though some, like the servlet security paradigm described in Chapter 12, “Writing Secure Applications,” are based on emerging Sun standards and will conform to those standards in the future.

This chapter contains the following sections:

- Accessing the Servlet Engine
- Caching Servlet Results
- Using Application Events
- Sending and Receiving Email from NAS
- Netscape Application Builder Features

Accessing the Servlet Engine

The servlet engine controls all servlet functions, including instantiation, destruction, service methods, request and response object management, and input and output. The servlet engine in NAS is a special class called an AppLogic. AppLogics are NAS components that interact with the core server. In previous releases of NAS, AppLogics were a part of the application model, though for current and future releases they are solely available to access NAS internal features.

Each servlet is scoped in an AppLogic. You can access the AppLogic instance controlling your servlet using the method `getAppLogic()` in the NAS feature interface `HttpServletRequest2`. When you do this, you also gain access to server context. These activities are necessary to take advantage of other NAS features, as described in the other sections in this chapter.

Accessing the Servlet's AppLogic

To access the controlling AppLogic, cast the request object as an `HttpServletRequest2`. This interface provides access to the AppLogic via the method `getAppLogic()`, which returns a handle to the superclass.

The following example servlet header shows how to access the AppLogic instance:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.kivasoft.applogic.*;
import com.kivasoft.types.*;
import com.netscape.server.servlet.extension.*;

public class AppLogicTest extends HttpServlet {

    public void service(HttpServletRequest req,
                        HttpServletResponse res)
        throws ServletException, IOException
    {
        HttpServletRequest2 req2 = (HttpServletRequest2)req;
        AppLogic al = req2.getAppLogic();
        //al is now a handle to the superclass
        ...
    }
}
```

Accessing the Server Context

Some NAS features, such as Application Events (see “Using Application Events” on page 239), require an `IContext` object. `IContext` defines a view of the server context. For more information about `IContext`, see the entry for the `IContext` interface in the *NAS Foundation Class Reference*.

To obtain an `IContext` from a servlet, the standard servlet context can be cast to `IServerContext`, and from there, a `com.kivasoft.IContext` instance can be obtained, as in the following example:

```
ServletContext ctx = getServletContext();
com.netscape.server.IServerContext sc;
sc = (com.netscape.server.IServerContext) ctx;
com.kivasoft.IContext kivaContext = sc.getContext();
```

Alternatively, you can access the underlying `AppLogic` instance from a servlet, as described in “Accessing the Servlet’s `AppLogic`” on page 236, and obtain the context from the `AppLogic`’s context member variable, as in the following example:

```
HttpServletRequest req2 = (HttpServletRequest)req;
AppLogic al = req2.getAppLogic();
com.kivasoft.IContext kivaContext = al.context;
```

From an EJB, the standard `javax.ejb.SessionContext` or `javax.ejb.EntityContext` can be cast to `IServerContext`, and from there, a `com.kivasoft.IContext` instance can be obtained, as in the following example:

```
javax.ejb.SessionContext m_ctx;
....
com.netscape.server.IServerContext sc;
sc = (com.netscape.server.IServerContext) m_ctx; /
com.kivasoft.IContext kivaContext;
kivaContext = sc.getContext();
```

Caching Servlet Results

NAS has the ability to cache the results of a servlet in order to make subsequent calls to the same servlet faster. NAS caches the results of a request (i.e. a servlet’s execution) for a specific amount of time, so that if another call for that data happens, NAS can just return the cached data rather than having to

perform the operation again. For example, if your servlet returns a stock quote that you only want to update every 5 minutes, you could set the cache to expire after 300 seconds.

Whether to cache results, and how to cache them, depends on the type of data involved. It makes no sense to cache the results of a quiz submission, for example, because the input to the servlet is different each time. However, you could cache a high-level report showing demographic data taken from quiz results and updated once an hour.

You can define how a NAS 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 NAS feature. For more information on servlet configuration files, see Chapter 10, "Creating Configuration Files."

Set the following variables in the `ServletData` section of the servlet's configuration file:

Name	Type	Value
<code>CacheTimeOut</code>	Int	Optional. Elapsed time (in seconds) before the memory cache for this servlet is released.
<code>CacheSize</code>	Int	Optional. Size (in kb) of the memory cache for this servlet.
<code>CacheCriteria</code>	Str	Optional criteria expression containing a string of comma-delimited descriptors. Each descriptor defines a match with one of the input parameters to the servlet.

You use the `CacheCriteria` field to set criteria used to determine whether the results of a servlet should be cached. The `CacheCriteria` field contains a test for one or more fields in the request. This allows you to conditionally cache results based on the value or presence of one or more form fields. If the tests succeed, the servlet results are cached.

Use the follow syntax for the `CacheCriteria` field:

Syntax	Description
<i>arg</i>	Test succeeds for any value of <i>arg</i> in the input parameter list. For example, if the field is set to "EmployeeCode", the results are cached if the request contains a field called EmployeeCode.
<i>arg=v</i>	Test whether <i>arg</i> matches <i>v</i> (a string or numeric expression). For example, if the field is set to "stock=NSCP", the results are cached if the request contains a field called stock that has the value NSCP. Assign an asterisk (*) to the argument to cache a new set of results every time the servlet runs with a different value. For example, if the criteria is set to "EmployeeCode=*", the results are cached if the request object contains a field called EmployeeCode and the value is different from the value used to create the currently cached result.
<i>arg=v1 v2</i>	Test whether <i>arg</i> matches any values in the list (<i>v1</i> , <i>v2</i> , and so on). For example: "dept=sales marketing support"
<i>arg=n1-n2</i>	Test whether <i>arg</i> is a number that falls within the given range. For example: "salary=40000-60000"

Using Application Events

In a NAS environment, you can create and use named events. The term *event* is widely used to refer to user actions, such as mouse clicks, that trigger code. However, the events described in this section are not caused by users. Rather, an event is a named action that you register with the NAS. The event occurs either when a timer expires or when the event is activated from application code at run time.

Events are stored persistently in the NAS, and are removed only when your application explicitly deletes them. Typical uses for events include periodic backups, reconciling accounts at the end of the business day, or sending alert messages. For example, you can set up an event that sends an email to alert your company's buyer when inventory levels drop below a certain level.

Each event has a name, a timer (optional), and one or more actions to take when the event is triggered. Application events have the following characteristics:

- Each event can cause the execution of one or more actions, which can include sending email or calling another application component.

- Actions can be synchronous or asynchronous with the calling environment.
- Multiple actions can be configured to execute concurrently with one another, or serially, one after the other.
- Multiple actions are executed in a specific order (the order in which they are registered).
- Request data can be passed to an application event in an `IVallList` object.

You can set up events to occur at specific times or at intervals, such as every hour or once a week. You can also trigger an event by calling the event by name from code. When an event's timer goes off or it is called from code, the associated action occurs.

The Application Events API

NAS uses two interfaces to support events:

- The `IAppEventMgr` interface manages application events. This interface defines methods for creating, registering, triggering, enabling, disabling, enumerating, and deleting events.
- The `IAppEventObj` interface represents the defined events an application supports. This interface defines methods not only for getting or setting attributes of an event, but also for adding, deleting, or enumerating actions of the event.

These two interfaces are described in the following sections.

The `IAppEventMgr` Interface

You can perform any of the following administrative tasks with an event by using the associated methods in the `IAppEventMgr` interface:

Method	Description
<code>createEvent()</code>	Creates an empty application event object.
<code>deleteEvent()</code>	Removes a registered event from NAS.
<code>disableEvent()</code>	Temporarily disables a registered event.

Method	Description
<code>enableEvent()</code>	Enables a registered event.
<code>enumEvents()</code>	Enumerates through the list of registered events.
<code>getEvent()</code>	Retrieves the <code>IAppEventObj</code> for a registered event.
<code>registerEvent()</code>	Registers a named event for use in applications.
<code>triggerEvent()</code>	Triggers a registered event.

The IAppEventObj Interface

The event's behavior is determined by its attributes, which define how and when an event executes, and its actions, which define what the event does when it is triggered. To set and examine attributes and actions, use the methods in the `IAppEventObj` interface:

Method	Description
<code>addAction()</code>	Appends an action to an ordered list of actions.
<code>deleteActions()</code>	Deletes all actions added to this <code>IAppEventObj</code> .
<code>enumActions()</code>	Enumerates the actions added to this <code>IAppEventObj</code> .
<code>getAttributes()</code>	Retrieves the list of attributes of an <code>IAppEventObj</code> .
<code>getName()</code>	Retrieves the name of the <code>IAppEventObj</code> .
<code>setAttributes()</code>	Sets a list of attribute values for the <code>IAppEventObj</code> .

For more details about these methods, see the *Netscape Application Server Foundation Class Reference*.

Creating a New Application Event

Follow these steps to create a new application event and register it with NAS. For more information about the interfaces and methods described in each step, see the entries for `IAppEventMgr` and `IAppEventObj` in the *NAS Foundation Class Reference*.

Note A `com.kivasoft.IContext` object is required to create an event. For more information, see “Accessing the Server Context” on page 237.

1. You first need the context parameter, an `IContext` object which provides access to NAS services. Obtain the context parameter from the server context `IServerContext`, as in the following example:

```
ServletContext ctx = getServletContext();
com.netscape.server.IServerContext sc;
sc = (com.netscape.server.IServerContext) ctx;
com.kivasoft.IContext ic = sc.getContext();
```

2. Next, use the `GetAppEventMgr()` method in the `GXContext` class to create an `IAppEventMgr` object:

```
IAppEventMgr mgr = com.kivasoft.dlm.GXContext.GetAppEventMgr(ic);
```

3. After creating the `IAppEventMgr` object, you can create an application event—an instance of `IAppEventObj`—by calling `createEvent()` on the `IAppEventMgr` object, as in the following example:

```
IAppEventObj evtObj = mgr.createEvent("myEvent");
```

4. You now have an empty event in the system. Create an `IValList` object to hold the event's attributes, and then set the attributes to the requirements you have for this event. Finally, assign the attributes to your event. For example, this event executes at 5:00 AM every morning:

```
IValList atts = GX.CreateValList();
atts.setValString(GXConstants.GX_AE2_RE_KEY_TIME, "5:0:0 */*/*");
evtObj.setAttributes(atts);
```

5. You can assign actions to the event in order to cause it to do something when it is triggered. This procedure is similar to creating event attributes; first create an `IValList` object to contain the actions, then assign them to the event. For example, this event runs a servlet called `myServlet`:

```
IValList action = GX.CreateValList();
action.setValString(GXConstants.GX_AE2_RE_KEY_SERVLET, "myServlet");
evtObj.addAction(action);
```

6. You must then register the event, or make NAS aware of it, by calling `registerEvent()`. Further, you must also instruct NAS to enable the event for access by calling `enableEvent()`. The following example shows the registration and enabling of an event:

```
if (mgr.registerEvent("myEvent", evtObj) != GXE.SUCCESS)
    return streamResult("Cannot register RepGenEvent<br>");
```

7. Once the event is registered and enabled, you can trigger it by hand if you want. To trigger the event, use the `IAppEventMgr` method `triggerEvent()`, as in the following example:

```
mgr.triggerEvent("myEvent");
```

Sending and Receiving Email from NAS

NAS supports email transactions through the `IMailbox` interface. For more information, see the entry for the `IMailbox` interface in the *NAS Foundation Class Reference*.

In order for email applications to work, you must have access to an SMTP server if you want to send email and a POP server if you want to receive email.

Security in Email

Security is often a concern when sending or receiving email. If the application generates and sends email using user input to set the address or content, then there is a risk of propagating inappropriate messages or mailing to incorrect recipients. Be sure to validate all user input before incorporating it in email.

Accessing the Controlling AppLogic

Using the `IMailbox` interface from a servlet requires access to the `AppLogic` instance that controls the servlet's functions. For more information, see "Accessing the Servlet Engine" on page 236.

For example, before you can use the `IMailbox` interface, create a handle to the `AppLogic` instance that controls your servlet:

```
HttpServletRequest req2 = (HttpServletRequest)req;
AppLogic al = req2.getAppLogic();
```

The examples in this section assume that the above code exists in your servlet.

Receiving Email

To receive email, your application must have access to a POP server.

Before retrieving messages, you can use `retrieveCount()` to see how many messages are waiting in the specified inbox on the mail server. By checking first, you can avoid attempting to retrieve messages if the mailbox is empty. You can also use this technique when you need to know how many messages are waiting in order to construct a loop that iterates through them one by one.

To retrieve messages, call `retrieve()`. Depending on the parameters you pass to this method, you can customize the retrieval process in the following ways:

- Retrieve all messages
- Retrieve only unread messages
- Delete messages from the mail server as they are retrieved

Only those messages received before the last call to `open()` are retrieved. You can not open a mailbox session, leave it open, and continuously receive email messages. Instead, you must open a new session each time you want to retrieve new email.

After retrieving messages, you can return the mailbox to its original state by calling `retrieveReset()`. This method undeletes and unmarks any messages that were affected by the previous `retrieve()` call.

To receive email

1. Create an instance of `IMailbox` by calling `createMailbox()` from the servlet's controlling `AppLogic` instance ("Accessing the Controlling `AppLogic`" on page 243). In this call, you specify valid user information and the name of the POP server you want to access. For example:

```
IMailbox mb;
mb = al.createMailbox("mail.myOrg.com", "myUserName",
    "pass7878", "sid@blm.org");
```

2. Open a session on your POP server by calling `open()` with the `OPEN_RECV` flag. For example:

```
result = mb.open(GX_MBOX_OPEN_FLAG.OPEN_RECV);
```

3. To find out whether you have messages, call `retrieveCount()`. For example:

```
int mbCount = mb.retrieveCount();
```

4. To retrieve messages, instantiate an `IVallList` object to contain the email messages, then call `retrieve()`. For example, the following code retrieves the latest unread messages and does not delete them from the mailbox:

```
IVallList messages = GX.CreateValList();
messages = mb.retrieve(true, false);
```

Only the messages received before the call to `open()` are retrieved.

5. To undo changes, call `retrieveReset()`. For example:

```
result = mb.retrieveReset();
```

6. To close the session, call `close()`. For example:

```
mb.close();
```

You can have only one mail server session open at a time. For example, suppose you open a session with the `OPEN_RECV` flag, then want to send email. You must first close the existing session, then open another one with the `OPEN_SEND` flag.

Example

The following code retrieves email in a servlet:

```
// Create mailbox object to connect to a POP mail server
IMailbox recvMB;
public void recvMail()
{
    // Only check messages received after the last open
    boolean Latest = true;
    // Remove retrieved messages from the mail server
    boolean Delete = true;

    // Create an IMailbox instance
    HttpServletRequest2 req2 = (HttpServletRequest2)req;
    AppLogic al = req2.getAppLogic();
    IMailbox recvMB;
    recvMB = al.createMailbox(recvhost, user, pswd, useraddr);

    if (recvMB != null)
    {
```

```

        if (recvMB.open(GX_MBOX_OPEN_FLAG.OPEN_RECV))
        {
            // Count the number of new messages
            int numMsgs = recvMB.retrieveCount();
            if(numMsgs > 0)
            {
                IValList msgList;
                // Retrieve the new messages
                msgList = recvMB.retrieve(Latest,Delete);

                // Use IValList methods to iterate through
                // the returned IValList. The keys in the
                // IValList are the message numbers. The
                // values are the email messages as strings
            }
            recvMB.close();
        }
    }
}

```

Sending Email

To send email, your application must have access to an SMTP server. Construct the email address and message separately, then use the `send()` method to send the email out through the server.

You can send email to a single recipient or to a group of recipients. To send email to a group, use one of the following techniques:

- Pass the email addresses to `send()` as an array.
- Use a loop to send a series of messages one at a time.

You can populate an address array dynamically using the results of a query, in which each row returned by the query is one email address. Use a loop to iterate through the rows in the query's result set and assign the data to successive elements of the array.

To send email

1. Create an instance of `IMailbox` by calling `createMailbox()` from the servlet's controlling `AppLogic` instance ("Accessing the Controlling `AppLogic`" on page 243). In this call, you specify valid user information and the name of the POP server you want to access. For example:

```
IMailbox mb;
mb = al.createMailbox("mail.myOrg.com",
                     "myUserName",
                     "pass7878",
                     "sid@blm.org");
```

2. Open a session on your SMTP server by calling `open()` with the `OPEN_SEND` flag. For example:

```
int result = mb.open(GX_MBOX_OPEN_FLAG.OPEN_SEND);
```

3. To send the message, call `send()`. Pass a single email address or an array of addresses to this method, along with the text of the message. For example:

```
java.lang.String[] ppTo = {"sal@dat.com", "sid@blm.com", null};
int mbSend = mb.send(ppTo, "Testing email");
```

4. To close the session, call `close()`. For example:

```
mb.close();
```

You can have only one mail server session open at a time. For example, suppose you open a session with the `OPEN_SEND` flag, then want to retrieve your email. You must first close the existing session, then open another one with the `OPEN_RECV` flag.

Example

The following code sends email in a servlet:

```
// Define the string parameters that will be passed
// to IMailbox methods
String sendhost = "smtp.kivasoft.com";
String rcvhost = "pop.kivasoft.com";
String user = "eugene";
String pswd = "eugenesSecretPassword";
String useraddr = "eugene@kivasoft.com";
String sendTo[] = {"friend@otherhost.net", null};
String mesg = "Hi Friend, How are you?";
public void sendMail()
```

```

{
    // Create an IMailbox instance
    HttpServletRequest2 req2 = (HttpServletRequest2)req;
    AppLogic al = req2.getAppLogic();
    IMailbox sendMB;
    sendMB = al.createMailbox(sendhost, user, pswd, useraddr);

    if (sendMB != null) // sendMB successfully created
    {
        // Open a session with the mail server
        if (sendMB.open(GX_MBOX_OPEN_FLAG.OPEN_SEND))
        {
            // Send a mail message
            sendMB.send(sendTo,mesg);
            // Close the mailbox session
            sendMB.close();
        }
    }
}

```

Netscape Application Builder Features

NAS includes APIs that were designed for use by the code generated from Netscape Application Builder (NAB) wizards. These APIs are available to servlet programmers, although we recommend that you use NAB to create servlets that use these features.

The features presented here include:

- Validating Form Field Data
- Creating Named Form Action Handlers
- Example Validation and Form Action Handler

Validating Form Field Data

You can set a servlet to automatically check form fields for certain types of values. Additionally, you can create a named error handler to control application flow if the validation fails.

In short, you specify the rules for validation in a servlet's configuration file, and then call the `HttpServletRequest2` method `validate()` to test the fields against the validation rules. If validation fails, you can let NAS generate an error page automatically, or you can provide an error handler method in your servlet to produce an error message.

Validation Methods

NAS provides a method called `validate()` that validates all the form fields configured in the servlet's configuration file. This method is defined in the NAS feature interface `HttpServletRequest2`. To use this interface, cast the standard request object to it, as in the following example:

```
public void service (HttpServletRequest req,
                    HttpServletResponse res)
    throws ServletException, IOException
{
    HttpServletRequest2 req2 = (HttpServletRequest2) req;
```

There are two method signatures for `validate()`, as shown here:

```
public boolean validate(HttpServletResponse response)
    throws IOException;

public boolean validate(HttpServletResponse response,
                        HttpServletRequest servlet,
                        String errHandlerName)
    throws ServletException, IOException;
```

In the first form, if a validation error occurs, the system automatically generates an error response page. In the second form, you pass a servlet and method name in the `servlet` and `errHandlerName` parameters, respectively, that correspond to a named error handler that you write. (See "Error Handlers" on page 251)

`validate()` returns `true` if the input matches the validation rule, or `false` otherwise.

Validation Rules

Input data are validated based on the rules configured for each form field in the servlet's configuration file, in the `Parameters` section. This section resides in the `ServletData` section. The `Parameters` section enables you to specify the following information for each form field:

Name	Type	Value
<code>inputRequired</code>	Str	Optional, indicates whether an input value must be supplied for that parameter. Valid values are y or n, the default is n.
<code>inputRule</code>	Str	Optional, specifies how the named parameter should be validated. <code>inputRule</code> indicates the type of data the value should be validated against.

You can also specify a flag `inputRequired` (set to y or n) for each parameter. This flag indicates whether the specified value must exist in the input stream. If the variable is not present, validation fails.

For more information about the `Parameters` section in servlet configuration files, see “Servlet Configuration File Fields” in Chapter 10, “Creating Configuration Files.”

The following types of data can be checked:

Data to Validate	Validation Rule	Description
Number	<code>VALIDATE_NUMBER</code>	Data is numerical.
Integer	<code>VALIDATE_INTEGER</code>	Data is an integer.
Positive integer	<code>VALIDATE_POSITIVE_INTEGER</code>	Data is a positive integer.
Alphabetic	<code>VALIDATE_ALPHABETIC</code>	Data is alphabetic, a-z and/or A-Z.
US phone number	<code>VALIDATE_US_PHONE</code>	Data consists only of 10 digits and optionally the characters (,), and -.
International phone number	<code>VALIDATE_INTL_PHONE</code>	Data consists only of numbers and the characters (,), +, and -.
Email	<code>VALIDATE_EMAIL</code>	Data is in the format <i>a@b.c</i>
Social Security Number	<code>VALIDATE_SSN</code>	Data consists only of 9 digits and optionally the character -.
Date	<code>VALIDATE_DATE</code>	Data is in the format "MM-DD-YY" or "MM-DD-YYYY". Month, day, and year are validated accordingly.

Data to Validate	Validation Rule	Description
Day	VALIDATE_DAY	Data is an integer between 1 and 31 inclusive.
Month	VALIDATE_MONTH	Data is an integer between 1 and 12 inclusive.
Year	VALIDATE_YEAR	Data is an integer between 1 and 99 inclusive, or between 1000 and 9999 inclusive.
US zipcode	VALIDATE_US_ZIPCODE	Data consists of only 5 or 9 digits and optionally the character -.

Error Handlers

You can create methods in your servlet to handle specific validation failures. These methods are called error handlers, and generally follow this method signature:

```
public void myErrorHandler(HttpServletRequest req,
                          HttpServletResponse res);
```

Use this method to generate an error page if the `validate()` method returns false, indicating a validation error. (For information on the `validate()` method, see “Validation Methods” on page 249).

You can examine the type of error using an error vector of type `Vector`, using the following methods in the `HttpServletRequest2` interface:

Error Handler Vector Method	Description
<code>getErrorCodes()</code>	Returns a vector of error codes that correspond to the input variables that failed to validate. An error code is associated with each type of data validated.
<code>getErrorMsgs()</code>	Returns a vector of error messages that correspond to the input variables that failed to validate.
<code>getErrorVars()</code>	Returns a vector of the input variables that failed to validate.

The following table shows the error code and message associated with each validation rule:

Validation Rule	Error Code	Error Message
VALIDATE_NUMBER	ERROR_NUMBER	Wrong number format!
VALIDATE_INTEGER	ERROR_INTEGER	Wrong integer format!
VALIDATE_POSITIVE_INTEGER	ERROR_POSITIVE_INTEGER	Wrong positive integer format!
VALIDATE_ALPHABETIC	ERROR_ALPHABETIC	Wrong alphabetic format!
VALIDATE_US_PHONE	ERROR_US_PHONE	Wrong US phone format!
VALIDATE_INTL_PHONE	ERROR_INTL_PHONE	Wrong international phone format!
VALIDATE_EMAIL	ERROR_EMAIL	Wrong email format!
VALIDATE_SSN	ERROR_SSN	Wrong social security format!
VALIDATE_DATE	ERROR_DATE	Wrong date format!
VALIDATE_DAY	ERROR_DAY	Wrong day format!
VALIDATE_MONTH	ERROR_MONTH	Wrong month format!
VALIDATE_YEAR	ERROR_YEAR	Wrong year format!
VALIDATE_US_ZIPCODE	ERROR_ZIP	Wrong zip code format!

Example Validation Rules

The following example shows the `Parameters` section from a servlet configuration file. This section describes a form consisting of several parameters, including a name, social security number, an address, an email address, and a US phone number:

```
"Parameters"    NTV {
    "name"      NTV {
        "inputRequired"    Str    "y",
    },
    "zip"       NTV {
        "inputRequired"    Str    "y",
        "inputRule"        Str    "VALIDATE_US_ZIPCODE",
    },
    "ssn"       NTV {
```

```

        "inputRequired"      Str      "y",
        "inputRule"          Str      "VALIDATE_SSN",
    },
    "email"  NTV {
        "inputRequired"      Str      "n",
        "inputRule"          Str      "VALIDATE_EMAIL",
    },
    "phone"  NTV {
        "inputRequired"      Str      "y",
        "inputRule"          Str      "VALIDATE_US_PHONE",
    }
}

```

In this example, the user does not need to supply an email address but must supply a name, zip code, a social security number, and a US phone number in the form. The zip code, social security number, phone number, and email (if it is present) are checked for valid data.

Note that this validation does not fail if the email value is missing, only if it is present and does not match the `VALIDATE_EMAIL` rule.

Creating Named Form Action Handlers

You can create methods that handle particular buttons on a form. This enables you to build in a level of modularity to your servlet to handle requests cleanly.

Form handlers are used in code generated by Netscape Application Builder (NAB). Usage consists of two methods in the `HttpServletRequest2` interface, coupled with entries in the servlet's configuration file.

Use the following methods in `service()` (generic servlets) or `doGet()` or `doPost()` (HTTP servlets) to handle requested actions. These methods reside in the `HttpServletRequest2` interface. Note that you must also configure form action handlers in the `FormActionHandlers` section of the servlet's configuration file.

Method	Description
<code>dispatchAction()</code>	Calls a method corresponding to a form action in a servlet.
<code>formActionHandlerExists()</code>	Determines whether the servlet has form actions defined on it. This method is used in code generated by NAB, and typically is not necessary for non-generated code.

For more information about the `FormActionHandlers` section in the servlet configuration file, see “Specifying Servlet Metadata, Result Cache, Required Session Variables, and NAS Features” in Chapter 10, “Creating Configuration Files.”

Example Validation and Form Action Handler

This example shows a servlet and its configuration file. The servlet performs some validation on the incoming request and then passes it to a form action handler called `submitHandler()`.

Servlet Configuration File

```
NTV-ASCII {
    "DispatchServlet"    NTV    {
        "ServletRegistryInfo"    NTV    {
            "type"                Str    "j",
            "enable"              Str    "y",
            "encrypt"             Str    "n",
            "lb"                  Str    "y",
            "descr"               Str    "Testing action dispatch",
            "group"               StrArr ["Actions"],
            "guid"                Str    "{6952A1AC-FED2-1687-9BB6-080020A16896}",
        },
        "ServletRunnerInfo"      NTV    {
            "ServletClassPath"    Str    "com.netscape.server.servlet.test.TestDispatchServlet",
        },
    },
}
```

```

        "ServletData"      NTV      {
            "FormActionHandlers"      NTV      {
                "submitAction"      Str      "submitHandler",
            },
        },
    },
}

```

Servlet Source Code

```

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

/**
 * An HTTP Servlet that responds to the GET and HEAD methods of the
 * HTTP protocol. It returns a form to the user that gathers data.
 * The form POSTs to another servlet.
 */

public class TestDispatchServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Get the user's session and shopping cart
        out.println("<html>"
            + "<head><title> TestDispatch </title></head>"
            + "<body bgcolor=\"#ffffff\">\n"
            + "<br>");

        out.println("<form action=\""
            + response.encodeUrl("/servlet/DispatchServlet")
            + "\" method=\"post\">"
            + "<strong>Please Update your account information"
            + "</strong><br><br><br>"
            + "<table>"
            + "<tr>"
            + "<td><strong>Your Name:</strong></td>"
            + "<td><input type=\"text\" name=\"personname\" "
            + "\" size=\"19\"></td>"
            + "</tr>"

            + "<tr>"

```

```

+ "<td><strong>Account ID:</strong></td>"
+ "<td><input type=\"text\" name=\"accountID\" "
+ "\" size=\"19\"></td>"
+ "</tr>"

+ "<tr>"
+ "<td><strong>Your Password:</strong></td>"
+ "<td><input type=\"password\" name=\"password1\" "
+ "\" size=\"19\"></td>"
+ "</tr>"

+ "<tr>"
+ "<td><strong>Match Password:</strong></td>"
+ "<td><input type=\"password\" name=\"password2\" "
+ "\" size=\"19\"></td>"
+ "</tr>"

+ "<tr>"
+ "<td></td>"
+ "<td><input type=\"submit\" name=\"submitAction\" "
+ "value=\"Submit Information\"></td>"
+ "</tr>"

+ "</table>"
+ "</form>"
+ "</td></tr></table></body>"
+ "</html>");
    out.close();
}

public void doPost (HttpServletRequest request,
                    HttpServletResponse response)
                    throws ServletException, IOException
{
    HttpServletRequest2 newReq = (HttpServletRequest2) request;
    if( newReq.validate(response)) {
        newReq.dispatchAction(response,this);
    }
}

public int submitHandler ( HttpServletRequest request,
                           HttpServletResponse response)
                           throws ServletException, IOException
{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String p1 = request.getParameter("password1");
    String p2 = request.getParameter("password2");

```



```

if( p1 == null ||
    p2 == null ||
    ! p1.equals(p2) )
{
    out.println("<html>"
        + "<head><title> TestDispatch </title></head>"
        + "<body bgcolor=\"#ffffff\">\n"
        + "<br><br>Your password does not match! "
        + "Please try again"
        + "</body>"
        + "</html>");
    out.close();
    return HttpServletRequest2.ERROR_USER;
}
out.println("<html>"
    + "<head><title> TestDispatch </title></head>"
    + "<body bgcolor=\"#ffffff\">\n"
    + "<br><br>Your Account information: <br><br>"
    + "<br>Your name: "
    + request.getParameter("personname")
    + "<br>Account ID: "
    + request.getParameter("accountID")
    + "<br><br>Updated successfully in the database"
    + "</body>"
    + "</html>");
out.close();
return HttpServletRequest2.NO_ERROR;
}
}

```


Inside the Online Bookstore Sample Application

This chapter describes the design, file components, and functionality of the online bookstore sample application.

The following topics are presented in this chapter:

- Design Overview
- Presentation Flow
- Directory Structure
- Servlets
- EJB Functionality

Design Overview

The Online Bookstore sample application, which is shipped with NAS, simulates an Internet e-commerce site where customers can search for books, show details, add books to their shopping cart, place orders, and register for billing. Furthermore, bookstore managers can access information about customer usage and order status.

The user interface is a web browser that displays HTML pages and JavaServer pages. The application was designed to run on Windows NT. In addition, the application can access an Oracle8 database and an LDAP server such as Netscape Directory Server.

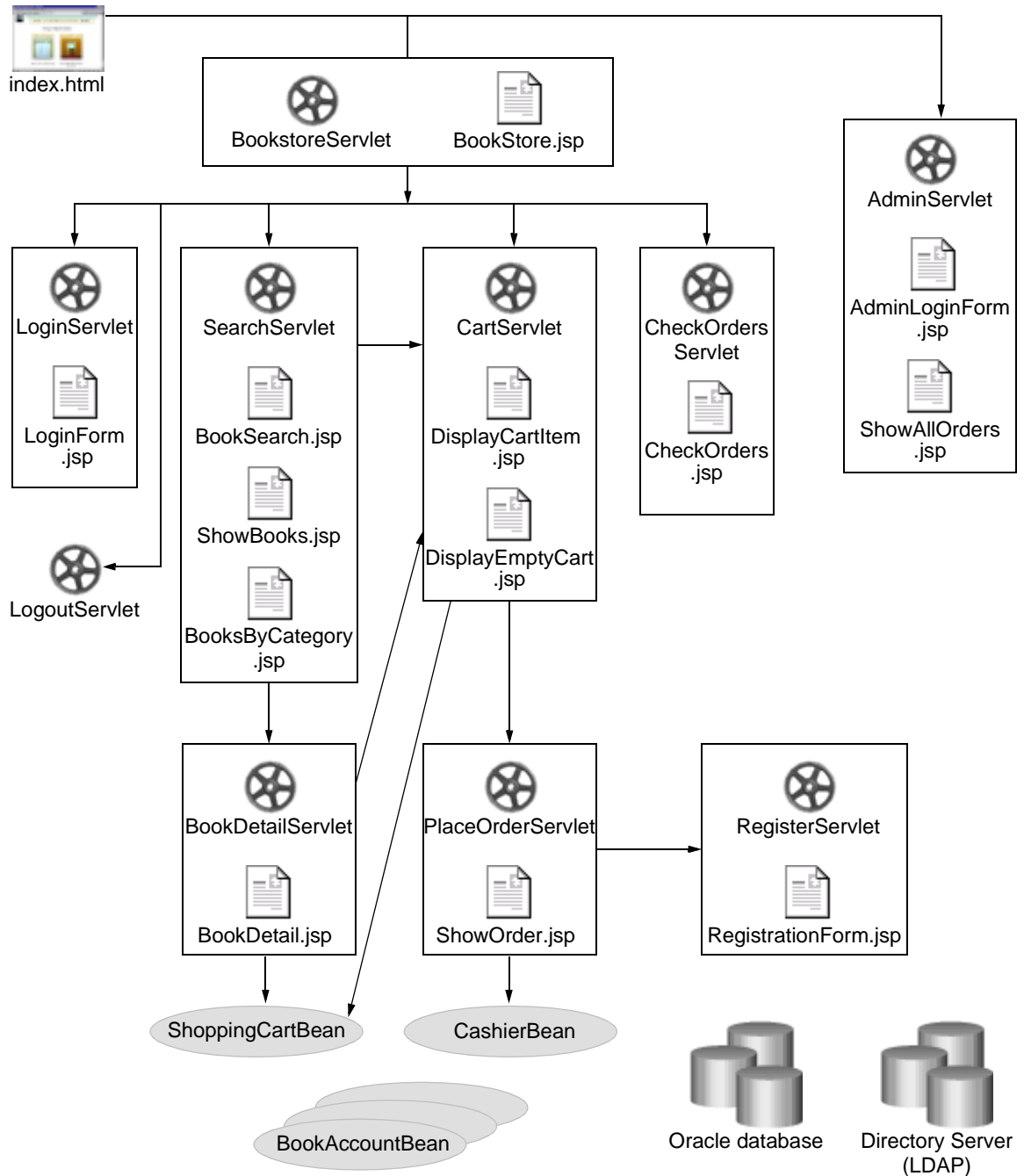
Application Model

The Online Bookstore demonstrates the NAS application model by incorporating the following features:

- HTML
- JSPs
- servlets
- EJBs (stateful session bean, stateless session bean, and entity bean)
- database access through helper classes and JDBC APIs
- transaction processing, both local and global
- integration with an LDAP server

Application Flow

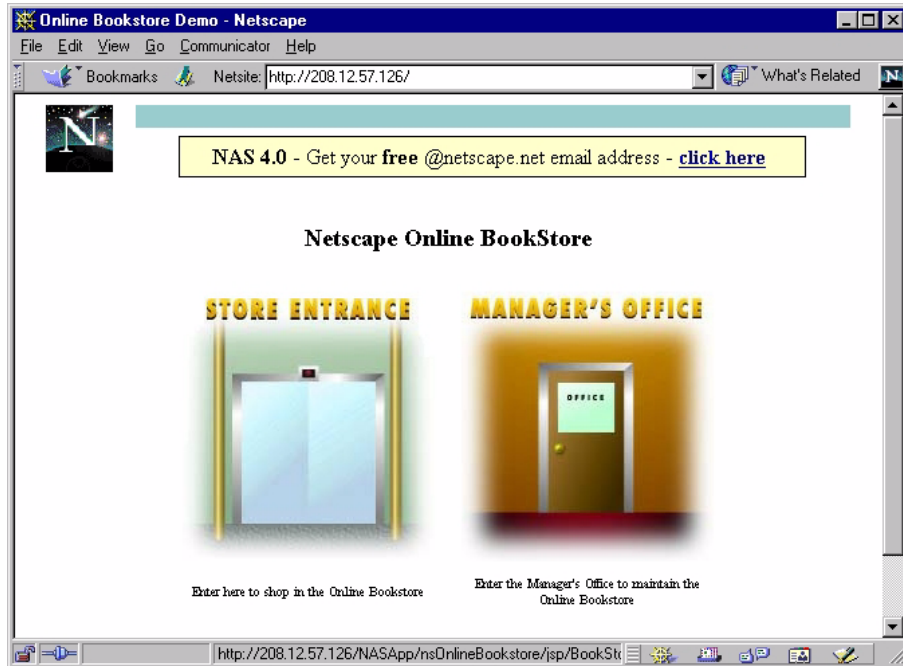
The flowchart on the following page summarizes the design of Online Bookstore. You can think of this application as a set of functional modules, each controlled by a servlet. This functional grouping is conceptual, not structural—a convenience for better understanding the application flow. This modular approach encourages parallel code development, through which different teams handle each design module at the same time, with minimal overlap.



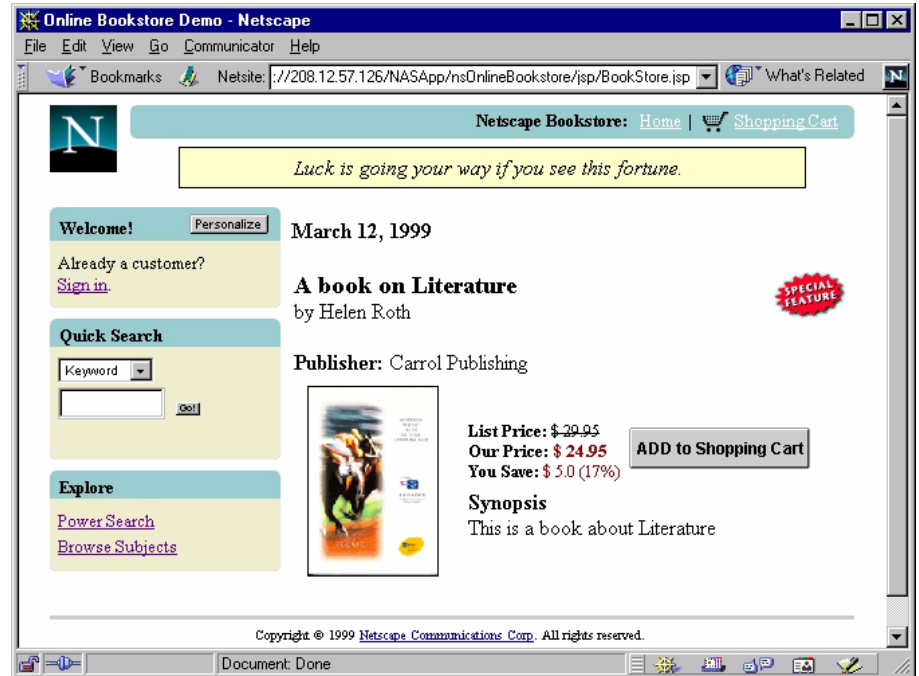
Presentation Flow

The following sequence describes a typical user session with the Online Bookstore:

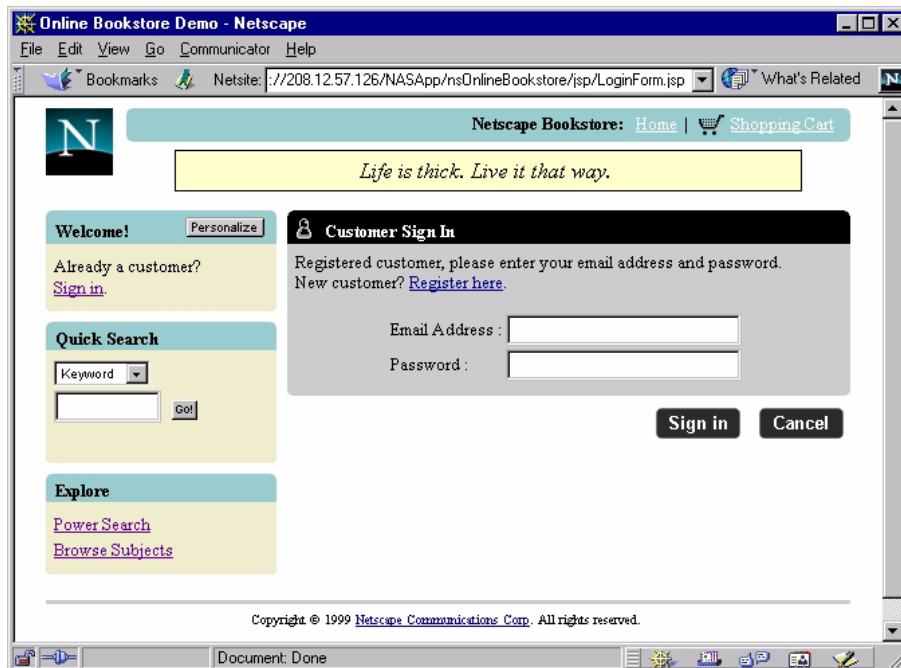
1. From a web browser, the user navigates the Internet to reach the first page of the application, index.html.



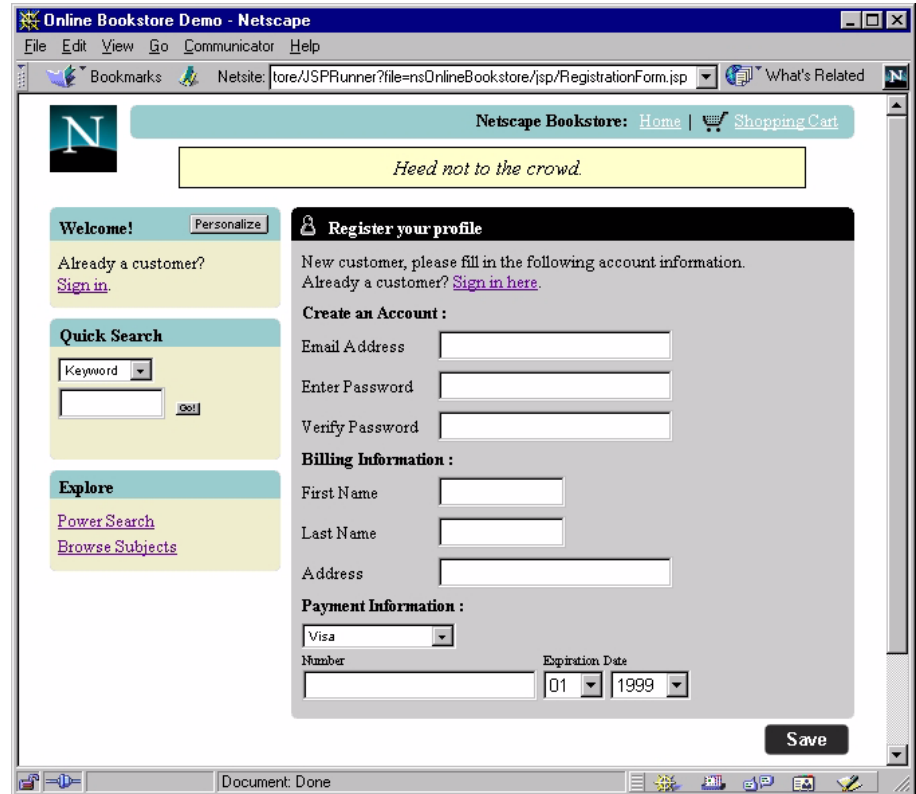
2. From index.html, users can follow two main branches: customers can click the Store Entrance image to display BookStore.jsp, and application administrators can click the Manager's Office image to invoke AdminServlet.
3. As a typical user, a customer clicks the Store Entrance image. The application then displays the main customer page, BookStore.jsp.



4. From the BookStore.jsp page, a customer can browse books by subject, search books by keyword, or add books to a shopping cart. Any of these tasks can be performed as an anonymous user. However, before placing an order or checking previous orders, a customer must sign in.
5. Clicking the **Sign In** link invokes the Login servlet. This servlet causes the right side of the page to display the Customer Sign In form, a JSP named LoginForm.jsp.



6. From the Customer Sign In form, customers can take either of two actions. Customers who previously registered can enter their email address and password, whereas new customers must click the **Register here** link.
7. Clicking **Register here** invokes the Register servlet. This servlet causes the right side of the page to display the Registration form, a JSP named RegistrationForm.jsp.



8. Regardless of whether a user is registered, the user can perform different types of searches from the left side of the page, using either the Quick Search table or the Explore table. These page elements invoke the SearchServlet.
9. When a results page appears, users can click a book title hyperlink. Clicking a book title invokes the BookDetail servlet, which in turn displays a JSP of item details.
10. If the book is of interest, the user presses a button to add the item to the shopping cart. This action invokes the CartServlet, which in turn displays a JSP of selected books. The displayed prices are calculated with the help of ShoppingCartBean.

11. When the user is ready to buy the items in the shopping cart, the user presses the Place Order button. This action invokes the PlaceOrder servlet. This servlet uses the CashierBean to calculate the charges, then displays the charges using the ShowOrder JSP.
12. The user can continue to navigate the application, searching for more books or checking previous orders, for example.
13. Eventually, the user clicks the Logout hyperlink to invoke the Logout servlet.

Directory Structure

The sample application includes component files organized into the following directories.

Directory	Contents
account	The BookAccount bean and related files to support accounting and inventory.
cart	The Cart bean and related files to support the selection of books.
cashier	The Cashier bean and related files to support the buying of books.
custom	Files involved with customizing the look and feel of the user interface. This functionality is invoked using the Personalize button on the Bookstore JSP.
database	Helper classes and other code for connecting to the database.
images	GIF and JPG files used in the web pages.
jsp	JavaServer Pages
ldap	A java file and property file to control LDAP access.
ntv	The .gxr file (used in deployment) and various configuration files.
util	Java files helpful to application developers.

In addition, many servlet files reside at the same level as the previous directories. These servlets are described in the next section.

Servlets

This section describes the servlets used in the Online Bookstore. The servlets are listed in the approximate order they would be encountered during a typical session.

BookstoreServlet, called from `index.html`, displays the Bookstore JSP. This JSP is the main customer page. Several of the application's servlets are invoked from the Bookstore JSP.

LoginServlet is invoked from the Bookstore JSP. This servlet launches `LoginForm.jsp`, which in turn displays the Customer Sign In form. `LoginServlet` verifies a customer's email ID and password against records in the database, then saves this information into an HTTP session.

SearchServlet is invoked from the Bookstore JSP. This servlet implements logic for basic keyword searches, for detailed, form-based searches, and for browsing by categories. `SearchServlet` also launches the JSPs necessary to input the search criteria or to display the results.

BookDetailServlet is called from any book title link, as might appear in `ShowBooks.jsp` and `BooksByCategory.jsp`. The servlet validates the `bookID` parameter and launches `BookDetails.jsp` to display more information about the item.

CartServlet manages the addition, deletion, and modification of shopping cart items. This servlet is called by the "Shopping Cart" link on the Bookstore JSP, as well as by buttons labeled "Add to Shopping Cart" or "Delete from Shopping Cart." `CartServlet` launches either of two JSPs: `DisplayCartItem.jsp` or `DisplayEmptyCart.jsp`. The `ShoppingCartBean` EJB determines the items and prices that appear.

PlaceOrderServlet is invoked from the "Place Order" button on `DisplayCartItem.jsp`. The servlet checks the customer object saved in the session. If the customer is registered, then control passes to the `CashierBean` EJB, the order is processed, and the results appear by way of `ShowOrder.jsp`. If the customer is not registered, then control passes to `RegisterServlet`.

RegisterServlet launches `RegistrationForm.jsp`, in which customers enter personal information. When users submit their data, `RegisterServlet` connects to a database, inserts the data, and saves the information into an HTTP session.

RegisterServlet is invoked under two circumstances: when the user clicks “Register Here” in the LoginForm JSP, or when the PlaceOrder servlet detects an unregistered user.

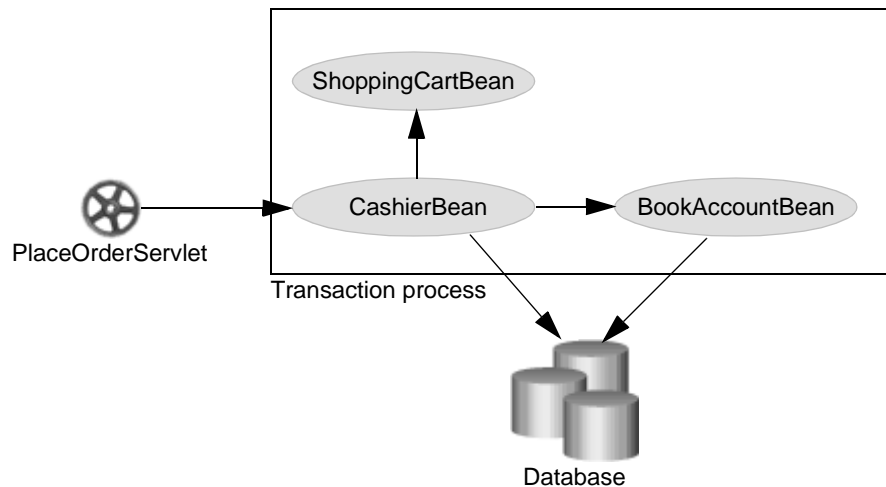
CheckOrdersServlet is invoked from the Bookstore JSP. When invoked by registered users, the servlet displays CheckOrders.jsp, which shows the user’s previous orders, if any. When invoked by an administrator, the servlet displays ShowAllOrders.jsp, which shows orders from all customers.

AdminServlet, called from index.html, displays AdminLoginForm.jsp. This JSP form accepts a bookstore administrator’s ID and password. AdminServlet passes this input to an LDAP server for authentication. If authentication succeeds, the servlet allows viewing of customer information on another JSP, ShowAllOrders.jsp.

LogoutServlet, invoked from the Bookstore JSP, implements the sign-out logic. LogoutServlet invalidates the customer’s HTTP session and creates a new one.

EJB Functionality

The sample application contains three EJBs, and their database interactions are shown in the following figure.



ShoppingCartBean is a stateful session bean. It holds the book items chosen during the shopping session. The bean is stateful because the data it holds is unique to each client and must not be shared. CartServlet relies on ShoppingCartBean to display book information to the appropriate JSP. The CashierBean EJB calls ShoppingCartBean to obtain data needed when an order is placed.

BookAccountBean is an entity bean that controls the inventory of each book. Unlike the other beans, BookAccountBean is involved in transactions only with the database. In other words, this bean does not calculate data for display in JSPs. CashierBean passes to BookAccountBean a set of book IDs and the quantities sold for each. BookAccountBean then connects to the database and performs a global transaction, thereby reducing the book counts as needed.

CashierBean is a stateless session bean. It coordinates with ShoppingCartBean and BookAccountBean to process the customer's order. CashierBean encapsulates business logic, such as applying the necessary shipping fees and taxes. BookAccountBean then connects to the database and updates it using a global transaction.



Summary of Standard APIs

This appendix lists the standard APIs that Netscape Application Server supports as part of the new application programming model.

The following topics are presented in this appendix:

- The Java Servlet API
- The Enterprise JavaBeans API
- The JDBC Core API
- The JDBC Standard Extensions API

Sun Microsystems owns the process of defining, creating, and publishing the API specifications. Even so, many of the methods, classes, and interfaces from these APIs are described throughout this *Programmer's Guide*, particularly in the context of designing applications for NAS. This appendix provides a convenient location for reviewing the standard APIs.

Note This appendix reflects the specifications that NAS supports for its 4.0 release. All supported specifications are accessible from `installdir/nas/docs/index.htm`, where `installdir` is the location in which you installed NAS. For further reading, see “Related Information” in the Preface. Also see the official Java web site at <http://java.sun.com>.

For more information about the Netscape APIs provided specifically for NAS, see Chapter 1, “Summary of the NAS API,” in the *Netscape Application Server Foundation Class Reference (Java)*.

The Java Servlet API

This section lists the classes and interfaces defined in Version 2.1 of the *Java Servlet API Specification*. The Servlet API specifies two packages:

- `javax.servlet`
- `javax.servlet.http`

For the latest documentation and source code, see the servlet 2. 1 specification. All specifications are accessible from `installdir/nas/docs/index.htm`, where `installdir` is the location in which you installed NAS.

The javax.servlet Package

`javax.servlet` contains the following classes and interfaces:

Class	Interface	
GenericServlet	RequestDispatcher	ServletRequest
ServletException	Servlet	ServletResponse
ServletInputStream	ServletConfig	SingleThreadMode
ServletOutputStream	ServletContext	
UnavailableException		

The javax.servlet.http Package

`javax.servlet.http` contains the following classes and interfaces:

Class	Interface
Cookie	HttpServletRequest
HttpServlet	HttpServletResponse
HttpSessionBindingEvent	HttpSession
HttpUtils	HttpSessionBindingListener
	HttpSessionContext

The Enterprise JavaBeans API

This section summarizes the packages defined in Version 1.0 of the *Enterprise JavaBeans Specification*. The Enterprise JavaBeans API specifies two packages:

- `javax.ejb`
- `javax.ejb.deployment`

For the latest documentation and source code, see the EJB 1.0 specification. All specifications are accessible from `install_dir/nas/docs/index.htm`, where `install_dir` is the location in which you installed NAS.

The javax.ejb Package

`javax.ejb` contains the following exceptions and interfaces:

Exception	Interface	
CreateException	EJBContext	EntityContext
DuplicateKeyException	EJBHome	Handle
EJBException	EJBMetaData	SessionBean
FinderException	EJBObject	SessionContext
ObjectNotFoundException	EnterpriseBean	SessionSynchronization
RemoveException	EntityBean	

The javax.ejb.deployment Package

`javax.ejb.deployment` contains the following classes:

Class		
• AccessControlEntry	• ControlDescriptor	• SessionDescriptor
• DeploymentDescriptor	• EntityDescriptor	•

• AccessControlEntry	• ControlDescriptor	• SessionDescriptor
• DeploymentDescriptor	• EntityDescriptor	

Note `javax.ejb.deployment` is deprecated in the EJB 1.1 specification, and thus will be deprecated in future versions of NAS.

The JDBC Core API

This section summarizes the `java.sql` package, which is defined in the *JDBC 2.0 Core API Specification*.

For the latest documentation and source code, see the JDBC 2.0 specification. All specifications are accessible from `install_dir/nas/docs/index.htm`, where `install_dir` is the location in which you installed NAS.

The `java.sql` package contains the following classes and exceptions.

Class	Exception
Date	BatchUpdateException
DriverManager	DataTruncation
DriverPropertyInfo	SQLException
Time	SQLWarning
Timestamp	
Types	

The `java.sql` package contains the following interfaces. Items that are shown in ***bold italics*** are recent additions to the specification; these items are not yet supported in NAS.

Interface		
<i>Array</i>	Driver	SQLData
<i>Blob</i>	PreparedStatement	SQLInput

CallableStatement	<i>Ref</i>	SQLOutput
<i>Clob</i>	ResultSet	Statement
Connection	ResultSetMetaData	<i>Struct</i>
DatabaseMetaData		

The JDBC Standard Extensions API

This section summarizes the `javax.sql` package, which is defined in the *JDBC 2.0 Standard Extensions API Specification*.

For the latest documentation and source code, see the JDBC 2.0 specification. All specifications are accessible from `install_dir/nas/docs/index.htm`, where `install_dir` is the location in which you installed NAS.

The `javax.sql` package contains the following classes and exceptions.

Class	Interface	
ConnectionEvent	ConnectionEventListener	RowSetListener
RowSetEvent	ConnectionPoolDataSource	RowSetMetaData
	DataSource	RowSetReader
	PooledConnection	RowSetWriter
	RowSet	XAConnection
	RowSetInternal	XADataSource

Dynamic Reloading

This appendix describes dynamic loading issues relating to the NAS 4.0 Java class loader.

Some NAS components, namely servlets and JSPs, can be dynamically reloaded into the server while it is running. This allows you to make changes to your application without restarting. You can also change some attributes in application or servlet configuration files, namely those that are not uploaded to the registry and those that do not change the session information.

Additionally, you can instruct NAS to make other classes dynamically reloadable by changing values in the NAS registry.

Note Enterprise JavaBeans (EJBs) are not dynamically reloadable. If you make a change to an EJB, you must restart the server in order to effect the change.

This appendix contains the following sections:

- How Dynamic Reloading Works
- Summary of Related Registry Entries

How Dynamic Reloading Works

By default, you can make changes to servlets or JSPs while NAS is running, and NAS notices the change within 10 seconds and instantiates the new component. EJBs can not be reloaded dynamically.

You can change the duration before NAS notices the change by editing the registry entry `SYSTEM_JAVA\GX_TASKMANAGER_PERIOD`.

You can make individual classes other than servlets and JSPs dynamically reloadable by entering the class name into a semicolon-separated list in the registry entry `SYSTEM_JAVA\GX_VERSIONABLE`. By default, all servlets and JSPs are reloadable.

You can make a set of classes dynamically reloadable depending on whether they implement a certain interface by entering the interface name into the registry entry `SYSTEM_JAVA\GX_VERSIONABLE_IF_IMPLEMENTS`. By the same token, you can make all classes that extend a certain class dynamically reloadable by entering the superclass name into the NAS registry entry `SYSTEM_JAVA\GX_VERSIONABLE_IF_EXTENDS`.

Finally, you can make all classes (other than EJBs) dynamically reloadable by setting the NAS registry entry `GX_ALL_VERSIONABLE` to 1 (the default is 0). This setting is normally used for backward compatibility with older versions of NAS.

If you start NAS from the command line, you can turn off dynamic reloading completely by using the `-d` option.

Summary of Related Registry Entries

The following registry entries control how dynamic reloading works in NAS. Note that these entries are not relevant if the server is started with the `-d` option.

Note NAS does not support dynamic reloading of EJBs.

Registry Entry	Default Value	Description
GX_TASKMANAGER_PERIOD	10	The maximum number of seconds before NAS notices that a class has been changed.
GX_ALL_VERSIONABLE	0	Set to 1 to indicate that all classes (except EJBs) are dynamically reloadable. This is normally used for backward compatibility with older versions of NAS.
GX_VERSIONABLE	null	Semicolon-separated list of classes that are explicitly reloadable.
GX_VERSIONABLE_IF_EXTENDS	<code>javax.servlet.GenericServlet; javax.servlet.HttpServlet</code>	Semicolon-separated list of classes. that if the a user's class extends, then it is considered to be "versionable." A user can add additional classes to the end of the predefined list in a semi
GX_VERSIONABLE_IF_IMPLEMENTES	<code>javax.servlet.Servlet</code>	Semicolon-separated list of interfaces. If a class implements one of these interfaces, it is dynamically reloadable. A user can add additional classes to the end of the predefined list in a semi

For example, the default value for `SYSTEM_JAVA\GX_VERSIONABLE_IF_IMPLEMENTES` includes the classes for generic servlets and HTTP servlets, which makes all servlets and JSPs reloadable. You could add `com.kivasoft.applogic.AppLogic` to this registry entry to make all AppLogics reloadable as well.

Glossary

ACL	Access Control List, a list of users or groups and their specified permissions. See <i>component ACL</i> , <i>general ACL</i> .
administration server	A process in Netscape 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 <i>servlet</i> is a small application that runs on a server.
application	A computer program that performs a task or service for a user. Also see <i>web application</i> .
application event	A named action that you register with the NAS 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	<p>A conceptual division of an application:</p> <ul style="list-style-type: none">• 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 NAS-specific class responsible for completing a well-defined, modular task within a Netscape 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 NAS 4.0.
AppPath	A NAS 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 <i>BasePath/APPS</i> , where <i>BasePath</i> is the base NAS directory. (<i>BasePath</i> is also a NAS variable.)
attribute	Attributes are name-value pairs in a request object that can be set by servlets. Contrast with <i>parameter</i> . More generally, an attribute is a unit of metadata.
authentication	The process of verifying a user-provided username and password.
BasePath	A NAS registry entry that contains the directory where NAS is installed, including the <i>nas</i> subdirectory (other Netscape products can also be installed in <i>BasePath</i>). For instance, if you install into <i>/usr/local/netscape</i> on a UNIX machine, <i>BasePath</i> is <i>/usr/local/netscape/nas</i> . <i>BasePath</i> is a building block for <i>AppPath</i> .
bean property file	A text file containing EJB deployment information. The type of information is defined in <i>javax.ejb.DeploymentDescriptor</i> .
bean-managed transaction	See <i>declarative transaction</i> .
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 Netscape Application Server that runs and manages C++ objects.
cached rowset	A <i>CachedRowSet</i> object permits you to retrieve data from a <i>datasource</i> , then detach from the <i>datasource</i> while you examine and modify the data. A cached rowset keeps track both of the original data retrieved, and any changes made to the data by your application. If the application attempts to update the original <i>datasource</i> , the rowset is reconnected to the <i>datasource</i> , 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 <i>interface</i> .
class file	A file that contains a compiled class, usually with a <code>.class</code> extension. See also <i>class name</i> , <i>classpath</i> . Normally referred to in terms of its location in the filesystem, as in <code>.../com/myDomain/myPackage/myClass.</code>
class loader	A Java component responsible for loading Java classes, according to specific rules.
class name	The name of a class in the Java Virtual Machine. See also <i>class file</i> , <i>classpath</i> .
classpath	The path that identifies a Java class or package, in terms of its derivation from other classes or packages. See also <i>class file</i> , <i>class name</i> . For example, <code>com.myDomain.myPackage.myClass.</code>
client	An entity that invokes a resource.
client contract	A contract that determines the communication rules between a client and the EJB container, establishes a uniform development model for apps that use EJBs, and guarantees greater reuse of beans by standardizing the relationship with the client. See <i>Enterprise JavaBean (EJB)</i> .
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 <i>transaction</i> .
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 <i>Enterprise JavaBean (EJB)</i> .

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 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 NAS and then retrieved programmatically in order to establish connections and interact with the datasource. A datasource definition specifies how to connect to the source of data.
declarative security	Declaring security properties in the component's configuration file and allowing the component's container (i.e., a bean's container or a servlet engine) to manage security implicitly. This type of security requires no programmatic control. Opposite of <i>programmatic security</i> .
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 <i>programmatic transaction</i> .

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 Netscape 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 <i>Enterprise JavaBean (EJB)</i> .
Directory Server	An LDAP server that is bundled with Netscape Application Server. Every instance of Netscape Application Server uses Directory Server to store shared server information, including information about users and groups.
distributable session	A user session that is distributable among all servers in a cluster.
distributed transaction	A single transaction that can apply to multiple heterogeneous databases that may reside on separate servers.
dynamic reloading	Updating and reloading a component without restarting the server. By default, servlet and JavaServer Page (JSP) components can be dynamically reloaded.
e-commerce	Industry buzzword, a term meaning electronic commerce, indicating business done over the Internet.
Enterprise JavaBean (EJB)	A business logic component for applications in a multitiered, distributed architecture. EJBs conform to the Java EJB standard specifications, which defines beans in terms of their expected roles. An EJB encapsulates one or more application tasks or application objects, including data structures and the methods that operate on them. Typically they also take parameters and send back return values. EJBs always work within the context of a container, which serves as a link between the EJBs and the server that hosts them. See also <i>container</i> , <i>session EJB</i> , and <i>entity EJB</i> .
entity EJB	An entity Enterprise JavaBean (EJB) relates to physical data, such as a row in a database. Entity beans are long-lived, because they are tied to persistent data. Entity beans are always transactional and multiuser aware. Also see <i>session EJB</i> .
executive server	Process in Netscape 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 <i>Enterprise JavaBean (EJB)</i> .
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 <code>javax.servlet.GenericServlet</code> . Generic servlets are protocol independent, meaning that they contain no inherent support for HTTP or any other transport protocol. Contrast with <i>HTTP servlet</i> .
global database connection	A database connection available to multiple component. Requires a resource manager.
global transaction	A transaction that is managed and coordinated by a transaction manager and can span multiple databases and processes. The transaction manager typically uses the XA protocol to interact with the database backends. Also see <i>local transaction</i> .
group	A group of users that are related in some way, maintained by a local system administrator. See also <i>user</i> , <i>role</i> .
GUID	128-bit hexadecimal number, guaranteed to be globally unique, used to identify components in a NAS application.
home interface	A mechanism that defines the methods that enable a client to create and remove an Enterprise JavaBean (EJB). See <i>Enterprise JavaBean (EJB)</i> .
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 <code>javax.servlet.HttpServlet</code> . These servlets have built-in support for the HTTP protocol. Contrast with <i>generic servlet</i> .
identity	An instance of <code>java.security.Identity</code> , a security object that contains information about a specific entity, such as a user or a group.
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 <i>instance</i> .
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 <i>class</i> .
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 <i>Enterprise JavaBean (EJB)</i> .
JavaBean	A discrete, reusable Java object.
Java server	Process in Netscape 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. NAS supports JNDI lookups for datasources and Enterprise JavaBean (EJB) components.
kas	See <i>administration server</i> .
kcs	See <i>C++ server</i> .
kjs	See <i>Java server</i> .

kxs	See <i>executive server</i> .
LDAP	Lightweight Directory Access Protocol. LDAP is an open directory access protocol that runs over TCP/IP. It is scalable to a global size and millions of entries. Using Directory Server, a provided LDAP server, you can store all of your enterprise's information in a single, centralized repository of directory information that any application server can access via the network.
load balancing	A technique for distributing the user load evenly among multiple servers in a cluster. Also see <i>sticky load balancing</i> .
local database connection	The transaction context in a local connection is not distributed across processes or across datasources; it is local to the current process and to the current datasource.
local session	A user session that is only visible to one server.
local transaction	A transaction that is native to one database and is restricted within a single process. Local transactions can work against only a single backend. Local transactions are typically demarcated using JDBC APIs. Also see <i>global transaction</i> .
memory cache	A NAS 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.
NAS registry	A collection of application metadata, organized in a tree, that is continually available in active memory or on a readily-accessible Directory Server.
NASRowSet	A RowSet object that incorporates NAS extensions. The <code>NASRowSet</code> class is a subclass of <code>ResultSet</code> .
NTV	A Name-Type-Value format, used in NAS for servlet and application configuration files.
package	A collection of related classes that are literally packaged together in a Java archive (<code>.jar</code>) 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 <i>Enterprise JavaBean (EJB)</i> .
permission	A set of privileges granted or denied to a user or group. See also <i>ACL</i> .
persistent	Refers to the creation and maintenance of a bean throughout the lifetime of the application. In NAS 4.0, beans are responsible for their own persistence, called <i>bean-managed persistence</i> . Opposite of <i>transient</i> .
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 NAS, 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.
process	A sequence of execution in an active program. A process is made up of one or more threads.
programmatic security	Controlling security explicitly in code rather than allowing the component's container (i.e., a bean's container or a servlet engine) to handle it. Opposite of declarative security.
programmatic transaction	Controlling a transaction explicitly in code rather than allowing an Enterprise JavaBean (EJB)'s container to handle it. Opposite of declarative transaction.

property	A single attribute that defines the behavior of an application component.
registration	The process by which NAS gains access to a servlet, Enterprise JavaBean (EJB), and other application resource, so named because it involves placing entries in the NAS registry for each item.
remote interface	Describes how clients can call a Enterprise JavaBean (EJB)'s methods. See <i>Enterprise JavaBean (EJB)</i> .
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 <code>java.sql.ResultSet</code> interface. ResultSets are used to encapsulate a set of rows retrieved from a database or other source of tabular data.
reusable component	A component created so that it can be used in more than one capacity, i.e., by more than one resource or application.
RMI	Remote Method Invocation (RMI), a Java standard set of APIs that enable developers to write remote interfaces that can pass objects to remote processes.
role	A functional grouping of subjects in an application, represented by one or more groups in a deployed environment. See also <i>user</i> , <i>group</i> .
rollback	Cancel a transaction. See <i>transaction</i> .
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 <code>java.sql.ResultSet</code> 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 <code>Servlet</code> class. A servlet is a reusable application that runs on a server. In NAS, 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 <i>servlet engine</i> .
session cookie	A cookie that is returned to the client containing a user session identifier.
session EJB	A session Enterprise JavaBean (EJB) relates to a unit of work, such as a request for data. Session beans are short lived—the lifespan of the client request is the same as the lifespan of the session bean. Session beans can be stateless or stateful, and they can be transaction aware. See <i>stateful session EJB</i> and <i>stateless session EJB</i> . Also see <i>entity EJB</i> .
session timeout	A specified duration after which NAS can invalidate a user session. See <i>user session</i> .
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 NAS feature interface <code>IState2</code> .
stateful session EJB	An Enterprise JavaBean (EJB) that represents a session with a particular client and which automatically maintains state across multiple client-invoked methods.
stateless session EJB	An Enterprise JavaBean (EJB) that represents a stateless service. A stateless session bean is completely transient and encapsulates a temporary piece of business logic needed by a specific client for a limited time span.
sticky cookie	A cookie that is returned to the client to force it to always connect to the same executive server process.

sticky load balancing	A method of load balancing where an initial client request is load balanced, but subsequent requests are directed to the same process as the initial request. Also see <i>load balancing</i> .
stored procedure	A block of statements written in SQL and stored in a database. You can use stored procedures to perform any type of database operation, such as modifying, inserting, or deleting records. The use of stored procedures improves database performance by reducing the amount of information that is sent over a network.
streaming	A technique for managing how data is communicated via HTTP. When results are streamed, the first portion of the data is available for use immediately. When results are not streamed, the whole result must be received before any part of it can be used. Streaming provides a way to allow large amounts of data to be returned in a more efficient way, increasing the perceived performance of the application.
system administrator	The person who is responsible for installing and maintaining Netscape Application Server software and for deploying production NAS 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 <i>local transaction</i> , <i>global transaction</i> .
transaction manager	Object that controls a global transaction, normally using the XA protocol. See <i>global transactions</i> .
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 <i>persistent</i> .
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 <code>http://domain:port/toppings/cheese.html</code> .

URL	Uniform Resource Locator. An address that uniquely identifies an HTML page or other resource. A web browser uses URLs to specify which pages to display. A URL describes a transport protocol (e.g. HTTP, FTP), a domain (e.g. <code>www.my-domain.com</code>), and optionally a URI.
user	A person who uses your application. Programmatically, a user name, password, and set of attributes that enables an application to recognize a client. See also <i>group</i> , <i>role</i> .
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.
validation	A NAS feature that enables validity checking of some types of form input.
versioning	See <i>dynamic reloading</i> .
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 Netscape 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

A

- access control lists 220, 224, 233
- accessing
 - business logic 62
 - databases 140, 155
 - parameters 61
- activating an entity bean 130
- appInfo.ntv file 177
- application events 239
- application model 26, 28, 150
- applications
 - designing user interface 41
 - generic 51
 - identifying requirements 35
 - partitioning 109
- AppName section 179
- AppPath 51
- authenticating a user 228

B

- BasePath 51
- batch updates
 - handling in JDBC 166
- bean, see EJBs
- bean tags 77
- bookstore sample application 259
- business logic layer 30
- business rules 123

C

- CacheCriteria field 238
- CallableStatement 166

- cancel 162
- class definition 117, 128
- classes 33
- class file 57
- client tier 23
- close() 245
- code re-use 54
- concurrency 163
- configuration files 67, 173
- configuring servlets 50
- Connection.isClosed() 162
- control descriptors 197
- cookies 212
- createMailbox() 244
- creating
 - entity beans 132
 - servlets 57
 - session beans 117

D

- data access layer 31
- databases
 - accessing from EJBs 111
 - accessing in servlets via rowsets 158
 - accessing through
 - java.transaction.UserTransaction 155
 - accessing with JDBC 155
 - connection handling with JDBC 162
 - EJBs as the preferred interface to 155
 - JDBC rowset support for 169
 - portability access choices 155
- databases supported by NAS 154
- database transactions 122, 137

- committing 138
- committing in entity beans 122
- distributed 167
- database vendor limitations 151
- data tier 24
- DB2 154
- deactivating an entity bean 130
- declaring an EJB remote interface 118, 134
- deploying servlets 52
- deployment descriptor 192
- design guidelines 42
- destroy() 49, 59
- destroying servlets 49
- development team 37
- DISPLAY tag 80
- distributed transactions 32, 167
- documentation 11
- doGet() 49, 59
- doPost() 49, 59
- dynamic reloading 50, 277

E

- ejbActivate() 130
- EJBContext 221
- ejbCreate() 117, 129, 132
- ejbFindByPrimaryKey() 129
- EJBHome 119, 134
- ejbLoad() 130
- EJBObject 115, 116, 135
- ejbPassivate() 130
- ejbPostCreate() 129
- EJBs 31
 - about 27
 - access control 202
 - accessing databases with through JDBC 155
 - accessing NAS value-added features from 120
 - client contract 104

- component contract 104
- container 102
- database access from 111
- defined 103
- entity beans 107, 110, 126
- granularity 124
- in NAS applications 107
- introduction to 101–112
- jar file contract 105
- partitioning guidelines 108
- planning guidelines 109
- property files 191
- purpose of 102
- remote interface 115, 116
- sample application 268
- session beans 106, 110, 113
- stateful vs. stateless 120
- transaction isolation level in 156
- using JDBC in 155
- using serialization 121, 137
- EJB specification 18
- ejbStore() 130
- email 243
 - receiving 244
 - required servers 246
 - security 243
 - sending 246
- Enterprise JavaBeans, *see* EJBs
- entity beans 31, 107, 110, 126
 - accessing 127
 - accessing NAS 136
 - class definition for 128
 - committing transactions in 138
 - declaring a remote interface 134
 - descriptor 196
 - ejbActivate() 130
 - ejbCreate() 132
 - ejbLoad() 130
 - EJBObject 135
 - ejbPassivate() 130
 - ejbStore() 130
 - granularity 140
 - home interface 134
 - requirements for 128

- events 239
- examples
 - email, getting 245
 - email, sending 247
- exception bean 87
- EXCLUDEIF tag 83
- executeBatch() 167
- extensions, pre-built 31

F

- FindByPrimaryKey() 133
- finder methods 133
- format
 - URLs, in manual 17
- forward() method 94
- FORWARD-ONLY READ-ONLY result set 163

G

- generic applications 51
- generic servlets 48, 53
- getAppLogic() 236
- getArray() 165
- getBlob() 165
- getCallerIdentity() 223, 228, 229
- getClob() 165
- getCreationTime() 214
- getCursorName() 164
- getId() 214
- getLastAccessedTime() 214
- getObject() 165
- getRef() 165
- getRemoteUser() 215
- getRequestedSessionId() 215
- getTypeMap 162
- getValue() 216
- getValueNames() 216

H

- handling requests 49
- home interface 119, 134
- HttpServletRequest 213
- HttpServletRequest2 236
- HTTP servlets 48, 53
- HttpSession 214
- HttpSession2 218

I

- IAppEventMgr 240
- IAppEventObj 240
- ICallerContext 221
- IContext 237
- IMailbox 243
- implementing a remote interface 118
- include() method 93
- INCLUDEIF tag 82
- Informix 154
- init() 49, 58
- InitArgs section 188
- instantiating servlets 49
- interfaces 33
 - how to use 34
- isAuthorized() 224
- isCallerInRole() 224, 229
- IServerContext 221, 237
- isNew() 214
- isRequestedSessionIdFromCookie() 215
- isRequestedSessionIdFromURL() 215
- isRequestedSessionIdValid() 215

J

- jar file 105
- java.transaction.UserTransaction 155
 - managing transactions with 156

Java Database Connectivity, see JDBC

Java Development Kit, see JDK

Java Naming and Directory Interface, see JNDI

JDBC

- 1.0 support in NAS 151

- 2.0 support in NAS 151

- about 27

- application model diagram 150

- batch updates 166

- concurrency support 163

- databases supported by NAS 154

- database support 151

- database vendor limitations 151

- defined 150

- distributed transactions 167

- handling database connections 162

- JNDI support in 171

- managing transactions with 156

- NASRowSet class 169

- restricting databases access with to EJBs 155

- result sets

 - updatable 163

- rowsets 169

- SCROLL-INSENSITIVE READ-ONLY result

 - sets 163

- servlet access via rowsets 158

- SQL-2 support 152

- SQL support 151

- transactions, distributed 167

- updating in batch mode 166

- using in EJBs 155

- using in servlets 155, 157–158

- using rowset with servlets 158

JDK

JNDI

- JDBC support for 171

- using in JDBC 171

JSPs

- about 27, 30, 73

- bean tags 77

- compared to servlets 53

- designing 75

- dynamic reloading 277

- embedded Java code 88

- examples 85

- implicit beans 87

- invoking with a URL 98

- invoking with include() or forward() 99

- script tags 88

JSP specification 18

L

libraries 33

loading bean state information 130

loginSession() 218, 223, 229

looking up remote interfaces

LOOP tag 81

M

messages, email 244

Microsoft SQLServer 154

N

NAS

- as part of multitier environment 22

- databases supported by 154

- documentation 11

NAS registry 51, 174

NASRowSet class 169

Netscape Application Builder 248

Netscape Application Server, see NAS

Netscape Directory Server 220

Netscape Extension Builder 32

NTV format 51, 67, 175

O

ODBC 154

open() 244

Oracle 154

P

- passivating an entity bean 130
- pooling servlets 50
- PreparedStatement 165
- presentation layer vs. logic 29
- property files
 - datasources 208
 - editing 207
 - example 205
- putValue() 216

R

- registry 51, 174
- remote interface 115, 116, 134, 135
 - declaring 118
 - implementing 118
- removeValue() 216
- removing servlets 49
- request bean 87
- request object 49
- resource allocation 50
- response pages 65
- restoring bean state information 130
- result cache 185, 237
- ResultSet 163
- ResultSetMetaData 165
- result sets
 - FORWARD-ONLY READ_ONLY 163
 - SCROLL-INSENSITIVE READ-ONLY 163
 - updatable 163
- retrieve() 244
- retrieveCount() 244
- retrieveReset() 244
- reusability 54
- rowsets 169
 - in servlets 158
 - NASRowSet 169

S

- sample application 24, 259
- script tags 88
- security 61, 212, 220
 - authenticating a user 228
 - cookies 225
 - email 243
 - groups and roles 225
- send() 246
- serialization 137
 - of bean references 121
- server-side includes 92
- server tier 23
- service() 49, 59
- ServletContext 221
- ServletData section 185, 238
- servlet engine 48, 50, 185, 236
- ServletFiles section 180
- ServletRegistryInfo section 183
- ServletRunnerInfo section 185
- servlets
 - about 27, 29, 45
 - accessing databases with through JDBC 155
 - compared to JSPs 53
 - configuration 50
 - creating 57
 - deploying 52
 - designing 52, 54
 - destroying 49
 - directory structure 51
 - dynamic reloading 67, 181, 277
 - execution cycle 47
 - generic vs. HTTP 48, 53
 - initial parameters 188
 - instantiating 49
 - invoking from a servlet 70
 - invoking using a URL 69
 - metadata 185
 - non-standard 46
 - pooling 50
 - registry and ACL information 183

- removing 49
- request handling 49
- sample application 267
- standard vs. nonstandard 54
- using JDBC in 157–158
- using rowsets in 158

servlet specification 18

SessionBean interface 117

session beans 31, 106, 120

- accessing NAS value-added features 120
- creation guidelines 120
- descriptor 194
- granularity 124
- stateful vs. stateless 114, 195
- using 110

SessionInfo section 178

sessions 61, 178, 185, 212

- invalidating 216
- sharing with AppLogics 218
- timeout 194

SessionSynchronization interface 118

setEntityContext() 132

SETFROMREQUEST tag 79

SETONCREATE tag 79

setSessionVisibility() 218

setTransactionIsolationLevel 162

setTypeMap 162

specifications 18

SQL

- support for in JDBC 152

standard APIs 23, 271

Statement class 167

storing bean state information 130

storing data 61

Sybase 154

T

tags

- DISPLAY 80

- EXCLUDEIF 83
- INCLUDEIF 82
- LOOP 81
- SETFROMREQUEST 79
- SETONCREATE 79
- USEBEAN 78

thread safety 63

tier

- client 23
- data 24
- server 23

transactions 122, 137

- committing 138
- committing in entity beans 122
- distributed 32, 167
- isolation level 156

TX_BEAN_MANAGED 145, 156, 199

TX_MANDATORY 145, 199

TX_NOT_SUPPORTED 145, 199

TX_REQUIRED 145, 199

TX_REQUIRES_NEW 145, 199

U

unsetEntityContext() 132

updates, batch mode 166

URLs

- format, in manual 17

USEBEAN tag 78

user interface 41

using JNDI 171

V

value-added NAS features 136