Domino and WebSphere Together
Second Edition

Installation and setup including SSO

Development using servlets, JSPs, EJBs and Domino

Application deployment and security

Søren Peter Nielsen
Mike Bartlett
Eric Ernst
Christian Steege

ibm.com/redbooks
Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix L, “Special notices” on page 523.

Second Edition (June 2001)

This edition applies to Lotus Domino R5.0.6a and IBM WebSphere Application Server Advanced Edition 3.5.3

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. TQH Mail Station P099
2455 South Road
Poughkeepsie, New York 12601-5400

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2000, 2001. All rights reserved.
Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
Contents

Preface ................................................................. xi
The team that wrote this redbook ................................. xi
Comments welcome ................................................. xiii

Part 1. Installation and setup ........................................ 1

Chapter 1. Introduction .............................................. 3
1.1 Domino and WebSphere defined ............................... 3
1.2 Why Domino and WebSphere are so complementary ..... 5
1.3 The information in this book ................................. 7
1.4 Summary ......................................................... 7

Chapter 2. WebSphere and Domino overview ...................... 9
2.1 WebSphere server overview .................................. 9
   2.1.1 Overview of WebSphere server components ......... 10
   2.1.2 WebSphere administrative server .................... 11
   2.1.3 WebSphere application server ......................... 13
   2.1.4 A note about URLs and URIs ......................... 17
2.2 Overview of the elements of WebSphere applications ... 18
   2.2.1 Java servlets .............................................. 19
   2.2.2 JavaServer Pages ........................................ 20
   2.2.3 Enterprise JavaBeans .................................... 22
   2.2.4 What about J2EE and Web services .................. 27
2.3 Domino object model .......................................... 28
   2.3.1 Domino services ......................................... 28
   2.3.2 Domino object hierarchy ............................... 30
2.4 Comparing the development models of Domino and WebSphere ... 32
   2.4.1 Why should you build your Web application with Domino R5? ... 32
   2.4.2 What WebSphere can add to a Domino R5 application .... 33
   2.4.3 Why should you build your Web application with WebSphere? .... 36
   2.4.4 What Domino R5 can add to WebSphere applications .... 36
2.5 Summary ......................................................... 37

Chapter 3. Installation and setup .................................. 39
3.1 Introduction ..................................................... 39
   3.1.1 Domino and WebSphere on the same computer ......... 40
3.2 Prerequisites .................................................... 40
   3.2.1 Platform .................................................. 40
   3.2.2 Product software levels ................................. 41
3.3 Creating a user with administration rights for DB2 and WebSphere ... 41
3.4 Installation of DB2 UDB Release 7.1 Enterprise Edition and FixPak 1 48
10.5.1 Installing the RMI server .................................. 446
10.5.2 Activating the Transfer Funds agent ......................... 447
10.6 Summary .................................................................. 448

Appendix A. Configuration of thin servlet redirector for Domino .... 449
A.1 Defining IIOP port numbers for WebSphere ....................... 449
A.2 Configuring the thin servlet redirector environment ............. 451
A.3 Testing the thin servlet redirector with Domino ................. 455

Appendix B. The ReadNamesRemote Servlet ....................... 459

Appendix C. Using WebSphere advanced LDAP properties ......... 463

Appendix D. Securing the LDAP server from anonymous access .. 471
D.1 Limiting fields an anonymous reader can query ................. 471
D.2 Requiring LDAP clients to authenticate to retrieve directory entries ... 473
  D.2.1 Securing the Domino LDAP server from anonymous access . . 474
  D.2.2 Configuring WebSphere to provide a distinguished name to bind . 476

Appendix E. Installing the IBM WebSphere 3.5 banking example .... 479
E.1 Creating a database for banking account data and connecting to it .... 479
  E.1.1 Create ACCOUNTS database on the database server ........ 479
  E.1.2 Connect to ACCOUNTS database from WebSphere server ... 480
E.2 Deploying the banking example EJBs ............................. 481
  E.2.1 Create a data source for the Account EJB .................... 482
  E.2.2 Create a container for the account example EJBs .......... 482
  E.2.3 Deploy and create the Account EJB ......................... 483
  E.2.4 Deploy and create the Transfer EJB ......................... 485
E.3 Applying security for the banking example ....................... 486
  E.3.1 Create servlets as resources of a Web application ......... 486
  E.3.2 Create an Enterprise Application ............................ 487
  E.3.3 Configure application security ............................... 490
  E.3.4 Configure resource security ................................. 491
  E.3.5 Configuring security permissions ............................ 495
E.4 Testing the banking example application ......................... 498

Appendix F. Using JDBC to access Domino ......................... 501
F.1 Installing the JDBC driver for Domino .......................... 501
F.2 Creating a servlet that uses JDBC to access Domino R5 data .... 502
  F.2.1 The init and destroy methods of the servlet ............... 503
  F.2.2 The doGet method of the servlet ........................... 503
Appendix G. HTML output from viewnav.jsp .......................... 507

Appendix H. JSP custom tags that create scripting variables .... 511
  H.1 Tag handler class .................................................. 512
  H.2 Tag Extra Info class ............................................. 512
  H.3 TLD file .............................................................. 513
  H.4 <redbook:dominoUserName> example .......................... 513

Appendix I. Variations of the People view using JSP custom tags .. 515

Appendix J. Domino Collaboration Objects for Java ................. 519

Appendix K. Using the additional Web material ........................ 521
  K.1 How to get the Web material ................................... 522

Appendix L. Special notices ............................................. 523

Appendix M. Related publications ....................................... 527
  M.1 IBM Redbooks ..................................................... 527
  M.2 IBM Redpapers ................................................... 528
  M.3 IBM Redbooks collections ....................................... 529
  M.4 Other resources .................................................. 529
  M.5 Referenced Web sites ............................................ 529

How to get IBM Redbooks ................................................. 531
  IBM Redbooks fax order form ....................................... 532

Index ............................................................................ 533

IBM Redbooks review ..................................................... 539
Domino and WebSphere Together
Preface

Lotus Domino, IBM WebSphere, and their related products can be used to deliver new value to customers with integrated solutions. In this IBM Redbook we explain how you can use the combined capabilities of Domino and WebSphere to provide a complete and integrated platform for solutions like collaborative commerce.

In the first part of the book we show you how to install and configure Lotus Domino R5.0.6a and IBM WebSphere 3.5.3. This includes setup of Single Sign On (SSO) between the different servers. We describe this for Windows 2000, but you should be able to use our description to guide you through setup on other platforms as well.

In the second part of the book we look at how the WebSphere Java server components—servlets, JavaServer Pages (JSP), and Enterprise JavaBeans (EJB)—can work together with the functionality of Domino in a Web application from a developer’s viewpoint. We use one common sample application to illustrate how the different technologies can work together.

In the last chapter we show how to deploy and secure our sample application. Among the topics we cover in appendices are advanced LDAP properties for WebSphere, securing the LDAP server, and JSP custom Domino tags that support scripting.

All examples are supplied as additional material for download from the IBM Redbooks Web site. The samples includes general ways to handle application login and token expiry, forwarding to JSPs when using a Domino session, and so on.

This redbook enables architects, solution developers, and administrators to understand how Domino and WebSphere integrate from a technical angle.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization Cambridge Center.

Søren Peter Nielsen works for the International Technical Support Organization at Lotus Development, Cambridge, Massachusetts. He manages projects that produce Redbooks about all areas of Lotus products. Before joining the ITSO in 1998, Søren worked as an IT Architect for IBM
Global Services in Denmark, designing solutions for a wide range of industries. Søren is a Certified Lotus Professional at the Principal level in Application Development and System Administration.

**Mike Bartlett** is an IT Architect with IBM Global Services in Toronto, Ontario, Canada. As an IT architect with the Knowledge and Content Management Practice, his main role is the development of strategic e-business solutions for IBM customers, including corporate extranets, messaging solutions, and custom application design. Mike has over 27 years experience in consulting with client organizations in the insurance, distribution, finance, telecommunications, retail, and manufacturing industries.

**Eric Ernst** is a Senior Consultant with United System Solutions, a Cognicase Company in Toronto, Canada. He holds developer certifications in Domino R5, the Java 2 Platform, and WebSphere Application Server. With United System Solutions, a Lotus and IBM Business Partner, he has developed a variety of e-commerce and e-CRM solutions using Domino and WebSphere.

**Christian Steege** is a Systems Architect at Lotus Professional Services in Zürich, Switzerland. He graduated in Information Management at the University of St. Gallen, Switzerland. Since 1990 he has developed Notes/Domino applications, first at the University of St. Gallen and then at different Lotus Business Partners. In 1998 he joined the Lotus Professional Services Team in Zürich. He is leading the implementation of Domino solutions for Lotus customers in Switzerland.

A number of people have provided support and guidance. In particular, we would like to thank **Arthur Fontaine**, Sr. Product Marketing Manager for e-business Applications at Lotus, for direction and contributions.

In addition, we would like to thank the following people:

- Bob Balaban, Looseleaf Software Inc.
- Spencer Brown, Lotus
- Richard Gemperle, Lotus
- Florian Germersdorf, Lotus
- David Morrison, ITSO Cambridge
- Alison Chandler, ITSO Poughkeepsie
- Gail Christensen, ITSO Raleigh
- The ITSO Poughkeepsie editing team
We built on the strong foundation of the first edition of this book, and would like to thank those of the authors that weren’t on the team this time around:

**Michael Laskey** is an IT Architect with IBM Global Services in Tampa, Florida. As a Lead Developer in the e-Commerce Development and Support organization, his main role is the development of service and custom solutions for IBM customers, especially those related to collaboration, messaging, and business-to-business applications. Mike has over 20 years experience with software development, including operating system, client-server, networking and messaging systems.

**Adrian Walmsley** is a Consultant IT Architect in the IBM UK Software Business, based in Hursley, England. He has over 30 years of experience in the IT industry. His most recent interests include Domino/WebSphere integration, and in 1999 he pioneered workshops on this subject for IBM and Lotus personnel in Europe. As an assignee to Poughkeepsie in the mid-1970s, he was involved in the inception of the ITSO redbook program, and since then he has contributed to a number of redbooks.

Also, a special thanks to **Richard Werbin**, Vice President, Prudential Insurance for coming to Cambridge to give us feedback for the first edition.

In addition, thanks to the following people for helping with the first edition:

- Kurt Deitrick, IBM
- Stephen Londergan, Lotus
- Aimee Stone Munsell, IBM
- Nataraj Nagaratnam, IBM
- Patrick Xuereb, Lotus

**Comments welcome**

**Your comments are important to us!**

We want our Redbooks to be as helpful as possible. Please send us your comments about this or other Redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks review” on page 539 to the fax number shown on the form.
- Use the online evaluation form found at [ibm.com/redbooks](http://ibm.com/redbooks)
- Send your comments in an Internet note to [redbook@us.ibm.com](mailto:redbook@us.ibm.com)
Part 1. Installation and setup
Chapter 1. Introduction

Interest in the integration between Lotus Domino and IBM WebSphere application servers remains high. There’s a strong marketplace sense that a tremendous opportunity exists in the combination of these two powerful technologies, but clarity about what the opportunity really means is, well, a bit elusive.

This chapter describes the Domino/WebSphere opportunity by answering two fundamental questions: What? and Why? First we need to understand what Domino and WebSphere are. Knowing where the platforms are similar, and where they are different, is a necessary first step in understanding how to use them together effectively. Armed with this knowledge, it then becomes easy to see why using them in tandem enables you to solve hard problems, and create new opportunities, in a model we call Collaborative Commerce.

Our experience shows that, once you gain these insights about What? and Why?, you will seek to architect solutions based on these complementary application server technologies. This redbook is a primary tool in enabling you to do that.

1.1 Domino and WebSphere defined

Lotus Domino and IBM WebSphere are premier application servers that address different parts of the market. Domino is the leading collaborative application server; it excels at tasks that deal with documents—particularly when documents need to be routed or shared between people—and that must be created swiftly and maintained easily. WebSphere is a definitive Java Web Application Server (WAS), and thus it excels at tasks that require massive scalability, transaction support, and a pure Java development model.

Of course, as premier application servers, both platforms support a wide range of tasks. They enable skilled developers to do amazing things. It’s possible, for example, to create rich applications in Domino that rely almost exclusively on back-end data systems. Similarly, it’s possible in WebSphere to create content-intensive applications based on dynamic documents. Examples exist of these application types—and in many cases the reason for creating them this way is as simple as already having the software or skills in place.

But the real opportunity comes from utilizing the tools in a way that leverages their fundamental strengths. It’s just quicker, easier, and more robust to do it that way. In fairness, when you use most tool combinations, it’s often simpler
to "shoehorn" the application to fit one or the other of the tools exclusively, than to invest energy and time in the integration work required to make them work in harmony. IBM understands this, and has taken the responsibility for wiring the two products in a way that makes integration simple. The Single Sign-On (SSO) capability covered in this redbook is a prime example of that work. Over time you can expect more and better integration points that make it even easier to capture the benefits of the complementary capabilities of Domino and WebSphere, as well as other key members of the IBM software family such as DB2 and Tivoli.

Perhaps the best way to understand the differences, and leverage the best capabilities of each, is to examine the respective design centers of Domino and WebSphere. Table 1 highlights some essential attributes that define, really, what makes Domino be Domino and what makes WebSphere be WebSphere. Remember, we're not talking about total capabilities, which in some respects overlap. We're describing the core capabilities, the “DNA” of the products that makes them particularly well suited for a specific class of application.

Table 1. Attributes of Domino and WebSphere

<table>
<thead>
<tr>
<th>Primary Attribute</th>
<th>Domino</th>
<th>WebSphere</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application type</td>
<td>Collaborative</td>
<td>Integrating or Transactive</td>
</tr>
<tr>
<td>Content type</td>
<td>Document</td>
<td>Data</td>
</tr>
<tr>
<td>Object type</td>
<td>Form, view, database</td>
<td>Servlet, JavaBean, Java Server Page (JSP), Enterprise JavaBean (EJB)</td>
</tr>
<tr>
<td>Architecture</td>
<td>Integrated object model</td>
<td>Java components</td>
</tr>
<tr>
<td>Scalability</td>
<td>Large</td>
<td>Massive</td>
</tr>
<tr>
<td>Skills required</td>
<td>Moderate</td>
<td>High</td>
</tr>
<tr>
<td>Development model(s)</td>
<td>BASIC, COM/COM+, Java</td>
<td>Java (J2EE)</td>
</tr>
<tr>
<td>Clients supported</td>
<td>Notes, Browsers</td>
<td>Browsers</td>
</tr>
<tr>
<td>Protocols supported</td>
<td>NRPC, HTTP, IIOP, SMTP, NNTP, IMAP/POP3, etc.</td>
<td>HTTP, IIOP</td>
</tr>
<tr>
<td>Application tools</td>
<td>Domino Designer</td>
<td>WebSphere Studio, VisualAge for Java</td>
</tr>
</tbody>
</table>

Understanding these fundamental design points answers the basic question: “When do I use what in an application?”
1.2 Why Domino and WebSphere are so complementary

Now that you can differentiate between the fundamental natures of Domino and WebSphere, you need to ask why it is advantageous to combine them in an application. Lotus and IBM have invested in research about the present and emerging forms that applications take on the Web. Our findings map closely to those of major analyst firms, concluding that collaboration remains the key missing element in e-commerce.

While the emergence of online transaction systems and Web self-service have truly changed the way businesses empower their customers, channels, and suppliers, the downside is the creation of a distance between people. As humans, we have been conditioned over the millennia to communicate and interact in a variety of situations. The tools and cues that we take for granted, such as the ability to seek help when we have a simple question, or to signal our uncertainty in a way that solicits extra service, have no equivalent in the digital world.

Lotus has always been about using technology as a collaboration tool. In fact, "Working Together" is the trademarked term that describes the Lotus vision. The ability to include Domino, and other Lotus offerings such as Sametime and QuickPlace, in a business-to-business (B2B) or business-to-consumer (B2C) scenario offers the opportunity to vastly improve the online experience—and generate more revenue.

Consider the elements of a typical e-commerce transaction. (Of course, the inherent assumption is that money changing hands in exchange for goods or services is why most of us do the things we do!) There are really only two things you need to create a transaction: information about what it is you're offering; and a transaction mechanism for things to change hands. Call this the "vending machine" model. Information is the description of what it is you're trying to sell; for a vending machine that's the buttons with the pictures of the soda bottles. The transaction mechanism is the coin or bill slot that takes your money and gives you change, if necessary. A first-generation B2C or B2B site works pretty much this way—an offer and a transaction engine. This basic model works more frequently than you'd expect, but not as often as it could.

The New York Times reports that up to two thirds of shoppers leave e-commerce sites without purchasing the merchandise they have placed in their virtual shopping carts. Giga estimates that if 10% of those sales were salvaged, through better customer service, it would add $1.6 billion US to the total market.
But Forrester found that almost 75% of people in a B2B setting were just too apprehensive to buy anything more than indirect materials (commodities such as office or cleaning supplies) online. Forrester concludes that trust and relationship are the key inhibiting elements in increasing online purchase efficiency. Those are human terms and they need people to exist. Therefore, we believe that there are two elements that can be added to the equation to bring people into the picture, as shown in Figure 1.

The first element to add is knowledge—which is the accrued experience of people, over time, regarding a particular topic. Domino has historically been strong in this area through tools such as knowledge bases and discussion forums. New tools such as the K-Station and the Lotus Discovery Server add even more value. The second piece to add is interaction—or the ability to collaborate with real humans for the purpose of solving a problem. Here, things like Domino workflow (to route exceptions and inquiries), Sametime (for instant meetings and awareness), and QuickPlace (for secure private business centers) deliver the missing pieces. The sum total of all four elements is something we call Collaborative Commerce. It's about tools that inject your most important asset—your people—into your e-business applications.

---

Figure 1. Elements in collaborative commerce
1.3 The information in this book

This book is written to help developers, architects, and administrators to understand how Domino and WebSphere integrate from a technical angle, and how to leverage the products’ combined functions.

In the first part of the book we focus on overview, installation, and setup. This material is relevant for administrators starting to work with a combined Domino/WebSphere installation, developers who need to set up their development and test environments, and architects who need to understand the security and object models of Domino and WebSphere.

In the second part of the book we explore the development of an application that uses Domino functionality for content management and workflow while WebSphere handles transactions and other functionality supplied via servlets, JavaServer Pages (JSPs) and Enterprise JavaBeans (EJBs). This part will be of most interest for developers and architects, though administrators also can benefit from the last chapter where we walk through the deployment of our sample application.

Finally, in the appendices we expand our coverage of some of the chapter topics. For example, there is additional information about how to work with LDAP extended attributes and securing the connection to the LDAP server that is relevant for administrators. For developers we explore source code, the use of JDBC to access Domino, additional abilities of JSPs Taglibs for Domino, and more.

1.4 Summary

In this chapter we have explained why integration of Domino and WebSphere is interesting from a business perspective and discussed how the products complement each other.
Chapter 2. WebSphere and Domino overview

In this chapter we present an overview of the WebSphere application server, then we explain briefly the elements of WebSphere applications.

We continue with an even more brief overview of Domino and the Domino Object Model.

In the last part of this chapter we compare the programming models of Domino and WebSphere and give some examples for applications that use the strengths of both WebSphere and Domino R5.

2.1 WebSphere server overview

The WebSphere application server is available in three different editions. Each of them provides increasingly powerful features to enable you to enhance your Web applications with certain elements.

- The standard edition provides you with an open, standards-based, Web server deployment platform. It enables you to use Java servlets and JavaServer Pages (JSPs) and provides connections to back-end database systems.
  
  WebSphere standard edition is included with Domino R5 Advanced and Enterprise server.

- The advanced edition adds support for Enterprise JavaBeans (EJBs) and provides better support for multi-server environments.

- Along with the above features, the enterprise edition includes the IBM Component Broker, which adds support for CORBA and the IBM TxSeries that provides a transactional application environment.

The WebSphere application server is part of the foundation of the IBM platform for e-business. In addition, IBM offers a range of foundation service products in areas like performance optimization, site analysis, security, and so on (for example, the WebSphere Edge Server). However, we primarily concentrate on the WebSphere application server and Domino in this redbook.

In this chapter we provide a high-level overview of the components of WebSphere necessary to understand the examples later in this book. We describe the WebSphere administrative server and WebSphere administrative domains in the following section, then we discuss WebSphere
application components such as servlets, JSPs, and EJBs in 2.2, “Overview of the elements of WebSphere applications” on page 18.

2.1.1 Overview of WebSphere server components

The WebSphere administrative server manages WebSphere application servers and applications running in them. It also provides a mechanism for operator management of the environment to start and stop application servers, monitor their activities, manage security, and manage the WebSphere environment by, for example, adding applications. The administrative server also provides services for naming, transaction monitoring, and security implementation for applications running in WebSphere application servers.

We illustrate the high-level view of the WebSphere administrative model in Figure 2.

![Overview of WebSphere Administrative Model](image)

**Figure 2. WebSphere administrative model**

WebSphere administration tracks the contents and activities of a WebSphere administrative *domain* by storing information in an administrative database.
The WebSphere installation process creates a new administrative database with a default name of WAS. Using this database will create a standalone WebSphere server in an isolated domain.

It is possible to specify a different name during installation if desired. In addition, it is possible to attach the new WebSphere server to another WebSphere administrative database after installation to replace the installation database connection. Such replacements would be necessary to group multiple computers running WebSphere into a single WebSphere domain. This is necessary if you wish to implement Single Sign On between multiple WebSphere servers.

A WebSphere administrative domain is the set of all computers running WebSphere whose application servers share a single WebSphere administrative database. In the diagram (Figure 2 on page 10) we show two physical computers (illustrated with the host names we used in our testing), each with WebSphere installed and using the same DB2 administrative database. For our testing, the DB2 repository was physically located on one of the servers. However, in practice the DB2 repository would typically be located on a standalone database server so that it could be tuned for maximum database performance, be hardened for availability and, often, isolated from the application environment (for example, by a firewall) to permit greater security.

Each WebSphere server as shown has within it both a (single) administrative server (process) and one or more application servers (processes).

<table>
<thead>
<tr>
<th>“Servers” and “Processes”</th>
</tr>
</thead>
</table>
| When we describe the administration ‘server’ and application ‘servers’ we really mean multiple processes running separate Java Virtual Machines (JVMs) on a physical computer or server. This usage of ‘server’ is so general that we employ it here.

Where it is not clear from the context, we will use the term ‘computer’ rather than ‘server’ to describe a physical machine and continue to use the term ‘server’ to describe processes running within a physical computer.|

### 2.1.2 WebSphere administrative server

The administrative server has multiple components to provide different services. Some of these are:
• Bootstrap service to initialize the WebSphere environment and support console interfaces.
• Naming services to create a Java Naming and Directory Interface (JNDI) name space. This service also services remote naming requests via a Location Service Daemon (LSD) listening on port 9000. The two components of this service (JNDI name service and LSD) are implemented as Enterprise JavaBeans (described later in this chapter).
• A security service that provides authentication and authorization (access control).
• Transaction monitoring services.
• A “nanny” process that monitors the administrative server and keeps it “alive” by restarting it if it stops unexpectedly.

We show a schematic chart of these services in Figure 3 on page 13. Note that we show the administrative console running on a separate computer. It can also run on the same physical computer as the WebSphere administrative server. We typically used this configuration for our testing.

It is also possible to configure the WebSphere administrative server using a command line or XML interface. This would be convenient in a production environment to automate certain tasks, but we did not do this in our testing environment.
Note that it is not absolutely necessary to run the WebSphere console; WebSphere V3.5 can be controlled and configured from a command line interface and configured using an XML interface or from a browser. We did not exploit these interfaces during our testing since we needed the interactive features of the Java-based console.

### 2.1.3 WebSphere application server

The WebSphere application server runs as a separate process with a Java virtual machine (JVM) for hosting WebSphere application components. It is possible to run more than one application server on a single computer.
running a WebSphere administrative server. The application server has two types of components:

1. An EJB container for running Enterprise JavaBeans
2. A servlet engine for running Web applications, which in turn consist of servlets and JavaServer pages

We illustrate these relationships in Figure 4.

![Figure 4. WebSphere Application Server](image)

These relationships are mirrored in the WebSphere administrative console. For example, Figure 5 on page 15 shows a physical server (Thor-2000) with a default (application) server containing a (default) EJB container and a servlet engine. It also contains a JDBC Driver for accessing databases.
As mentioned previously, it is possible to create multiple application servers. This allows you to use different Java virtual machines for different kinds of applications, or in case you use the same JVM you can still optimize it for different applications through its initialization parameters when it is loaded as part of the different application servers. For each application server, one can create multiple EJB containers and/or servlet engines. For our testing we found it convenient to create an EJB container to quickly locate EJBs that are related. We found that the default servlet engine created during installation was adequate.

If the nodes on the console tree in Figure 5 are expanded, one can see the contained components. As shown in Figure 6 on page 16, expanding the EJB container displays the EJBs defined to it. Expanding the servlet engine node displays Web applications.

A Web application is comprised of one or more related servlets, JSPs, and Web pages that can be managed as a unit. For example, you can start and stop the Web application in a single action. The files in a Web application are typically related in the sense that they work together to perform a business logic function.

Each Web application has a classpath. This specifies where to find the servlets that belong to the application. A Web application also has a document root, which is where JSPs and HTML pages are placed, and a webpath, which is used to invoke the Web application from a browser.

To see the servlets and JSPs in a Web application, you need to expand the node by clicking on the “+” sign beside it. We expand the default_app to illustrate.
The console tree will expand, as shown in Figure 7 on page 17, to show the servlets (there are no JSPs in this example) contained within the Web application “default_app.” Note that this view is the WebSphere view of the components. When you invoke components such as servlets they are mapped to a virtual host; this is a mapping device to allow translation of URLs into the WebSphere namespace. Although we do not illustrate any virtual hosts here, the mapping is visible by scrolling the console tree to the bottom and expanding the virtual host of interest. A virtual host is a configuration enabling a single host machine to resemble multiple host machines. You can use this to host different Web sites on the same physical machine. Resources associated with one virtual host cannot share data with resources associated with another virtual host, even if the virtual hosts share the same physical machine. Each virtual host has a logical name and a list of one or more DNS aliases by which it is known. In 3.5.1.1, “Adding aliases to the WebSphere default_host” on page 62 we discuss the mapping of DNS names to virtual hosts. WebSphere provides a default host at initial configuration, which was all we needed for our application.
2.1.4 A note about URLs and URIs

In this book we use the terms URL and URI somewhat interchangeably. URL is an abbreviation of *Uniform Resource Locator* and URI is an abbreviation of *Uniform Resource Identifier*. The Web site [www.w3.org](http://www.w3.org) says: *The URI syntax represents the generic set of all names/addresses that are short strings that refer to resources*. Uniform Resource Locators (URLs) are subsets of URIs that contain sufficient information about which access algorithm to use when accessing the resource referred to. The important thing for us is that URIs allow Web services to be defined in a way that they are not bound to a specific server.

For example, a fully specified URL has the form:

http://odin.lotus.com//webapp/examples/showCfg

However, a URI may be fragmentary, like:

/webapp/examples/showCfg

This means that Web pages and resources can be hosted on different machines and still be identified by the URI without specifying the machine.
name (as we would have to do with the URL). Thus we should use URI when we refer to other services and resources in our code.

This completes our high-level overview of the WebSphere application server components. For more detail, see the product documentation shipped with WebSphere V3.5 (especially the Infocenter) or the redbook *WebSphere V3.5 Handbook* SG24-6161, Chapter 3 *WebSphere Components*. We based our discussion on this reference.

### 2.2 Overview of the elements of WebSphere applications

In this section we introduce you to the following common components in a WebSphere transactional application:

- Servlets
- JavaServer Pages
- Enterprise JavaBeans

In Figure 8 you can see how the server-side components of transactional WebSphere applications play together.

---

*Figure 8. Use of servlets, EJBs and JSPs in a WebSphere transactional application*
Since server-side applications are typically 3-tiered, the components are spread across the tiers as shown in the figure. The components from left to right are as follows:

- **Tier 1**: This tier is the user's view. From a system perspective, it is the client view. The user contacts a server-side application over the Web through a Web browser (containing an HTML file served by an HTTP server, in our case Domino). The HTML file is linked to a servlet in tier 2 at the server. The user sees information returned from the server in an HTML file generated from a JSP file.

- **Tier 2**: This tier is where the business logic resides at the server. The application's processing takes place here, typically at a high-end server. The EJB containing the non-visual functions resides in a container that in turn resides in a Web application server. In our case, the server is WebSphere application server. The container ensures persistence for those EJBs that are of type entity beans; that is, like a unit of work in a database transaction, containers ensure that a function is either completed or rolled back in order to maintain application integrity.

- **Tier 3**: This tier provides the services that the server-side application accesses. In the figure the services are in a database system, but they could include many other resources, such as CICS transactions, MQSeries messages, SAP data, and IMS transactions. These services are collectively referred to as the Enterprise Information System. Since there can be many services, a complex server-side application can have many tiers.

Another way of imagining the three tiers is to see them as layers. The first layer contains the user interface. The second layer contains the application programs. The third layer contains the services available to the application programs. The layers are loosely coupled, meaning they have little dependency on each other.

**Note**: When you add Domino functionality to your solution you may chose to use Domino forms for both input and presentation of results on the client side.

We now go a bit more into detail with Java servlets, JavaServer Pages and Enterprise JavaBeans.

### 2.2.1 Java servlets

Servlets are small Java programs that run on the Web application server. They are portable across platforms and across different Web servers. They usually interact with the servlet engine running on the Web application server
through HTTP requests and responses, which are encapsulated as objects in the servlet.

A servlet is loaded by the Web application server, and remains loaded across client requests. This means that the servlet can maintain system resources, like a database connection, between requests.

Servlets extend the javax.servlet.http.HttpServlet class. Usually the actions the servlet performs are implemented in one or more of the following methods:

- `init()` - This method is called when the servlet is loaded. It is not called again if the servlet URL is called for the second or subsequent time.

- `doGet(HttpServletRequest req, HttpServletResponse resp)` is called if the servlet is invoked via an HTTP GET request. This method is called when a servlet is called by entering its URL in a browser.

- `doPost(HttpServletRequest req, HttpServletResponse resp)` is called if the servlet is invoked via an HTTP POST request. This request usually is sent using a form. You can call a servlet from a Domino R5 form.

The `HttpServletRequest` parameter contains attributes of the request for the servlet. You can use the `HttpServletResponse` to generate a response the servlet engine is sending to the requester after the servlet has performed its tasks.

The documentation for the whole Java servlet API 2.2 that WebSphere uses is available on:

http://java.sun.com/products/servlet/2.2/javadoc/index.html

### 2.2.2 JavaServer Pages

JavaServer Pages (JSPs) are an easy way to combine Java code with HTML, which means data access programs with layout. In Web applications the response sent to the client is often a combination of static page design and dynamically generated data. In this situation, it is much easier to work with JSPs than with servlets that use the `HttpServletResponse` class to construct Web pages.

JSPs are HTML files containing additional tags. WebSphere V3.5 supports versions 0.91, 1.0, and 1.1 of the JSP specification and adds some specific
tags. You can place Java code between some of the JSP tags. The most important tags that allow Java code are:

- Import statements for Java packages. The packages are available on the whole page. Example:

  ```html
  < %@language="java" import="java.sql.*" %>
  ```

- Declaration: declares a variable or method. Example:

  ```html
  <! int iStatus = 0; %>
  ```

- Expression: contains one expression. Example:

  ```html
  <%= request.getParameter("message") %>
  ```

- Scriptlet: contains a code fragment, that is, a short Java program. Example:

  ```html
  <% for (int iCounter = 0; iCounter < vejbAccounts.size(); iCounter ++) {
      AccountDataBean ejbAccount =
          (AccountDataBean) vejbAccounts.elementAt(iCounter);
      int iAccountID = ejbAccount.getID();
  }%>
  ```

In addition, you can include HTML pages, and refer to servlets and EJBs from a JSP. You can find a description of all standards tags at:


If you are using the JSP 1.1 API you also can create your own custom tags. This allows removing logic that would have had to be coded in scriptlets (Java code embedded in the page) within the JSP to external components. This further separates presentation from code so that graphics designers can concentrate on layout and presentation, and application designers can focus on code to retrieve and manipulate business objects. We will discuss the custom tags in 8.4, “Custom tags” on page 295 and the creation of the custom tags for our example application in Chapter 8.6, “Navigation tree JSP in our sample” on page 340 and 8.7, “Banking example: Top frame JSP” on page 358.

The first time a new or modified JSP is called, WebSphere generates a temporary servlet. This servlet is called every time the browser calls the JSP. This is illustrated in Figure 9 on page 22.
In some ways, JSPs are similar to the Active Server Pages (ASP) technology from Microsoft. The difference is that the scripting language is Java in JSPs while it is Visual Basic in ASPs. However, having access to the full strength of the Java language using Java Beans, scriptlets or Tag libraries makes JSPs much more powerful than Active Server Pages.

It is also worth noting that JSPs in many cases are used to display dynamic result pages, as shown in Figure 8 on page 18. If you have an application that includes Domino and WebSphere elements, there may be situations where it actually makes more sense to use Domino forms to display the dynamic results being returned from servlets.

### 2.2.3 Enterprise JavaBeans

*Enterprise JavaBean* is Sun's trademarked term for their EJB architecture (or "component model"). When writing to the EJB specification you are developing enterprise beans (or, if you prefer, EJBs).

Enterprise JavaBeans are designed to be installed on a server, and accessed remotely by a client. The EJB framework provides a standard for server-side components with transactional characteristics.

An EJB client program can be any program that can communicate via the Java protocol Remote Method Invocation (RMI) or via the Internet Inter-ORB Protocol (IIOP). RMI is only possible if the client program is written in Java.

In the following we will explain the most important pieces in the EJB architecture and in part 2 of the book we will describe how we developed an EJB to access Domino. If you want to explore EJB development in detail refer
The EJB framework specifies clearly the responsibilities of the EJB developer and the EJB container provider. The intent is that the “plumbing” required to implement transactions or database access can be implemented by the EJB container. The EJB developer specifies the required transactional and security characteristics of an EJB in a deployment descriptor (this is sometimes referred to as declarative programming).

For an example of how to deploy EJBs in an EJB container in WebSphere, refer to Appendix E.2, “Deploying the banking example EJBs” on page 481.

There are two types of Enterprise JavaBeans:
- Session
- Entity

### 2.2.3.1 Session beans
A typical session bean has the following characteristics. It:
- Executes on behalf of a single client.
- Can be transactional.
- Can update data in an underlying database.
- Is relatively short-lived.
- Is destroyed when the EJB server is stopped. The client has to establish a new session bean to continue computation.
- Does not represent persistent data that should be stored in a database.
- Provides a scalable runtime environment to execute a large number of session beans concurrently.

For example, the task associated with transferring funds between two bank accounts can be encapsulated in a session bean.

### 2.2.3.2 Entity beans
A typical entity bean has the following characteristics. It:
- Represents data in a database.
- Can be transactional.
- Shares access from multiple users.
- Can be long-lived (lives as long as the data in the database).
• Survives restarts of the EJB server. A restart is transparent to the client.
• Provides a scalable runtime environment for a large number of concurrently active entity objects.

Typically, an entity bean is used for information that has to survive system restarts; in contrast to session beans, where the data is transient and does not survive when the client's browser is closed. For example, the information about a bank account can be encapsulated in an entity bean.

**Bean- or container-managed persistence**

An important design choice when implementing entity beans is whether to use **bean-managed persistence** (BMP), in which case you must code the JDBC logic, or **container-managed persistence** (CMP), where the database access logic is handled by the EJB container.

The business logic of a Web application often accesses data in a database. EJB entity beans are a convenient way to wrap the relational database layer in an object layer, hiding the complexity of database access. Because a single business task may involve accessing several tables in a database, modeling rows in those tables with entity beans makes it easier for your application logic to manipulate the data.

Another advantage of entity beans with container-managed persistence is that they provide a standard-based access to database systems. You can change the EJB container, the database system, or the database without changing the code of the EJB. Only the deployment descriptor of the bean must be changed.

If you are using a database system that is not supported by your Web application server, you write entity beans with bean-managed persistence. For example, WebSphere does not support CMP using Domino databases as the persistent store. If you want to store entity beans in Domino R5, you have to use BMP. For entity beans that must implement their persistence themselves, the code is more complicated than that for entity beans with CMP. In addition, the code for entity beans with bean-managed persistence often uses special features of the database system. The database system cannot be changed without changing the EJB code.

**2.2.3.3 EJB architecture**

We will go one level deeper in our discussion of EJBs to give you a better understanding of our examples involving EJBs. Figure 10 on page 25 shows a diagram of client-to-EJB interaction in more detail.
In order for the management of EJBs to be handled by the server properly, a client must access EJBs only through a proxy provided by the EJB container. This allows the container to control persistence, security, caching, and connection management with no knowledge by the client that all of these functions are occurring with no code in the EJB to control them. In order to facilitate this, all client access to EJBs is done by means of instances of the EJB home and the EJB Object interfaces, which are created by the developer during development. (The EJB Object interface is sometimes also called the remote interface.) This allows the server to perform management tasks under the covers by mapping the calls to these interfaces to appropriate calls to the EJB itself, and also by calling infrastructure methods on the EJB to control transactions and storage to databases.

The EJB home interface instance is responsible for allowing clients to find and create EJBs. For entity beans, the home interface includes methods for finding single beans or groups of beans based on certain criteria, including at least one method that allows the location of a bean in a database using a primary key class. For both entity and session beans the home interface includes methods to create new instances of an EJB inside the container and return a reference to an EJB Object interface instance for the bean. Note that there is one EJB home interface instance per class of EJB in a container, but there may be many EJB Object interface instances, depending upon how many actual instances of the EJB class are present.
The EJB Object interface is responsible for providing access to the operations of an EJB. Each call to an EJB Object interface instance is mapped to a corresponding call to a bean instance by the container, subject to security considerations. Because of the separation from the actual bean, the container is free to release resources used by the bean (such as database connections or even the bean instance itself) to other uses, and restore the EJB instance when a call is made to it by a client.

2.2.3.4 Steps in using an EJB

Figure 11 shows the steps involved in a client accessing an EJB.

Figure 11. Steps used to connect to an EJB from a client program

The sequence of actions that occur when a client program wants to use an EJB are as follows:

1. The client requests from the naming service (provided as one of the components of WebSphere) a reference to the EJB home interface of a particular class of EJBs.

2. The naming service replies with the location of the Home interface instance for the EJB class in the container in which the EJB is deployed.

3. The client performs either a create (for a new bean instance) or a find (for an existing entity bean instance) on the EJB home interface instance.

4. The EJB home interface instance locates or creates the EJB instance and places it in the container, and creates the EJB Object interface instance.
5. The EJB home interface instance replies to the client with a reference to the EJB Object instance.
6. The client calls methods on the EJB Object interface instance to access business logic on the EJB.
7. The EJB Object interface instance calls the corresponding methods on the EJB while the container manages the resources needed to accomplish this task.

The latest EJB specification is 1.1 and version 2.0 is in *Proposed Final Draft*. The most significant changes from EJB 1.0 to 1.1 are the use of XML-based deployment descriptors and the need for vendors to implement entity bean support to claim EJB compliance. WebSphere Advanced (and Enterprise) Server 3.5 that we worked with in preparing this book support the EJB 1.1 specification.

### 2.2.4 What about J2EE and Web services

While reading this book you will notice that we mention the terms *J2EE* and *Web services*. We very briefly explain what these terms mean. All the material in this redbook is relevant to J2EE and Web services, but to explore these topics in full is beyond the scope of this book.

*Java 2 Platform, Enterprise Edition (J2EE)* is a brand created by Sun. J2EE is an umbrella for a set of coordinated specifications and practices that together enable solutions for developing, deploying, and managing multi-tier server-centric applications. EJB technology is the basis of J2EE. Some of the technologies under the J2EE umbrella are:

- Enterprise JavaBeans (EJBs)
- JavaServer Pages (JSPs)
- Java servlets
- Java Naming and Directory Interface (JNDI)
- Java Transaction API (JTA)
- CORBA
- JDBC data access API

#### 2.2.4.1 Web services

Web services is a standards-based approach to integrating applications running across distributed servers that are connected via an intranet, extranet, or the Internet.
Open standards and technologies necessary to develop, publish, and deploy Web services applications includes:

- Universal Description Discovery and Integration (UDDI)
- Simple Object Access Protocol (SOAP)
- Java 2 Enterprise Edition (J2EE)
- Web Services Description Language (WSDL)
- Extended Markup Language (XML)

The latest WebSphere application server product from IBM is *WebSphere Technology for Developers*, which is J2EE 1.2 certified, and includes native support for Web services technology.

To learn more about WebSphere Technology for Developers, see:


---

### 2.3 Domino object model

The Domino object model gives you as a developer access to a wide range of services—like object store, directory, security, replication, messaging, workflow, automation through agents, and more—in a consistent way. Through more than 30 objects, with over 600 methods and properties, you can use all those services from Java and other languages that can use Common Object Request Broker Architecture/Internet Inter-ORB Protocol (CORBA/IIOP) or Microsoft’s COM model.

We will now give an overview of the services that can be accessed through the Domino objects and discuss the hierarchy in the Domino object model.

#### 2.3.1 Domino services

Domino services are offered by Domino servers, of which there are three types:

- Domino Mail Server
- Domino Application Server
- Domino Enterprise Server

You can utilize all three server types from Java programs.

The most important services you can access through the Domino Object Model are described in the rest of this section.
2.3.1.1 Object store
Documents in a Domino database can contain any number of objects and data types, including text, rich text, numerical data, structured data, images, graphics, sound, video, file attachments, embedded objects, and Java and ActiveX applets. A built-in full text search engine makes it easy to index and search documents. The object store also lets your Domino applications dynamically present information based on variables such as user identity, user preferences, user input, and time.

2.3.1.2 Directory
A single directory manages all resource directory information for server and network configuration, application management, and security. Domino includes user account synchronization between NT and Domino, and it is Light Weight Directory Access Protocol (LDAP)-compliant which we will utilize when we set up security for WebSphere. The directory is the foundation for easily managing and securing your Internet and intranet applications.

2.3.1.3 Security
The Domino security model provides user authentication, digital signatures, flexible access control, and encryption. Domino security enables you to extend your intranet applications to customers and business partners.

2.3.1.4 Replication
Bi-directional replication automatically distributes and synchronizes information and applications across geographically dispersed sites. Replication makes your business applications available to users around your company or around the world, regardless of time or location.

2.3.1.5 Messaging
An advanced client/server messaging system with built-in calendaring and scheduling enables individuals and groups to send and share information easily. Message transfer agents (MTAs) seamlessly extend the system to Simple Mail Transfer Protocol (SMTP)/Multipurpose Internet Mail Extension (MIME), X.400, and cc:Mail™ messaging environments. The Domino messaging service provides a single server supporting a variety of mail clients: Post Office Protocol V3 (POP3), Internet Message Access Protocol V4 (IMAP4), Message Application Programming Interface (MAPI), Lotus Notes clients and Lotus iNotes Web Access.

2.3.1.6 Workflow
A workflow engine distributes, routes, and tracks documents according to a process defined in your applications. Workflow enables you to coordinate and
streamline ad hoc business activities across an organization, and with customers, partners, and suppliers.

If you need more structured workflow you can get the product Domino Workflow that leverages all of Dominos functionality and adds tools for structured people oriented workflow.

2.3.1.7 Automation by agents
Agents enable you to automate frequently performed processes, eliminating tedious administration tasks and speeding up your business applications. Agents can be triggered by time or events in a business application. Agents can be run on Domino servers or Lotus Notes clients.

2.3.2 Domino object hierarchy
If you program applications in Domino’s development client Domino Designer you can work directly with objects that the user has open in the Notes client or a Web browser (database, view, document and so on). You access this functionality through front-end and back-end classes.

Front-end classes require a user interface to access Domino functionality. Back-end classes, in contrast, do not require a user interface to achieve the same access. When you access Domino from Java you work with the Back-end classes.

There is a hierarchical relationship for Domino objects. Higher hierarchical objects contain the lower ones. The figure below is an example of the hierarchical relationship between a few of the Domino objects.
Figure 12. Example of the Domino object hierarchy

Each object has defined members, properties and methods. Using these members, you can access other objects. The relationship of containment and access means that the higher object has the property or the method to access the lower one.

For example, you can see all the views when you open the database. This means that the opened database(object) includes the views(object). Furthermore, you can see the documents when you select one of the views. This means that your selected view(object) contains the documents(object). This hierarchy is important when using Domino objects. The Session is the top level object in the Domino Object Model. You can work your way to any Domino object if you start from the Session object.

In your Java code you import the Domino classes like this:

```
import lotus.domino.*;
```

These classes resides in Notes.jar (for local access, if Domino and WebSphere are on the same machine) and NCSOW.jar for remote access from WebSphere. One of these files must be in your classpath.

You create a Session object like this (except in agents where the environment supplies the session):

```
sesCurrent = NotesFactory.createSession (sServerName, sLtpaToken);
```
There are many different version of the `createSession` method that takes different parameters. The enablement of Single Sign On (SSO) between Domino and WebSphere added a few more to allow passing the SSO token as a parameters. Once you have a session object you are ready to work with the Domino objects. You can read more about the Domino objects, methods and properties is in the documentation coming with the *Domino Toolkit for Java/CORBA*. You can download the toolkit from http://www.lotus.com/developer

For the latest information about any updates to the `NotesFactory.createSession` methods or other Domino Java APIs refer to the release notes for the version of Domino that you are using.

2.3.2.1 Domino Collaboration Objects for Java

One more thing worth noting in connection with the Domino Object model is that the upcoming version of the Domino Toolkit for Java/CORBA for Domino R5.0.8 is planned to include new Domino Collaboration Objects for Java.

The Domino Collaboration Objects are high-level abstractions of Lotus Domino services. Each Domino Collaboration Object represents a Domino service, such as login service, mail service, or calendar-entry service. The goal is to make it easier for Web developers to tap into Domino messaging and calendaring by simplifying the necessary back-end classes into a few component JavaBeans. This enables developers who are not familiar with Domino to rapidly integrate Domino services into their Java applications.

See Appendix J, “Domino Collaboration Objects for Java” on page 519 for more information.

2.4 Comparing the development models of Domino and WebSphere

IBM WebSphere and Lotus Domino R5 are positioned as Web application servers because you can use both products to create applications that use Web browsers as their clients and run in the Internet or in company intranets.

However, there are major differences in the programming models of Domino and WebSphere that makes each of them more suitable for different kinds of applications.

2.4.1 Why should you build your Web application with Domino R5?

Domino R5 is one of the most successful Web application servers on the market. Some of the reasons for this popularity are identified in this section.
2.4.1.1 Ease of use
With Domino you get all you need for building your Web application in one easy-to-install box. You get a document database, an application server that supports all of the major internet standards, a directory server including an integrated security model, a mail server and a Web server.

With Domino designer you get an easy-to-use integrated development environment, together with many application templates that you can use directly or to base your development on.

2.4.1.2 Development cost and time to market
Domino has a very simple development model, which enables you to build document-based applications very quickly. Since you also can reuse components of any other Domino application, the development effort is reduced.

On the other hand, Domino contains support for very powerful development languages (LotusScript, Java, and C/C++), which enables you to address complex business problems. Of course, this does need more development time than designing forms and views using the document model; however, because you only use procedural code where you need to, the overall development cycle should be much faster than in most other development models.

2.4.1.3 Flexibility
Since Domino R5 applications store data separately from the application design, you can change the application code at any time and update the existing documents using an agent. This possibility gives a Domino developer great flexibility and enables quick adjustment to quickly changing targets during the development process.

2.4.1.4 Replication of data and application logic
The ability to replicate data and application logic between servers, or between server and client, is still unique in the software market. Using Domino off-line services (DOLS), all Domino Web applications can be executed locally. Of course this only applies to applications (or parts of them) that do not connect to other server-based resources or data.

2.4.2 What WebSphere can add to a Domino R5 application
Domino R5 is a unique development platform that can solve many business problems without using external resources. However, this development platform has limits. WebSphere can help a Domino R5 application to overcome these limits. Here we will show how useful the different elements of
WebSphere applications can be to enhance Domino applications. We also will demonstrate this later in Part 2, “The Redbook Banking example application” on page 181.

2.4.2.1 Servlets

To a Domino Web programmer, servlets in many cases are preferable to Web agents. This is because servlets, once loaded, stay in memory, while Domino Web agents have to be loaded and unloaded for every invocation. This makes applications that use servlets instead of agents more scalable. From a Domino perspective you can call servlets “agents on steroids.”

In addition, you are able to store values in servlet variables or session objects. Values that are stored in servlet variables are shared among all users of the servlet, whereas values in session objects are stored per user by the servlet engine. This reduces the number of read and write operations you need to perform in the database.

In Domino 5.0.5 or later the Domino Java API enables you to connect to Domino databases with the rights of the servlet user. This means that using servlets in place of agents is possible in most cases, now without compromising the security of the underlying database. To use this new feature you have to enable Single Sign On. We explain how to do this in Chapter 4, “WebSphere - Domino security and single sign-on” on page 109.

Domino R5 also has its own servlet engine, so you do not absolutely need WebSphere to include servlets in a Domino-based application. However, the WebSphere servlet engine does have some advantages over the Domino servlet engine, specifically:

• Flexibility

  The servlet management of WebSphere is much more flexible than that of Domino. You can group your servlets into different Web applications and store them in different directories. This directory structure is totally independent from the URL the user enters to call a servlet. In Domino you only have one servlet directory and the user must enter a URL that contains the file system directory to call a servlet.

• Security

  A second important advantage of the WebSphere servlet engine is security. In WebSphere you can specify a group of allowed users for every servlet action, whereas Domino does not have the ability to secure servlets. Of course, if you are connecting to a Domino database in your servlet, an unauthorized user should not be able to perform any task in the
database. But this must be handled in the servlet code if you use the
Domino servlet engine.

- Scalability

The WebSphere servlet engine is more scalable and is always kept close
to the most current level of the Java servlet API. The Domino R5.0.6a
servlet engine is at the 2.0 level of the servlet API.

Although you can create servlets that run in the Domino servlet engine, the
WebSphere servlet engine offers a number of advantages.

2.4.2.2 JavaServer Pages (JSP)

From version 3.5.2 WebSphere contains support for the JSP 1.1
specification. This means you can encapsulate the access to Domino
databases into custom tags and build JSPs containing Domino R5 data.

As we discussed previously, this separation of Java code and HTML design
makes the development of display pages easier than in servlets. In addition, a
JSP can be created by a Web designer without any Java knowledge.

Domino R5 does not support JSPs, so WebSphere is your only option if want
to use this development model. The next major release of Domino will contain
JSP support and even custom tags that create support for all the back end
classes of Domino.

2.4.2.3 Enterprise JavaBeans (EJB)

EJBs can be very useful if you want to encapsulate business logic or access
legacy systems. In Domino applications, you can access EJBs from servlets,
JSPs, or even Domino Java agents.

Since Domino R5 includes no EJB container and EJBs can only run in EJB
containers, you must deploy your EJBs in a different application server. The
ability to run EBJs is another important benefit WebSphere adds to Domino
R5.

2.4.2.4 Distributed execution of tasks

In 3.7, “Connecting Domino to WebSphere V3.5 via a network connection” on
page 76, we show how Domino and WebSphere can be installed in a
distributed environment. This enables you to use the HTTP server of your
choice and to distribute your application across your network. You can
include firewalls between your servers. The user does not see that he or she
is using a distributed application.
Especially if your application only consists of servlets and JSPs, this opens a new flexibility that Domino R5 does not contain by itself, because it is one integrated server.

Note though that this may change again with the next major release of Domino where the inclusion of custom JSP Domino tags are being considered as well as de-coupling the HTTP server to allow it to run on another machine for better security and scalability.

2.4.3 Why should you build your Web application with WebSphere?
IBM WebSphere is the right tool if you need a very high scalability or if accessing enterprise data and legacy systems is the main target of your application.

2.4.3.1 Scalability and distributed computing
WebSphere already is a very scalable server if it runs on a single machine. But WebSphere also contains many possibilities to enhance its scalability by distributing the execution among many computers.

This makes WebSphere the Web application server of choice if you expect a very large number of hits or transactions on your Web site.

2.4.3.2 Accessing enterprise data and legacy systems
Using EJB technology and its transaction management, WebSphere is the perfect tool for building Web sites that access enterprise data or legacy systems. You can encapsulate all transactions into EJBs and reuse them from multiple servlets, EJBs, and Java applications.

2.4.3.3 Writing vendor-independent code
Java code has the ability to run in any Java virtual machine. For example, servlets that were written and tested using the Domino R5 servlet engine can be deployed on WebSphere without any code change. The same is true for different JSP compilers.

In EJBs, even the distribution and the access of databases can be programmed to be vendor-independent.

This protects your investment in writing WebSphere code.

2.4.4 What Domino R5 can add to WebSphere applications
The WebSphere programming model is very flexible and allows you to build almost any application. This poses the question of what can Domino do for your WebSphere application?
You certainly would be able to reproduce everything Domino does in WebSphere, but compared with using Domino, the cost would be higher and the time to market much slower.

For example, there are not many Internet or intranet sites that have an amount of traffic and a number of information pages that Domino cannot handle. In Part 2, “The Redbook Banking example application” on page 181 we show how to combine a content management system that was built in Domino with the transactional capabilities of WebSphere.

Domino delivers much functionality that you could build in WebSphere, but at a much higher cost. This is why Domino is positioned as an application accelerator for WebSphere.

2.5 Summary

In this chapter we have given a brief overview of WebSphere’s internal structure and the components in a WebSphere transactional application. We have discussed:

- WebSphere administrative domains
- WebSphere administrative server
- WebSphere application server and its components:
  - Servlet engine(s), which provide a runtime environment for:
    - Servlets
    - JavaServer Pages
  - Enterprise JavaBean Container(s), which provide a runtime environment for:
    - Enterprise JavaBeans

We also have given you some hints on the selection of Domino or WebSphere as the Web application server for a specific solution and pointed out why you should consider using both servers together for your solution.
Chapter 3. Installation and setup

This chapter describes how to install and configure Domino and WebSphere to work together. There are many possible variations of a Domino and WebSphere installation. We do not intend to replace the installation guides included with the products. Instead, we will walk through a scenario where we install Domino Application Server R5.0.6a and WebSphere Application Server Advanced Edition V3.5.2 on computers running the Windows 2000 Server operating system.

The terms WebSphere and WebSphere Application Server (or WAS) will be used interchangeably here, even though WebSphere refers to an entire product line. Other products will be referred to by their full names.

The intended audience is a Domino or WebSphere developer with some Domino administrator experience who wishes to test and develop solutions with WebSphere-Domino integration.

3.1 Introduction

This chapter describes the installation of DB2 and WebSphere on one computer and Domino on another computer. This configuration allows separation of the components so that, for example, WebSphere and DB2 could be installed behind a firewall, with the Domino HTTP server in a DMZ. This is a high-level view of the installation steps we will be going through to install WebSphere and Domino and separate computers:

1. Check for necessary prerequisites and network configuration.

   Working with the WebSphere computer:

2. Log on as a user with administration privileges for WebSphere and DB2.

3. Install DB2 UDB v6.1 or v7.1 Enterprise Edition and its current fixpack (Fix Pack 4 for v6.1 and Fix Pack 1 for v7.1).

4. Install WebSphere V3.5, configure, apply Fix Pack 3 (on the same machine as DB2) and any relevant e-fixes.

   Working with the Domino computer:

5. Install and configure Domino R5.0.6a, along with a core WebSphere code installation with the fixes noted in step 4 to support remote connectivity to WebSphere.

6. Reconfigure Domino R5.0.6a to use the WebSphere Domino 5 plug-in and access WebSphere via a network connection.
7. Verify servlets can be loaded successfully.

Although we do not illustrate this, it will also be very straightforward to install DB2 on a third computer with further security and backup. If this were done, we would still need to install a DB2 client on the WebSphere computer to allow WebSphere to access its configuration database.

In Chapter 4, “WebSphere - Domino security and single sign-on” on page 109, we describe how to set up a Single Sign On environment for these computers. We also discuss the prerequisite security setup for SSO on both platforms.

3.1.1 Domino and WebSphere on the same computer

If you want to install Domino and WebSphere on a single computer (for instance, for development, testing, or small application purposes), follow the same steps as for two computers up to and including the procedures described in 3.6.2, “Installing the Domino administration client” on page 75.

Section 3.7, “Connecting Domino to WebSphere V3.5 via a network connection” on page 76 is not relevant for a one-computer installation. In 3.8, “Installing Domino and WebSphere on the same computer” on page 104 we discuss consideration for a one-computer installation.

3.2 Prerequisites

In this section we list the hardware and software requirements for the computer you want to install on, as well as the different product software levels required.

3.2.1 Platform

**Hardware:**

- Pentium III or higher, 256 MB absolute minimum, 512 MB recommended for WebSphere 3.5. We used 512 MB for the computer we installed WebSphere and DB2 on and 385 MB for the computer we installed Domino on for our testing, with satisfactory results.

- At least 600 MB free on the drive or drives used to install the products. The disk space requirements of the products after installation are:

<table>
<thead>
<tr>
<th>Product</th>
<th>Disk Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2</td>
<td>475 MB</td>
</tr>
<tr>
<td>Domino</td>
<td>300 MB</td>
</tr>
<tr>
<td>IBM HTTP Server</td>
<td>20 MB</td>
</tr>
<tr>
<td>WebSphere</td>
<td>220 MB</td>
</tr>
</tbody>
</table>
The DB2 space requirements can be reduced by 125 MB if you choose to install the DB2 Administration client on a separate machine. The space requirement for Domino can also be reduced to a certain extent by choosing to install fewer components. For example, it is possible to not install the help files. These files alone require 50 MB. Similarly, the space for WebSphere specified above includes documentation files of about 60 MB, which would not be needed in a production installation.

**Software:**
- Microsoft Windows 2000 or Windows NT. Either workstation or server code can be used. We used the server code for our testing.
- TCP/IP networking with a fixed IP address for each machine.

### 3.2.2 Product software levels

The product software levels we used were:

- DB2 Universal Database Version 7.1 Enterprise Edition plus Fix Pack 1. (DB2 UDB Version 6.1 with Fix Pack 4 is also supported for WebSphere 3.5; we used v7.1 for our testing.)
- Domino R5.0.6a.
- WebSphere Application Server V3.5 plus Fix Pack 3.

Higher versions of the products should also work. The highest currently available product level should be used except where specifically stated otherwise. However, care should be taken to check WebSphere for recent e-fixes. For example, the WebSphere Fix Pack 2 ‘broke’ the Domino DSAPI plug-in for WebSphere so that Single Sign On (SSO) did not work; this was resolved by e-fix PQ45555.

### 3.3 Creating a user with administration rights for DB2 and WebSphere

WebSphere Application Server, Domino, and DB2 must run under the permissions of a user or as system services. For testing purposes, it is more flexible to use a user ID with rights to run as an extension of the operating system rather than load the products as system services (however, WebSphere Application Server and DB2 must run as system services). For a production system, these products should run as services so that they will automatically load when the system is started without operator signon.

In this section we describe how to create a Windows 2000 user ID with rights to run as an extension of the operating system. To do this, you must have the right to create an ID on your local machine (and in any Windows
2000 or NT Domain Control server’s user registry if your machine logs onto a network).

---

**User ID has an 8 character size limit for V6.1**

The user ID must be 8 characters or less to work with DB2 UDB 6.1. Thus, the default Windows 2000 Administrator user name *Administrator* will not work with DB2 UDB 6.1. We used *db2admin* for our testing. However, in DB2 UDB 7.1 this restriction has been lifted.

If you already have a user ID (no longer than 8 characters for DB2 UDB version 6.1) with the right permissions, you can skip this section.

A new user ID can be created in Windows 2000 using the *Computer Management Panel* or the *Users and Password Wizard* in the control panel. Here we will describe the steps using the the *Computer Management Panel*.

1. Start the Windows 2000 Computer Management panel by selecting:
   
   **Start -> Programs -> Administrative Tools -> Computer Management**

---

**Administrative Tools menu**

If you cannot find *Administrative Tools* in your Windows 2000 start program menu, it probably has been deselected in your start menu settings. You can access *Administrative Tools* via the Control Panel or you can activate the menu again by selecting:

**Start -> Settings -> Taskbar & Start Menu...**

Then click on the *Advanced* tab in the Taskbar and Start Menu Properties box and select the checkbox for *Display Administrative Tools* in the listbox for Start Menu Settings. Click **OK** to save your changes, and you should now be able to select Computer Management.

2. In Computer Management select the left pane, expand the tree under Computer Management (local), and then expand the node **System Tools**. Expand **Local Users and Groups** to display the Users and Group Folders as shown in Figure 13.
Figure 13. Windows 2000 Computer Managerment

3. Either open the Users Folder or right-click on the Users Folder and select **New User ...** This starts the New User dialog shown in Figure 14 on page 44.

4. Fill out the user information. We specified the user name **db2admin**.

Make sure to specify a password you can remember. Change the default setting for password expiration so only **Password Never Expires** is selected.

Click the **Create** button to create your new user and then click the **Close** button to dismiss the dialog box.
5. Right-click the new user ID in the Computer Management window and select **Properties**. Once the Properties are displayed, select the **Member of** tab on the top of the panel to show the groups the user belongs to. This is shown in Figure 15 on page 45.
Click the **Add** button on the lower left of the panel. Highlight **Administrators** in the upper pane displayed and click the **Add** button in the middle of the panel as shown in Figure 16.

**Figure 15. Group membership for new user ID**

**Figure 16. Adding new user ID to administrators group**
Click **OK** to confirm and return to the Computer Management window.


7. Select **Start -> Programs -> Administrative Tools -> Local Security Policy** from the Windows 2000 desktop to set the user rights. Expand **Local Policies** in the left pane and select **User Rights Assignment** as shown in Figure 17. In the right pane you can see a list of system policies which users can be granted.

8. Right-click the entry **Act as part of the operating system** to get a pop-up menu and select **Security** .... You will be presented with a panel that shows the current list of users with the right to act as part of the operating system (the list may be empty).

9. Click the **Add** button to select the new user ID (db2admin in our example) just created. You will be presented with a dialog similar to Figure 16 on page 45, allowing you to select the user ID (db2admin) to be granted the right **Act as part of the operating system**.

   **Note** You can also select groups such as **Administrators** to grant this right to.

10. Select the user and click the **Add** button to add the local security right to the user. Click **OK** to return to the Local Security Settings Policy window.
11. The new user also needs the following rights if DB2 is to run under the user ID just created:
   - Create a token object
   - Increase quotas
   - Replace a process-level token

   You can go ahead and assign them to the user. However, the DB2 install process will add these rights if they are not set, provided the right *Act as part of the operating system* is set.

   ![Local Security Policy Setting](image)

   **Figure 18. Adding a user to a system policy setting**

12. Finally, close the Local Security Policy Setting panel and log on with the new user ID just created.

   If you are unable to log on with this ID, you need to resolve the situation (perhaps with your administrator's help) before proceeding. It may be that you need to create a stand-alone server not associated with your existing Windows 2000 or NT Domain for testing purposes.

   The user ID we just created only needs to be on the WebSphere computer. However, to make it easier to work with both computers, we recommend that you create the same user ID on both computers.
We now describe the installation of DB2 on the WebSphere computer.

WebSphere requires a database system that supports Java to store its configuration and state information in the WAS database. If your system does not have DB2 or one of the other database systems supported by WebSphere, the WebSphere installation will automatically install a limited subset of DB2s functionality called Instant Database. For a production environment, or to have access to all of the DB2 features, you should obtain the DB2 package separately.

The supported level of DB2 is 6.1 plus fix pack 4 or higher. We used DB2 UDB Version 7.1 for our tests.

Make sure you are logged on to Windows NT with the user ID you created in 3.3, “Creating a user with administration rights for DB2 and WebSphere” on page 41, or another user ID with similar rights. Although this is not strictly necessary to run the install program, you will need to be logged on with this user ID when accessing DB2 after installation.

3.4.0.1 Installation of DB2 UDB Release 7.1 Enterprise Edition

DB2 UDB can be installed either from a zip file from the IBM software Web site or from a product CD. If the installation program on the CD doesn’t start automatically (or if you are installing from an unpacked zip file) you must run the setup.exe program manually.

1. First, the welcome screen is displayed as shown in Figure 19 on page 49. Click the Install tab in the left column of the screen to begin the installation.
2. The product selection screen shown in Figure 20 is displayed. We selected all three products. You must select DB2 Enterprise Edition, and should select DB2 Administration Client if you wish to have WebSphere connect to DB2 on a remote computer in the future.
3. You are presented with a selection of a Typical, Compact, or Custom installation.

   Select Typical and press Next.

4. On the next screen you can select where to install DB2 ("Choose Destination Location").

   The installation directory can be changed to a disk with adequate space if the default drive (C:) does not have enough space. Click the Browse button to specify a new location for DB2. In our case, we accepted the default location of C:\Program Files\SQLLIB

---

**Note: hint for reinstallation**

If you later uninstall DB2 and then wish to reinstall it, ensure that the directories SQLLIB, DB2, DB2CTLSV, and DB2LOG are completely removed first. If you do not do this, it may be impossible to recreate the WebSphere WAS database.

5. You are then prompted to supply a user ID and password for the DB2 administration server to run under.

   ![Figure 21. Specifying the user ID to be used by DB2](image)

   If necessary, replace the defaults provided with the user ID set up earlier with Windows 2000 computer management (db2admin in our case). This user ID will also become the default user ID in DB2.

   Ensure that the option *Use the same values for the remaining DB2 Username and Password settings* is selected.
Click **Next** to continue.

A warning will be shown if you did not assign all necessary local security rights to your user, as described in 3.3, “Creating a user with administration rights for DB2 and WebSphere” on page 41. However, as long as you have assigned your user the right to act as part of the operating system the DB2 installation can assign the other rights and you can continue with the installation.

6. The next screen displays the installation options chosen. You can review these and, if necessary, click **Back** to change your selections. Click **Next** to confirm your choices and start the installation of the program files.

DB2 will install and configure itself. At the end of the configuration process you will be presented with the option to install the OLAP starter kit. Select **Do not install the OLAP starter kit** and press **Continue**. You will be presented with a confirmation screen. Click **Finish** to complete the installation.

You may be prompted to reboot your machine. If so, do so.

By default, DB2 will display the **First Steps** program, which you can use to create a sample database if you wish.

Creating a sample database is not necessary to continue, but is an easy way to verify that DB2 has been installed successfully. In our case we created the database named **DB2 UDB Sample**.

DB2 UDB is now ready to have its fix pack installed.

### 3.4.1 Installation of DB2 FixPak 1 for DB2 V7.1 UDB

WebSphere V3.5 requires DB2 7.1 FixPak 1 or higher. You can download the current DB2 fixpaks from the support area of the DB2 Web site at:

http://ibm.com/db2

Look for **Maintenance for DB2 Universal Database Version 7.1 products** and select the fixpak that applies to your product. In our case we installed FixPak 1.

Before you install the fixpak you must stop all DB2 services.

1. Open the Windows NT Services Panel by selecting:

   **Start -> Programs -> Administrative Tools -> Services**

   The Windows NT Services panel opens as shown in Figure 22 on page 52.
2. Stop all services in the panel beginning with DB2 by selecting them individually and clicking the **Stop** button. Note that the service “DB2-DB2” must be stopped last since other DB2 services depend on it running. Also, if running DB2 UDB 7.1 the services *Warehouse server* and *Warehouse logger* will be stopped when you stop the DB2 services.

There may be other services that use DB2 (such as Netfinity Support Manager) which should also be stopped. If any are missed, the initial prompt on starting the fixpack installation will identify running services and give you the option of stopping them before proceeding or cancelling the installation.

3. Once the services have been stopped, run the Setup.exe program from the DB2 FixPak 1 package.

The installation is similar in appearance to the original installation. The difference is that any selections already made in the original product installation will already be selected. For example, if the original installation were to `D:\SQLLIB` the fixpak would be installed to this directory.

4. You will be prompted to reboot the computer. Do this and the installation of the DB2 fixpak is complete.

You are now ready to install WebSphere V3.5
3.5 Installing WebSphere Application Server V3.5

You can either install WebSphere from a product CD or one large installation file. If you use a product CD the installation program should start automatically when you insert the CD. If not, you have to run the setup.exe program on the CD. If you have one big file, simply start the installation by running the file. It will automatically unpack itself (as shown in Figure 23) and start the installation program. Note that the installation process will require 70 MB or more free in the system TEMP directory (normally the C drive) even if installation is to another drive.

![Unpacking IBM WebSphere Application Server...](image)

*Figure 23. Installing WebSphere with a single large executable*

If your installation file supports installation using different languages, you must pick **English** to follow the procedure we describe here.

1. The first installation panel says your Web server must be shut down before proceeding with the installation.

   Make sure that all HTTP servers on this computer are shut down before proceeding. Note that by default Windows 2000 server will be running the IIS HTTP service. Ensure that it is shut down and set to start manually in the services panel (see Figure 22 on page 52 for an overview of the services panel). The IIS HTTP service will be listed as **World Wide Web Publishing Service**. Ensure that it is stopped and its startup type is set to **Manual** by right-clicking the service, selecting **Properties** and changing **Startup Type** in the General tab.

   Click **Next** to continue with the WebSphere installation.

2. The Install Options panel is displayed.

   Select the radio button for **Custom Installation** or **Full Installation** and click **Next**.

3. If you selected **Custom Installation** on the previous panel, the next panel will show selectable WebSphere components. Select all of the
components shown. This panel is illustrated in Figure 24. If you selected **Full Installation** you will go directly to the panel described in step 5.

4. The next panel allows selection of the Web server you want to use with WebSphere. WebSphere will install plug-ins for the Web server(s) you select as shown in Figure 25.

WebSphere will only show plug-ins for the IBM HTTP Server 1.3.12, Apache 1.3.12 and IIS v4 and v5 if you have not installed any other Web server software on this computer. Simply accept the default selection for
the IBM HTTP server. Note that plug-ins for Domino and IIS will still be available for installation later if necessary.

5. You will be prompted for a user ID for WebSphere server to log on to the operating system as shown in the Security Options Panel (Figure 26). Enter the user ID created (db2admin in our case) for DB2 administration in 3.3, “Creating a user with administration rights for DB2 and WebSphere” on page 41.

![Security Options Panel](image)

Figure 26. Prompt for user ID and password for WebSphere server

6. The next panel allows you to specify the program folders for WebSphere and the IBM HTTP server. (If you have previously installed the IBM HTTP server on this computer, you will not be given a location for it to install to.) Accept the default suggestions or specify alternative(s) on the panel shown in Figure 27 on page 56 and click Next to start the installation of the program files.
7. The next panel, shown in Figure 28, is the Database Options panel. This configures the database access and user IDs to be used by WebSphere. The database access is used by WebSphere to store the server configuration in a set of tables in the WAS database. Note that this default selection can be changed later. For example, the database name can be changed to point to a remote DB2 database on another computer if desired, provided the link is created in the DB2 Client Administration environment.

We kept the default database name was and specified the database user ID as db2admin.
8. If you installed DB2 UDB Version 7.1 as we suggested you will get a warning, as shown in Figure 29. Click OK to continue the installation.

![Database version warning](image)

*Figure 29. Database version warning*

9. Accept the suggested program folder for DB2 or select another and click Next to start the installation.

10. When the files have all been copied you will receive a confirmation screen like the one shown in Figure 30. Click Finish and accept the suggestion to reboot the computer to complete the installation. If you wish, you may also accept the offer to read the README file. It can also be read later if you wish.

![WebSphere installation completion panel](image)

*Figure 30. WebSphere installation completion panel*

Once the computer has been restarted, you are ready to install FixPak 3 for WebSphere Application Server version 3.5, which we describe in 3.5.2, “Applying WebSphere V3.5 FixPak 3” on page 64.

However, we will first verify that WebSphere has been installed correctly and also configure WebSphere to accept URLs that include domains.
3.5.1 Confirmation of successful installation of WebSphere v3.5

You can start the WebSphere Administrative Server from either the Windows 2000 services panel or the Start Programs menu (Start-> Programs-> IBM WebSphere -> Application Server V3.5 -> Start Admin Server). The service should start and you should be able to start the WebSphere administrative console to monitor and configure the administrative server by specifying Start-> Programs-> IBM WebSphere -> Application Server V3.5 -> Administrator's Console). If WebSphere has been successfully installed, both should start. Once the console is displayed, click the ‘+’ sign beside the node in the left pane with your server’s host name. For our example, the server’s host name was Thor-2000. Locate the sub-node ‘Default Server’ and right-click it. Select Start from the context menu (or press the Start icon on the toolbar) to start the default server. Your administrative console should look similar to Figure 31.

![Administrative console with default server ready to start](image)

If the WebSphere Administrative Server does not start, there may be a problem with the creation of the WebSphere Administrative Database (the WAS database). Normally, once the system reboots after the installation a script is run to create the WAS database. If this database is not created, you will be unable to start the WebSphere Administrative Server and will get very little diagnostic information. Although the creation is run in a command window, it executes rapidly so that it will likely not be possible to see whether or not it was successful.
The easiest way to confirm successful creation of the WAS database is to navigate to the WebSphere logs directory (C:\WebSphere\AppServer\logs in our default installation) and view the wasdb2.log. This is the screen output of the WAS database creation session. The command window in Figure 32 shows a successful creation. If your script does not resemble our results you must investigate and correct the situation before proceeding. You can try to issue the commands from a DB2 command window manually. If this does not work, you may need to consider uninstalling DB2, ensuring that all directories and their contents are completely removed, as mentioned in “Note: hint for reinstallation” on page 50. You should then reinstall DB2 and rerun the commands to create the WAS database. Problems creating the WAS database are most likely to occur if WebSphere has been previously installed and uninstalled on your computer before you started your WebSphere installation.

![Screen output of WAS database creation](image)

**Figure 32. Confirmation of successful creation of WAS database**

If both the WebSphere Administrative Server and the WebSphere administrative console start successfully, you are ready to test your implementation from a Web browser. The IBM HTTP server service is by default installed to start automatically. If the service has been changed to manual start you can start it in the program menu installed with the product (Start -> Programs -> IBM HTTP Server -> Start HTTP Server) or from the Windows 2000 Services panel. Both WebSphere and the IBM HTTP server can now be tested. The result of accessing the IBM HTTP server from a browser is shown in Figure 33 on page 60.
It is sufficient to access one of the default servlets installed with the WebSphere Administrative Server; for example, the simple snoop servlet (described in the WebSphere InfoCenter). In our case we chose the showCfg servlet (by default with virtual URL hostname/webapp/examples/showCfg) since this servlet will be useful when we need to confirm successful connection from Domino. Specify only the host name (thor-2000 in our case), IP address, or localhost; see 3.5.1.1, “Adding aliases to the WebSphere default_host” on page 62 for changes we must make before we can use names that include domains.

We entered the following in a browser running on our WebSphere computer:

http://thor-2000/webapp/examples/showCfg

If this does not work first start the WebSphere administrative console and make sure that the default server is started.
Note: Servlet names are case sensitive, but can have several mappings. The showCfg servlet has the following four mappings: showCfg, ShowCfg, showConfig and ShowConfig. Thus you will get an error if you write showcfg in all lowercase or all uppercase.

The results of accessing the showCfg servlet are shown in Figure 34. We will be reconfiguring the WebSphere server as part of the process of enabling Domino to access it and utilize SSO; the showCfg servlet is a useful way to double check that the configuration is as expected. Note, as well, that the configuration display is scrollable and has links to detail information.

Figure 34. WebSphere showCfg servlet
3.5.1.1 Adding aliases to the WebSphere default_host

The observant reader may notice that we used URLs of the form: http://hostname/ rather than http://hostname.domain (in our case http://Thor-2000 rather than http://Thor-2000.lotus.com). This is because, by default, WebSphere is initially configured to recognize only the hostname in the rules accessed by the Web server plug-in file. For the installation of the Single Sign On option we will also need to recognize the domain name in URLs. This is done in the WebSphere Advanced Administrative Console by selecting default_host (installed by default when WebSphere is installed; it is possible to define other virtual hosts if necessary) from the tree in the left pane, right-clicking and selecting Properties in the context menu. When a dialog box is shown, select the Advanced tab as shown in Figure 35.

![Figure 35. Defining aliases for the default host recognized by WebSphere]

Scroll the Host Aliases list box until there is a blank entry.

Enter the full host and domain names (referred to as aliases on this panel) to be recognized by the default server on this WebSphere server.

We entered the following (press Enter or click Apply for each entry):

- Thor-2000.lotus.com
- Thor-2000.lotus.com:443
Note that you cannot enter another host alias until you press the Enter key or the Apply button for the current alias.

In addition, we made the following entries for the server we intended to install Domino on:

- Odin
- Odin.lotus.com
- Odin.lotus.com:443
- The IP address of the Odin host

These entries will be necessary when we connect Domino to the WebSphere server.

The entries without port information assume the normal HTTP port number of 80; if you wish to use a non-standard port for HTTP or use HTTPS for SSL (standard port number 443) you need to add additional entries in the list. We added entries for the hosts using port 443 so that later we would be able to use HTTPS between the Web browser and the (Domino) HTTP server. Another reason you might need to add port numbers with extra host name entries would be the case of running more than one HTTP server on a single computer. In this case, you will need to have the HTTP servers listen on different ports; often port 8080 is used in place of the default port 80.

Once you have entered all the host names and aliases you need, click **OK**. You will have to stop and start the **Default Server** under your host name (Thor-2000 in our example) using the context menu by right-clicking or by highlighting it and selecting **Stop** and **Start** from the toolbar. You should now be able to use any of the hostnames for the server you installed WebSphere on.

The results of your changes are written to 3 properties files in the WebSphere temp directory (\WebSphere\AppServer\temp by default):

a. queues.properties
b. rules.properties
c. vhosts.properties

These files and their use are be described in 3.7.8, “Configuration of OSE remote transport for Domino” on page 93. For the moment, it is sufficient to note that clicking the button “Regen Plugin Co ...” shown in Figure 56 on page 94 will recreate the vhosts and rules properties files, but not the queues
properties file; this will be automatically regenerated by WebSphere within 5 minutes of a change.

Again, you can confirm these entries are recognized by using a Web browser to invoke the showCfg servlet (specifying a host known to work!) and following the link labelled Configuration for Virtual Host: *default_host*. The resulting panel is shown in Figure 36.

![Configuration for Virtual Host default_host](image)

3.5.2 Applying WebSphere V3.5 FixPak 3

We upgraded WebSphere V3.5 using FixPak 3. This fixpak can be applied to a system that already has Fix Pak 1 or 2 installed (FixPak 2 would only install over the base release, but this is not the case for this fixpak). The fixpak can be obtained from http://www-4.ibm.com/software/webservers/appserv/efix.html

Download `was35_adv_ptf3.zip` into an empty directory on the computer you installed WebSphere on.

Ensure that the following two servers are stopped:

- WebSphere Advanced Administrative Server
- IBM HTTP Server
Both can be stopped from the Windows 2000 services panel; the IBM HTTP Server can also be stopped from the Windows 2000 Programs menu.

Unzip the FixPak zip file (*was35_adv_ptf3.zip*).

Start a command prompt window. Change to the directory to which you unpacked the fixpak files and run the *install.bat* file.

Follow the instructions on the screen. At different points during the update installation you will be prompted for input:

- The WebSphere installation directory (*C:\WebSphere\AppServer* in our example)
- Whether you want to upgrade the examples (we replied *Yes*)
- The HTTP server’s document root path (*C:\IBM HTTP Server\htdocs* in our example)
- Whether you want to upgrade the HTTP server, as it may have implications for other applications on the system (we replied *Yes*)
- The HTTP server’s installation directory (*C:\IBM HTTP Server*)

Once installation is complete, you will get the message **IBM WebSphere Application Server V3.5.3 Advanced Fixpack install complete.** If you later wish to review the fixpak installation, a log is stored in the WebSphere logs directory in the file *was35_ptf_3.log*. This file can be reviewed with any text editor, such as Notepad or Wordpad.

**Date and number format problem with FixPak 2**

We found that FixPak 2 would report insufficient disk space if we used non-US date and number formats. Resetting the computer’s locale to US allowed us to apply the fixpak. We were then able to reset the computer to the original settings. However, FixPak 3 did not have this problem.

It is not necessary to reboot the computer: WebSphere with FixPak 3 installed is ready for service. You may wish to confirm this by starting the WebSphere Server and IBM HTTP server and then accessing the computer with a Web browser as before. If you do so, and access the *showCfg* servlet, you will be able to confirm the new product version by scrolling down to the section “WebSphere Application Server Version Info” as shown in Figure 37 on page 66, and verifying that the entry for Product Version is now 3.5.3 (it was 3.5 before installing the FixPak).
3.5.3 Apply any relevant e-fixes

If you have problems with your application in WebSphere and suspect you have found a WebSphere product problem, check the support download page for relevant e-fixes:


E-fixes are individual fixes for critical problems. They have been individually tested, but not integration tested, and should only be applied if you have a critical problem without a valid workaround. They may be applied to both Standard and Advanced Editions, except where specifically noted. All e-fixes are rolled into the next scheduled FixPak. Each fix has a readme file with installation instructions.

This completes the installation of WebSphere and its FixPak. We are now ready to install Domino on a separate computer (“Odin” in our example).

3.6 Installing and configuring Domino V5.0.6a

We now turn our attention to our Domino computer and describe how to install Lotus Domino.

The Lotus Domino R5 Server family consists of Domino Mail Server, Domino Application Server and Domino Enterprise Server. If you want to do more with Domino than just use its HTTP stack to integrate with WebSphere you should install the application or the enterprise server.
To use the Domino HTTP stack and enable Single Sign On, WebSphere V3.5 requires Domino Server R5.0.5 or higher. We installed Domino Application Server R5.0.6a since it was the latest level available.

1. If you install Domino from a CD, the installation program should start automatically. If it does not, start the installation by running `setup.exe` from the installation CD.

   A welcome screen will be shown.

2. Click **Next**, read the license agreement and click **Yes** to accept it.

3. Enter your name and company on the next panel if the installation program hasn’t been able to pick up this information from the system.

   Click **Next**.

4. Specify where you want the Domino program and data files placed or accept the default locations. We chose to change the default locations to C:\Domino and C:\Domino\Data. The installation program will create these directories if they don’t already exist.

   Click **Next**.

5. Choose the type of server to install.

   In our case we selected Domino Application Server as shown in Figure 38 on page 67.

![Lotus Domino Installation](image)

**Figure 38. Selecting Domino server type**
You can use the Customize button to further refine the installation process. If you click **Customize** you will get a list of options to select or deselect as shown in Figure 39.

![Figure 39. Selection of Domino components to be installed](image)

For example, you may not need the help files because you will install the help together with the administration client on another machine or in another directory later on. Deselecting help from the server install will save you around 40 MB.

Make sure that **Domino Web Services** is selected; other components can be selected or deselected at will. For testing purposes, you should not select the option to allow Domino to run as a Windows 2000 service. It is possible to add more components later by running the installation program again and only selecting the desired components to be added.

6. Click **Next**. It does not matter whether you picked the default installation or the customized one. You will be brought to the panel where you select which Program Folder to add the Domino Server to.

7. Click **Next** to accept the default suggestion (or change it as desired; we chose to use ‘Domino’ as the program entry) and to start the installation of the files.

8. Once all the files are installed, the installation program shows a message confirming installation has been completed. This panel also gives you an opportunity to read the ReadMe file for this release.

Click **Finish** to end the installation program.
3.6.1 Domino Server configuration and setup

After the Domino Server files are installed the server must be configured before we can start it.

1. Start Domino by selecting **Start -> Programs -> Lotus Applications -> Lotus Domino Server** (the third entry is the default; you may have changed it during installation) from the Windows 2000 task bar.

   This will launch the Domino configuration program after a brief display of the server console.

2. The initial configuration screen asks whether this is the first or an additional server in your Domino domain.

   We selected **First Domino Server** to set up a stand-alone test domain, as shown in Figure 40 on page 69.

   ![Figure 40. Initial Domino Server setup: Panel One](image)

3. Click the forward button (> at the top of the pane.

   The next panel asks you to specify whether you want to set up the server using a Quick and Easy or an Advanced Configuration, as shown in Figure 41 on page 70.

   ![Figure 41. Advanced Domino Server Setup](image)
4. Select Advanced Configuration so that further options can be selected on the succeeding screen.

To continue, click the > button on the top bar.

5. The Advanced Configuration panel is shown.

We selected a reduced set of services for the server since we were only going to test it as a Web server. Note that the choices here only change the initial configuration setup of the server and can easily be changed later. For example, we specified using the Domino HTTP stack, but this is easily changed to use IIS during testing.

We added the following to the default set of services:

- HTTP for Web browsers (Both Mail and Applications)
- IIOP for Web browsers (needed for our application example)
- LDAP for Internet Directory Services (We will use this as our Directory for both WebSphere and Domino.)

We deselected the following additional services:

- Calendar Connector
- Schedule Manager
When you select HTTP you will be prompted whether you wish to use the Domino HTTP Engine or the IIS HTTP Engine (since, by default, IIS is installed with the Windows 2000 server operating system). Accept the default of Domino. The selections we made are shown in Figure 42. Note that not all of the panel shows in this figure; however, the remaining entries are not relevant for our testing.

Figure 42. Initial Domino Server setup: Panel three

6. Once the desired initial services are selected, again click the > button on the top to continue.

The Administration Settings panel will be shown.

- Here you specify the names of your Domino domain (ITSO-DOMWAS in our case), certifier/organization name (DOMWAS), and administrator identity (Domino WebSphere Administrator).

Be sure to keep a record of the passwords you used for the certifier and the administrator since these are not retrievable if you lose them.

- Select Customize in the Ports section under Network Options. You may have to scroll down a bit to do this. Then click the Edit Ports.
button. This opens a new window where you can see all communication ports that will be activated by default.

Disable all ports except **TCP/IP** and click **OK** to return to the setup panel.

The setup panel should now look similar to Figure 43.

![Figure 43. Initial Domino Server setup: Panel four](image)

7. Verify all of the names and options are correct and click **Finish** to complete the server setup.

   *Do not* close the installation window when the installation is finished. We will also configure access control lists as described in the next step.

Three Notes ID files will be created during the setup:

- **Cert.id**: The certifier ID file for the new organization.
- **Server.id**: The server’s ID file.
- **User.id**: The administrator’s ID file. By default, this will be saved in the administrator’s person record in the Domino Directory.
Once the setup has finished you will see the panel shown in Figure 44.

8. There is a button labelled *Set Access Control Entry* near the middle of the panel. Click this button to set default ACL entries for all databases and database templates just installed. By default the “Administrators” group will be filled in. Also select the entry to add “Anonymous with no access” since this will be needed later to test SSO.

9. Click **OK** to accept the entry and return to the final Domino configuration panel. Note that the panel will now report the number of databases and templates whose ACLs were altered. In our case 81 databases were
updated, but this number may be greater or fewer if you selected installation options different from the ones we used.

10. Click the **Exit Configuration** button to close the setup program. You may have to scroll down a bit to see the button.

### 3.6.1.1 Verifying the Domino server configuration

We now check that the Domino HTTP stack loads correctly and that it can be reached from a Web browser.


2. Ensure that the HTTP task is running by issuing a `show tasks` command from the console or by observing the console log when the server starts.

   Figure 46 shows part of the output from the `show tasks` command. The bottom line confirms that the HTTP task is loaded.

   ![Figure 46. Domino Server console showing that the HTTP task loaded](image)

   **Note:** The Domino HTTP task will fail to start if another HTTP task is running using the default HTTP port 80. For example, if you have IIS installed on your server, it should be stopped from the Windows 2000 Services panel and set to be started manually.
3. Next, Web HTTP access to the Domino server can be checked by using a browser. Start your Web browser and type in the host name of your Domino server as the URL. (If you are running on the same machine, you can just type localhost as the URL.) Because of the access control changes you made previously, you will be prompted with a browser authentication panel. Simply enter the full name of the Domino Administrator (Domino WebSphere Administrator in our example) and the password you entered during Domino configuration.

4. If Web browser access works correctly you will see the default R5 home page in your Web browser, as shown in Figure 47.

![Figure 47. Home page displayed by Domino HTTP server](image)

3.6.2 Installing the Domino administration client

You need to install the Domino administration client on your server or another workstation (Lotus recommends using a separate workstation for administration). This will allow you to change the server’s settings easily, especially those in the Domino Directory; although much, but not all of the
testing we describe can be managed by direct access to the server's text console as illustrated in Figure 46 on page 74.

We will not describe the installation of Domino administrator in detail. Be sure that the Administration Client is selected for installation; you can accept all other default values during the installation.

One of the first things you can do after installing Domino Administrator is allow your users to run Java programs on the Domino server. This is described in 4.4.2, “Configuring the Domino server to support LDAP and IIOP” on page 123.

This completes the installation and configuration of Domino R5.0.6a for the purposes of this exercise.

### 3.7 Connecting Domino to WebSphere V3.5 via a network connection

We need to install a subset of WebSphere v3.5 and apply WebSphere FixPak 3 on the computer we just installed Domino on. During our testing we investigated the possibility of simply copying files from an existing WebSphere installation, but found a new install was simpler and faster.

Once this is done (and FixPak 3 has been applied), Domino can be configured to use the WebSphere DSAPI plug-in and communicate with the WebSphere server previously installed and configured.

Before we describe the actual installation of the WebSphere files, we discuss the different ways that we can connect our Domino and WebSphere servers. If you already know that you want to have Domino and WebSphere communicate via OSE or thin servlet redirectory, you can jump ahead to 3.7.6, “Installation of WebSphere components on the Domino server” on page 89 and read this section at a later time.

There are three basic ways to connect between Domino (actually, our discussion is relevant to any HTTP server) and WebSphere:

- **OSE remote**
  
  The Open Servlet Engine or OSE transport is used to communicate between an HTTP server WebSphere plug-in and one or more WebSphere servers running on another computer or computers.

- **Servlet redirectors**
  
  A servlet redirector is a special case of an application server that runs on the same computer as the HTTP server and its plug-in file. It receives
requests from the plug-in by using (local) OSE and forwards them to a remote servlet engine.

- Reverse proxy/IP forwarding

An HTTP reverse proxy intercepts HTTP requests and forwards them to an HTTP server behind a firewall. This HTTP server can be on the same machine as the WebSphere application server or on a different machine.

### 3.7.1 OSE remote

The Open Servlet Engine or OSE transport can be used to communicate between an HTTP server WebSphere plug-in and one or more WebSphere servers running on another computer or computers. The HTTP plug-in loads necessary files as needed from a core WebSphere installation on the HTTP server. WebSphere server does not run and is not configured on this server. However, configuration files need to be copied or generated from your WebSphere server(s) to support this method of connection. The configuration files can be manually copied and edited or generated via a configuration script. If you use the configuration script (supplied with WebSphere V3.5), the servers must all be in the same WebSphere administrative domain.

If a company implements network address translation (NAT) on a firewall between the (Domino) HTTP server and WebSphere Application server, this communications methodology will work. This is in contrast to the servlet redirector methods described in 3.7.2, “Servlet redirectors” on page 78, which use IIOP; these cannot be routed through NAT because this protocol imbeds IP addresses in the body of the IP packets carrying the protocol. The NAT process inspects and changes IP addresses only in packet headers, leaving the packet body contents (including IP addresses) unchanged.

OSE generally gives the best performance of all the connection methods. In some cases, overall performance using this connectivity may be better than communication to a local instance of WebSphere because of the separation of the HTTP server (and Domino application server) from WebSphere.

Communications using OSE remote are not encrypted by WebSphere (but could be if one used network encryption such as IPSec). The InfoCenter documentation (section 1.4.2.4) supplied with WebSphere V3.5 describes this methodology:

> Remote OSE is the preferred DMZ configuration unless some business requirement prevents its use.
3.7.2 Servlet redirectors

A servlet redirector is a special case of an application server that runs on the same computer as the HTTP server and its plug-in file. It receives requests from the plug-in by using (local) OSE and forwards them to a remote servlet engine session bean. Each WebSphere application server has a stateless session bean (the RemoteSRP bean) that listens for requests and forwards them to the servlet engine in the application server. The servlet redirector is thus an EJB client of the RemoteSRP bean and its receiving method. As for Enterprise JavaBeans, it uses JNDI to locate the bean. A schematic of this flow is shown in Figure 48 on page 79. We do not show the data flows for configuration since these depend on the specific servlet redirector topology selected.

IPSec provides cryptographic security services. These services allow for authentication, integrity, access control, and confidentiality. IPSec is similar to SSL, but operates on the network layer, making it completely transparent to applications. Any IP protocol can be used over IPSec. IPSec operates in two modes. Transport mode secures existing IP packets between source and destination. Tunnel mode encapsulates the existing IP packet inside another packet. You can use IPSec for Virtual Private Networks (VPN) or simply encryption of traffic between computers. To learn more about IPSec see the IBM Redbook TCP/IP Tutorial and Technical Overview, SG24-3376.
There are two major servlet redirector implementations: thin and thick. They have the following characteristics:

- **Thin servlet redirector** - This methodology runs a Java class to redirect servlet requests (sent by local OSE from the plug-in) to one or more WebSphere servers. WebSphere server does not run on the HTTP server, but some configuration is necessary to support the servlet redirector. This configuration is done by using a configuration script to import information from your WebSphere server. This communication methodology uses Internet Inter Object Request Broker Protocol (IIOP) to communicate between the servlet redirector and the WebSphere application server; this traffic is encrypted using SSL if WebSphere global security is enabled.
• **Thick servlet redirector** - This requires that WebSphere Application server be installed and running on the same computer as the Domino HTTP server. There are two variations of this basic configuration:

- **Thick servlet redirector with DB client** - In this configuration, the Administration server has a DB client to connect to the WebSphere administrative repository (or WAS, by default) used by the target WebSphere server(s). The *remote servlet redirector* is a separate Java process which accepts OSE connections (local or remote) and forwards them using IIOP to an instance of WebSphere (or, more accurately, a RemoteSRP bean with a specific servlet engine). A disadvantage of this method is that the user ID and password of the DB client is stored unencrypted in the admin.config file on the Domino server; WebSphere administrative server needs this to connect to the administrative repository when it starts.

  An advantage of this methodology is that any WebSphere configuration changes made to the remote WebSphere server(s) will immediately be reflected in the configuration of the thick servlet redirector since it shares the same administrative repository. In addition, the IIOP communications will be encrypted using SSL by WebSphere if WebSphere security is enabled.

- **Thick servlet redirector with administrative agent** - In this configuration, the administrative server’s behavior is modified with entries in the admin.config file on the forwarding server (the one with the Domino HTTP server) to tell the administrative server to run in *agent* mode. In this mode, the administrative server does not connect directly to the administrative repository, but instead connects via IIOP with the administrative server on another computer, which handles access to the administrative repository.

  Otherwise, this configuration is identical to the thick servlet redirector with DB client.

### 3.7.3 Reverse proxy/IP forwarding

We mention Reverse proxy/IP forwarding for completeness, but it is not really equivalent to the other configurations, where requests are forwarded between a (Domino) HTTP server and a WebSphere application server. Instead, an HTTP reverse proxy intercepts HTTP requests and forwards them to an HTTP server behind a firewall. The HTTP server could be on the same or a different computer than the WebSphere application server; if different, one of the above methods (likely OSE remote) would be employed to communicate. The advantages of this configuration are that the HTTP server (and, of course, WebSphere application server) can be located behind a firewall, and that...
identity of the HTTP and WebSphere servers are hidden from the requesting browser client.

We tested servlet redirection using OSE remote and thin servlet redirector. The setup for OSE remote is described in the following and thin servlet redirector is described in Appendix A, “Configuration of thin servlet redirector for Domino” on page 449. We also ran with multiple WebSphere administrative servers using DB2 clients connecting to a single WAS database, but we did not test the thick servlet redirector configurations.

3.7.4 Characteristics of the connection methodologies

The following table, extracted from the Infocenter documentation (section 1.4.2.9) supplied with WebSphere V3.5, summarizes the characteristics of the connection methodologies.

<table>
<thead>
<tr>
<th>Feature</th>
<th>OSE remote</th>
<th>Thin servlet redirector</th>
<th>Thick servlet redirector</th>
<th>Thick servlet redirector with admin agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compatible with WebSphere product security for applications</td>
<td>Yes</td>
<td>No? according to Infocenter (but our test with application security “worked”)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Avoids data access from DMZ</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Supports network address translation (NAT)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Avoids protocol switch in DMZ (only true for a reverse proxy)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Encryption of link between Web server and WebSphere application server (SSL - WebSphere)</td>
<td>No (but could use IPSec)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DB password on HTTP - Domino computer</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
### Table

<table>
<thead>
<tr>
<th>Feature</th>
<th>OSE remote</th>
<th>Thin servlet redirector</th>
<th>Thick servlet redirector</th>
<th>Thick servlet redirector with admin agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebSphere WLM enabled?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Performance relative to local OSE</td>
<td>95 - 100⁺%</td>
<td>70-85%</td>
<td>70-85%</td>
<td>70-85%</td>
</tr>
<tr>
<td>Administration</td>
<td>Manual</td>
<td>Manual</td>
<td>Automatic</td>
<td>Automatic</td>
</tr>
<tr>
<td>Avoids single point of failure (Admin Server on WebSphere target server)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Minimum firewall holes</td>
<td>1 (bootstrap - port 8110 for admin server - only used for intial configuration. This can be avoided by manual configuration), plus 1 per WebSphere Application Server</td>
<td>3 (RMI-IIOP, Location Service Daemon, bootstrap), plus 1 per application server</td>
<td>3, plus 1 per application server</td>
<td>3, plus 1 per application server</td>
</tr>
</tbody>
</table>

#### 3.7.5 Configuring Domino DSAPI plug-in connections to WebSphere

Requests from a Web server plug-in to WebSphere are *always* handled by a proprietary internal WebSphere protocol called Open Servlet Engine (OSE) transport. OSE uses interprocess communication provided by the native operating system(s). If the Web server is installed on the same computer as the target WebSphere application server (or a component like the servlet redirectors described earlier), OSE will typically use pipes if provided by the operating system. For remote connections, OSE uses TCPIP sockets to connect.

The behavior of the Web server plug-in (in our case, the Domino DSAPI plug-in) in all cases is controlled by three files generated by WebSphere which dictate how it uses OSE:

1. **vhosts.properties** - This file specifies the mapping from Web hosts to WebSphere virtual hosts. The entries we made in 3.5.1.1, “Adding aliases to the WebSphere default_host” on page 62 were written to this file.

---

82 Domino and WebSphere Together
Clearly, we expect to find the host name of the Domino server (specified as either a host, host with domain, or IP address) in this file; if found, it is then mapped to a target virtual host. WebSphere will create a virtual host named “default_host” on installation. Once a target virtual host is identified for the request, we can then parse the remaining part of the URL (or URI).

2. **rules.properties** - As the name suggests, this lists the rules that the plug-in uses to decide whether a request is to be handled by WebSphere. Each valid combination of virtual host and URI is listed together with the OSE queue to which the request for the resource should be forwarded; there could be multiple queues to direct requests to multiple servers. Any URL that is not found in the rules is returned to the native Web server to be handled.

3. **queues.properties** - This lists the OSE queues that the Web server plug-in can use to send requests to WebSphere, and the IP port, hostname (if this is a remote connection), and clone number to which the request will be forwarded.

We diagram these relationships in Figure 49.

![Parsing a URL to send it to WebSphere](image.png)

Figure 49. Parsing a URL to decide whether to send it to WebSphere

If you use OSE Remote and manually copy the properties files to your Domino server, the `queues.properties` file must be edited if the plug-in is to
access WebSphere from a network-connected computer rather than the local system that WebSphere is installed on. For the servlet redirection techniques or automatic OSE configuration, this file is generated automatically. The number of queues listed will vary with the number of application servers, clones, and servlet engines defined in the WebSphere administrative domain. Our example is simplified and shows only one queue.

3.7.5.1 Overall data flow using OSE remote
The overall flow for OSE remote, from browser request to ultimate response, is shown in Figure 50.
The steps in the diagram are:

1. The HTTP request is sent from the browser to the Domino HTTP server. The request consists of a host name part and a URI.

2. The request is unconditionally sent to the DSAPI plug-in to be checked regarding whether it should be sent to WebSphere for processing.

3. The host part of the browser URL request is scanned against the list of acceptable host names in the vhosts.properties file. If it is found, the associated virtual host is determined.

4. The virtual host from step 3, together with the URI (everything after the host name) part of the request, is matched against the list of acceptable virtual host - URI combinations in the rules.properties file. If found, the OSE queue to send the request to is determined.

5. The OSE queue from step 4 is used to find the IP port number, host name, and clone identity to send the WebSphere request to. This is done by searching the queues.properties file to find a matching set of definitions.

6. The response has now been parsed; if it has been successfully assigned to a destination queue, it is sent to WebSphere. If not, the DSAPI plugin returns to the Domino HTTP server with a 0 return code for the Domino HTTP task to process as described in step 10.

7. The request is sent to WebSphere on the OSE queue found instep 5.

8. The response is received from WebSphere.

9. The response data from WebSphere (using a callback mechanism) is sent to the HTTP server to be sent to the browser.

10. The DSAPI plug-in returns to the HTTP server with a return code indicating the status of the request:
    - 0: The request was not handled by the plug-in, and should be processed by the HTTP server.
    - 1: The request was handled by the plug-in (and thus WebSphere); no further processing is required.

11. The response (whether from WebSphere or the Domino system) is returned to the browser.

If any of the tests against the properties files in steps 3, 4, or 5 fail, the request is returned to the Domino HTTP server (DSAPI return code: 0) for processing and not sent to WebSphere.
3.7.5.2 Overall data flow using thin servlet redirector
The overall data flow for thin servlet redirector is shown in Figure 51.

The flows are very similar to the ones for OSE remote except that there are extra steps to move the request to and from the servlet redirector on the computer with Domino and the DSAPI plug-in. The steps in the diagram are:

1. The HTTP request is sent from the browser to the Domino HTTP server. The request consists of a host name part and a URI.
2. The request is unconditionally sent to the DSAPI plug-in to be checked regarding whether it should be sent to WebSphere for processing.
3. The host part of the browser request is scanned against the list of acceptable host names in the vhosts.properties file. If it is found, the associated virtual host is determined.

4. The virtual host from step 3, together with the URI part of the request, is matched against the list of acceptable virtual host - URI combinations in the rules.properties file. If found, the OSE queue to send the request to is determined.

5. The OSE queue from step 4 is used to find the local OSE queue to send the WebSphere request to. This is done by searching the queues.properties file to find a matching set of definitions.

6. The response has now been parsed; if it has been successfully assigned to a destination queue, it is sent to the thin servlet redirector to be sent to WebSphere. If not, the DSAPI plugin returns to the Domino HTTP server with a 0 return code, as described in step 10.

7. The request is sent (using local OSE) to the thin servlet redirector over the queue found in step 5.

8. The thin servlet redirector, acting as an EJB client of the the RemoteSRP bean in the destination application server, sends the request to the RemoteSRP bean using RMI/IIOP. The bean then sends it to the servlet engine in the application server.

9. The response is received from the WebSphere application server by the thin servlet redirector.

10. The thin servlet redirector returns the response, in turn, to the DSAPI plug-in using (local) OSE.

11. The response data from WebSphere is sent to the HTTP server (using a callback mechanism) to be sent to the browser.

12. The DSAPI plug-in returns to the HTTP server with a return code indicating the status of the request:
   - 0: The request was not handled by the plug-in, and therefore should be processed by the HTTP server.
   - 1: The request was handled by the plug-in (and therefore WebSphere); no further processing is required.

13. The response (whether from WebSphere or the Domino system) is returned to the browser.

If any of the tests against the properties files in steps 3, 4, or 5 fail, the request is returned to the Domino HTTP server (DSAPI return code: 0) for processing and not sent to WebSphere.
If you want to use thin servlet redirector see Appendix A, “Configuration of thin servlet redirector for Domino” on page 449.

Although our examples show connection between a single Domino server and WebSphere application server, the topologies we have described would also accommodate multiple WebSphere servers provided they were all in the same WebSphere administrative domain. In principle, there is no reason why manual configuration of OSE remote transport (or multiple thin servlet redirectors) would not also allow directing requests to multiple WebSphere administrative domains, provided the naming structure of the WebSphere resources was unambiguous. The drawing in Figure 52 illustrates this concept.

![Diagram](figure_52.png)

*Figure 52. Domino with multiple WebSphere application servers*

The Domino HTTP server is mapped to multiple virtual hosts (*Default_HostW1*, *Default_HostW2*, *Default_HostW3*). WebSphere will manage the namespace in Domain 1 so that the virtual host names are
unique; you must ensure that they are unique between the two WebSphere domains if you wish to have a single Domino server send requests as shown.

For a complete discussion of possible topologies, see the IBM Redbook *WebSphere Scalability: WLM and Clustering Using WebSphere Application Server Advanced Edition*, SG24-6153. We used the discussion in Chapters 3 and 8 to guide us through the configuration of the Domino plug-in to access WebSphere using OSE remote and the thin servlet redirector.

### 3.7.6 Installation of WebSphere components on the Domino server

This step must be done for all connectivity techniques, whether you want to connect via OSE remote or via a servlet redirector.

Install WebSphere V3.5 on the same computer as Domino, following the same instructions as previously described in 3.5, “Installing WebSphere Application Server V3.5” on page 53, but with the following changes:

- **Do not** install all components; instead select **Custom Installation** on the **Installation Options** panel.
- On the Choose Application Server Components panel, select only the following components:
  - Application and Administrative Server
  - Web Server Plugins
  - IBM JDK 1.2.2. These options are illustrated in Figure 53 on page 90.
  - If you were using a thick servlet redirector (discussed in 3.7.2, “Servlet redirectors” on page 78), you also need to select Administrator’s Console.
Click **Next**.

- The next panel will now allow selection of the Domino server plug-in since WebSphere installation detected the presence of Domino on this computer, as shown in Figure 54. Select “Lotus Domino V5.0 or higher” and click **Next** to proceed to the next panel.

You will be prompted to enter a user ID with administrative rights on the Domino server. You can enter the administration ID you used to log into the Windows 2000 server; the rights of this ID are not important since we
will never start the WebSphere server on this computer (for the case of OSE remote or thin servlet redirector; we would need a user ID for the thick servlet redirector cases). Instead, we will execute components of WebSphere as loaded by the Domino HTTP server plug-in and, for the thin servlet redirector, a stand alone JVM. Click **Next** to continue.

- You will be prompted for an installation directory for WebSphere (but not the IBM HTTP server, since you should not have specified this earlier). Accept the default (c:\WebSphere\AppServer) or change the drive to one you prefer and click **Next**.

- Since DB2 is not installed on this computer, you will be presented with a Database Options panel prefilled with “InstantDB” settings. Accept these defaults and click **Next**.

- You are now prompted to specify the Program Directory to be used to access WebSphere V3.5. Accept the default of “Application Server V3.5” or specify a string you prefer and click **Next**.

- The code for the components you selected will now be copied to the computer. When installation is complete, dismiss the completion dialog offering to let you read the README file and click **Finish**. Select the option to restart your computer and click **OK**.

- Once the computer has restarted, you can install WebSphere FixPak 3, basically as described in 3.5.2, “Applying WebSphere V3.5 FixPak 3” on page 64, with the difference that you should reply **No** to upgrade examples and IBM HTTP server. If following the installation messages in the command prompt window, you may notice two error messages about files that do not exist. When we checked the names of the files not found we determined that these files are used by the administrative console, which we did not install on our Domino computer, but apparently the fixpak is trying to upgrade it anyway.

- Finally, obtain any relevant e-fixes from the IBM site you obtained the fixpak from and install them as described in their readme files. We did not need to install any e-fixes for FixPak 3. (However, if you are using FixPak 2 you also need to apply e-fix PQ45555 to make SSO between Domino and WebSphere work.)

This completes the installation of WebSphere components for Domino.

### 3.7.7 Configuring Domino to use the WebSphere plug-in

Start the Domino server and the Domino administration client and edit the server document for the Domino server. Navigate to the Internet Protocols tab and select the HTTP tab in that section. Enter the exact path to the
Domino and WebSphere Together

domino5.dll DSAPI plug-in installed during the previous step. In our case, this was C:\WebSphere\AppServer\bin\domino5.dll as shown in Figure 55 on page 92. You should not copy the domino5.dll since this will prevent it from being upgraded if you apply a newer WebSphere fixpak.

Figure 55. Specifying the Domino DSAPI plug-in domino5.dll to access WebSphere

Once this is done, stop and restart the Domino HTTP task by entering:

    tell http quit

followed by:

    load http

or:

    tell http restart

In either case, you should see the message WebSphere DSAPI filter loaded on the Domino console when the HTTP server starts and reads the changed configuration information from the Domino directory.

For the moment, stop the Domino HTTP server again while we complete the configuration to allow it to access the WebSphere server on a remote computer.

Do not attempt to access WebSphere yet from a Web browser accessing the Domino HTTP server; further configuration is necessary to allow the Domino HTTP DSAPI plug-in to find the WebSphere server. This is described in 3.7.8, “Configuration of OSE remote transport for Domino” on page 93 or Appendix
A., “Configuration of thin servlet redirector for Domino” on page 449 depending on which connection type you want to use.

This completes the configuration of Domino to use the WebSphere DSAPI component. Note that if you have other DSAPI plug-ins for your environment, you can specify them in the entry in the server document we just used, separated by commas from the entry we just made. The DSAPI exits will be called in the order they are listed in the server document.

We now continue our configuration setup on the WebSphere computer by describing how to configure OSE remote transport from the Domino machine to WebSphere.

If you have requirements like being able to encrypt the transport between Domino and WebSphere using SSL you need to install a servlet redirector instead. To see how do this see Appendix A, “Configuration of thin servlet redirector for Domino” on page 449 and then proceed with 3.7.9, “Enabling tracing in the Domino WebSphere DSAPI plug-in” on page 102.

### 3.7.8 Configuration of OSE remote transport for Domino

This step is done on the WebSphere computer, that is, the machine you have your WebSphere administrative server (and administrative console) installed on.

There are two steps that must be done:

1. Ensure that the servlet engine on the WebSphere application server is using TCP/IP sockets rather than local pipes to listen for requests.

2. Ensure that the three properties files (vhosts, rules, and queues) created by the WebSphere server are accessible to the DSAPI plug-in. You can do this manually or through an automatic configuration. If you do this manually, you will copy the files from the WebSphere computer to the Domino computer and edit the queues file by adding a line to point to the WebSphere computer. If you choose automatic configuration, the files will be created on the Domino computer by retrieving the configuration information from the WebSphere administrative server. We describe both methods.

Ensure that you have performed the steps in 3.7.6, “Installation of WebSphere components on the Domino server” on page 89 and 3.7.7, “Configuring Domino to use the WebSphere plug-in” on page 91 before beginning.
3.7.8.1 Configuring WebSphere OSE transport to use INET sockets

You must do this step before proceeding or the configurations will not support OSE remote transport.

Ensure that the WebSphere server is running and start the administrative console. Expand the tree under the local node (Thor-2000 in our example), and locate Default Servlet Engine under Default Server and highlight it. Select the Advanced tab on the pane on the right side as shown in Figure 56 on page 94.

![Figure 56. Setting transport settings for WebSphere](image)

Ensure that OSE is set for the the Queue type (this is the default) and click the Settings button. The dialog box “Edit Servlet Engine Transport” will be displayed as shown in Figure 57 on page 95. Select INET Sockets from the drop-down list in the Transport Type field and press OK. You will be returned to the previous panel within the administrative console.
Click the **Apply** button to commit the changes you have just made.

WebSphere will now use TCPIP sockets to receive OSE requests. Note that this does not mean that the IBM HTTP server also installed on this computer cannot communicate with the WebSphere server; its plug-in files were also reconfigured by this task.

You are now ready to use (or recreate) the configuration files created in the WebSphere\AppServer\temp directory to configure the Domino HTTP server plug-in. Close or minimize the WebSphere administrative console.

### 3.7.8.2 Manual update of properties files for the HTTP plug-in

Manual configuration is convenient for testing and also allows you to modify the contents of the properties files if you wish. For a production computer, automatic configuration may be more appropriate.

Locate the three properties files in the WebSphere\AppServer\temp directory on the WebSphere computer (Thor-2000 in our example). These are:

1. vhosts.properties
2. rules.properties
3. queues.properties

Copy these files to the corresponding directory on the Domino server (that is, \\WebSphere\\AppServer\\temp) and edit the queues.properties file with a text editor such as Notepad or WordPad. We need to add one line to the queues.properties file to allow the plug-in to locate the host running WebSphere. This is shown in Figure 58.

We added the line in bold face to point to the Thor-2000 host. The form of the entry is:

```
ose.srvgrp.<queue>.<clone>.host=system
```

In this example the queue is ibmoselink, the clone is clone1 and the system is Thor-2000. Our entry was:

```
ose.srvgrp.ibmoselink.clone1.host=Thor-2000
```

We could have used the full DNS name of Thor-2000.lotus.com or its IP address; this entry is used to locate the WebSphere server using TCP/IP and not the WebSphere node name if this is different from the TCP/IP host name. Be very careful to ensure that this entry is consistent with the lines created by WebSphere since there is little diagnostic information if it is wrong. Save the queues.properties file.
The files are now ready for use.

3.7.8.3 Automatic generation of plug-in properties files

You would do this if you wanted to automate the configuration process. For example, you might want to have the properties files generated automatically every time the Domino computer restarted or you might simply not want to transfer files manually (often across a firewall in production) and modify them manually, however simple the process.

The configuration is done with a modified script (batch) file supplied with WebSphere V3.5. You need to edit the file to ensure that it points to your WebSphere administrative server so it can retrieve the configuration information using IIOP.

If you have a firewall between your Domino and WebSphere computers, ensure that the administrative server is using a fixed port number to listen for administrative requests using IIOP. It is preferable to stop your WebSphere administrative server before proceeding; it will have to be stopped and started in any case to recognize this change. Configure the administrative server listener port by editing the admin.config file located in the \WebSphere\AppServer\bin directory on your WebSphere computer. Open the file in a text editor such as Notepad or Wordpad and locate the line beginning

   com.ibm.ejs.sm.util.process.Nanny.adminServerJvmArgs=

There will already be several arguments specified. Do not change these. At the end of the line add the string

   -Dcom.ibm.CORBA.ListenerPort=33000

Queues refresh interval!

If you change the Servlet Engine Transport Type in WebSphere and immediately copy the queues property file, it may not have been updated by WebSphere. Wait until the time stamp or the content of the file reflects your changes before copying it to the Domino server. If you do not do this, you will have to edit queue type to remote and ensure the port number is the same as the one entered in WebSphere.

The refresh interval for queues properties appears to be about 5 minutes.

You can immediately refresh the rules and vhosts properties files by clicking the Regen Plugin Co... button on the Servlet Engine Panel.
The actual port number you assign is arbitrary (in the range 1024 to 64000), but it must not conflict with any other port numbers in use on the WebSphere computer host. Save the file with your changes.

You must stop (if you did not already do this) and restart the WebSphere administrative server for this change to become effective.

---

**Admin node name**

By default, WebSphere will define its node name to be the same as the hostname of the computer it installed on. You can change this by specifying the argument `com.ibm.ejs.sm.adminServer.nodeName=<your new host name>` in the `admin.config` file for your WebSphere Administrative Server.

If you do this, all references to "adminNodeName" in the batch files described in the following sections refer to the **node name you defined and not** the WebSphere server computer's TCP/IP host name. If you do not make this change, the Java executables will not be able to find your WebSphere installation. This also applies to `nameServiceNodeName`.

This only applies to the Java command line arguments; the definition of host name in the `queues.property` file is the TCP/IP host name since this is used to find the actual computer over a TCP/IP link.

Open the file `OSERemoteConfig.bat` in the `\WebSphere\AppServer\bin` directory on your Domino computer using a text editor. Locate the last line of the file and add the following arguments, all on the same line:

- **-adminNodeName**
  
  Set this to your WebSphere server’s host name. In our case we used Thor-2000.

- **-nameServiceNodeName**
  
  Set this to your WebSphere server’s host name. In our case we used Thor-2000.

If you wish to capture the results of the configuration process for possible problem determination, add the following arguments:

- **-traceString**
  
  Set this to `com.ibm.servlet.engine.*=all=enabled` to capture all messages during the configuration.

- **-traceFile**
  
  We set this to `%WAS_HOME%/logs/configRegen.log` so that we could
easily identify the configuration log. (%WAS_HOME% is an environment variable set by the batch file to point to the WebSphere installation directory, usually c:\WebSphere\AppServer).

The edited file should look like Figure 59. Note that we have inserted line breaks in the Java command line for clarity. You must ensure that all of the arguments are on the same line as the Java command to run the file.

![OSERemoteConfig - WordPad](image)

Figure 59. OSE remote automatic configuration batch file

Save the file. It is now ready for use. Ensure that the WebSphere server is running and execute the batch file. Once it has run, you can view the properties files generated in the \WebSphere\AppServer\temp directory to see the results. Note that there are timestamps in the file header to allow you to confirm that these files have just been generated. If the files are not generated (or generated incorrectly) you can view the configuration log to diagnose the problem. Apart from the host line in the queues.properties files (and the file time stamps), these files should be identical to those on the WebSphere administrative server computer.

If you have enabled WebSphere (Global) security (we describe how to do this in 4.4.3, “Configuring WebSphere V3.5 global security for single sign-on” on page 127), you will get a login prompt when you run the batch file since it contacts the WebSphere administrative server to retrieve the configuration.
information. This obviously defeats the purpose of automatic or unattended configuration generation. To automate the authentication, you need to edit the file sas.client.props found in the \WebSphere\AppServer\properties directory (on your Domino computer).

Change the lines beginning with:

- `com.ibm.CORBA.loginSource=prompt` to `com.ibm.CORBA.loginSource=environment`
- `com.ibm.CORBA.loginUserid=` and add the user ID you configured with WebSphere Global security. This is the same user ID you would use for the login prompt.
- `com.ibm.CORBA.loginPassword=` and add the password you specified with the user ID above.

Do not alter the rest of the file. Once you are satisfied with your entries, save the file.

The portion of the file we edited is shown in Figure 60.

![Figure 60. The sas.client.props file edited to eliminate login prompt for thin servlet redirector](image)

Once these changes are made, you will not get an authentication prompt when the batch file runs. You should protect the sas.clients.props file appropriately because it contains a user ID and password used to sign onto the WebSphere administrative server.
3.7.8.4 Testing OSE remote from Domino to WebSphere

Start the Domino server (including the HTTP server task) and the WebSphere server. Note that you may receive a warning message on the Domino Console from the DSAPI plug-in of the form:

```
ws_init_ip_addr - gethostbyname : - 10093
```

Our investigations showed that this is a communications initialization warning message and that the server will automatically reconnect in all cases.

Point a Web browser at your Domino server (in our case odin.lotus.com) and request a WebSphere resource. We chose /webapp/examples/showCfg for our test.

You should see a panel resembling the one in Figure 61.

--- Login ID format must be consistent! ---

Our example assumes that you have not altered the field User Filter in the LDAP Advanced Properties panel in the WebSphere Global Security Wizard. Thus we show the use of `shortname` or `uid` in this field. If you alter the format acceptable to WebSphere as described in Appendix C, “Using WebSphere advanced LDAP properties” on page 463, you must also change the format of this entry in sas.client.props for the automated login to work properly.
Note that the hostname (circled in the diagram) is the hostname of the WebSphere server, not the Domino server running the HTTP task (shown in the URL line circled in the browser).

Note that the properties files generated (or copied) in this step will need to be recreated if any of the following changes to the WebSphere server are made:

- Adding or removing a URL (Web resource)
- Securing or unsecuring a URI
- Adding or removing a host alias
- Changes to the queue properties of a servlet engine (name and/or port)
- Adding or removing a servlet engine
- Adding or removing a server clone

Again, ensure that the queues.properties file is updated before copying it to your Domino server after any changes that would change its content. If you use automatic configuration, you do not need to wait.

This completes the configuration and testing of Domino to access WebSphere via OSE remote transport.

3.7.9 Enabling tracing in the Domino WebSphere DSAPI plug-in

By default, very little information is captured during the running of the plug-in. You may wish to log more information to understand the underlying processing or to diagnose error situations.

Stop the Domino HTTP server by issuing the console command:

```
tell http quit
```

Locate the bootstrap.properties file in the WebSphere properties directory. By default this will be `WebSphere\AppServer\properties` on the drive you installed the WebSphere executables. Open this file in a text editor and scroll to the line beginning `ose.native.log.level=`. The installation version of this line will read:

```
ose.native.log.level=ERROR|WARNING
```

Change this line to:

```
ose.native.log.level=ERROR|WARNING|TRACE|INFORM
```

as shown in the highlighted line in Figure 62 on page 103.
There is one trace log created each time the HTTP server (and DSAPI plug-in) is loaded. The names of the logs are (for the Domino HTTP server) of the form:

```
trace.log.domino.date,time and year
```

For example, the trace log displayed in Figure 63 on page 104 was named:

```
trace.log.domino.Tue-Feb-20-10.52.30-2001
```

Once the bootstrap.properties file has been edited you can restart the Domino HTTP server; it will immediately start to create detailed trace logs. These may be helpful in determining why URLs are processed in unexpected ways. An example of a portion of the log entries generated for an access to the showCfg servlet is shown in Figure 63; note that we have scrolled the display to the right. The left side of the screen contains timestamp and other detail information. The actual trace for this request takes two screens to display! We highlighted the line where the request was parsed by the rules file. This confirmed that the request was intended for WebSphere. If a request is parsed against the three properties files (vhost, rules, queues) to be handled by WebSphere, the plug-in will send it to WebSphere and pass the WebSphere response back to the Domino HTTP server to be returned to the Web browser. It then returns to the HTTP server with a return code set to 1; this tells the Domino HTTP server that the request has been handled.
If the HTTP request cannot be parsed against the properties files to find a destination WebSphere server, it will be returned to the Domino HTTP server for processing with a return code of 0, indicating that the request was not processed in the plug-in (and should be handled in the HTTP server). An example of such a request would be a URL pointing to a Domino database or an HTML file in the file system. Note that any requests intended for WebSphere but entered incorrectly will also be returned to Domino and likely generate an error return. For example, if you spell a servlet’s name incorrectly, you will likely get a 404 - Not Found error from Domino.

Note that you cannot view the trace log while the plug-in is running. If you need to view the trace for an event, stop and restart the Domino HTTP task.

Figure 63. Domino plug-in detailed trace log

### 3.8 Installing Domino and WebSphere on the same computer

Simply follow the installation steps described in 3.5, “Installing WebSphere Application Server V3.5” on page 53 and 3.6, “Installing and configuring Domino V5.0.6a” on page 66, but install Domino on the same server used by WebSphere server; then skip to 3.7.7, “Configuring Domino to use the WebSphere plug-in” on page 91. There is no need to configure the properties
files as described in 3.7.8.3, “Automatic generation of plug-in properties files” on page 97 since the Domino plug-in will access the WebSphere properties files in the WebSphere installation directory. By default—and there is no reason to change this—it will be configured to use OSE local transport. You would only adjust the servlet engine’s transport mechanism from local pipes to INET sockets if there was another HTTP server on a different computer sending work to WebSphere. If this is necessary, we show this in Figure 57 on page 95.

You should ensure that the IBM HTTP server and/or the Microsoft Internet Information Server do not start automatically in the Windows 2000 Services Control panel; otherwise the Domino HTTP server will not be able to bind to port 80. Instead, it will exit with a message:

HTTP Socket Bind Error, hostname/ip <hostname>
HTTP Server: Could not bind port 80. Port may be in use

even though it loaded the Web SSO configuration and DSAPI plug-in.

If, for some reason, you do wish to have both Domino and the IBM HTTP server (or another Web server) both run on the same computer, you can do this by changing the port number (for example, to 8080) that the Domino Server uses for its HTTP server in the Domino Directory (Server document, Ports, Internet Ports, Web, TCP/IP Port Number). If you also want both HTTP servers to access the WebSphere application server you also have to add a host entry under the default_host, as described in 3.5.1.1, “Adding aliases to the WebSphere default_host” on page 62, specifying the new port number to be used (8080). If you do not do this, the plug-in will not be able to resolve the host entry and will reject the request. The error message would be returned by the Domino server in this case (usually 404 - Not Found). If you have other virtual hosts defined that you want to access, you would also have to do the same for them.

Again, you can increase the detail traced in the WebSphere plug-in log by editing the bootstrap.properties file as described in 3.7.9, “Enabling tracing in the Domino WebSphere DSAPI plug-in” on page 102. This will also increase the tracing level for any other HTTP plug-ins on the same computer; for example, if you also run the IBM HTTP server its trace level would also be more detailed.

This configuration is convenient for testing in a testing environment that does not require separate computers. We used both this configuration and one with separate computers to test SSO in our environment. However, one would typically not install both Domino and WebSphere on the same computer in a production environment.
3.9 Adding the Domino ncsow.jar file to the WebSphere classpath

We tested our implementation of single sign-on (SSO) using a servlet that accesses the Domino classes in order to read the Domino Directory. We needed to add the ncsow.jar file to the WebSphere classpath to enable this.

Ncsow.jar contains the classes required to access the Domino Object Model over IIOP. It is a WebSphere-specific variation of the default file (ncso.jar) for remote access to the Domino object model. The difference is that ncsow.jar does not contain any object request broker (ORB) because WebSphere already has its own ORB.

In our configuration, we copied the ncsow.jar file from the computer we installed Domino on to the computer where we installed WebSphere. By default, this file is located in the Java subdirectory of the Data\Domino directory in your Notes or Domino installation. In our case, it was located in:

C:\Domino\Data\Domino\java\ncsow.jar

We chose to copy the jar file to the \WebSphere\AppServer\lib directory, but it can be copied anywhere in the file system on the computer you have WebSphere installed on. Preferably, this should be on a drive local to the WebSphere computer for performance. If you install Domino and WebSphere on the same computer, it is not necessary to copy it to a WebSphere directory.

In either case, you need to edit the admin.config file in the \WebSphere\AppServer\bin directory to add the exact path to the line beginning:

    com.ibm.ejs.sm.adminserver.classpath=

By default, this will be the third (uncommented) line in the file after installation. Add the path and file name of the ncsow.jar file to the end of the classpath. Note that the convention in this file uses forward slashes between directories. Thus the entry for the ncsow.jar file in our case looked like:

    ...;C:\WebSphere/AppServer/lib/ncsow.jar

You may find it easiest to copy and paste an existing entry and then simply change the name of the jar file rather than typing the entire string. Once you are satisfied with your entry, save the admin.config file. You will need to stop and restart the WebSphere administrative server for this change to be recognized and take effect. You can stop the WebSphere administrative server from the console by right-clicking on the node representing your WebSphere computer and selecting any of stop, restart or stop for restart.
If your WebSphere console is running on the same computer as the WebSphere administrative server, you will receive a warning that the console will be stopped as well. Once the WebSphere Administrative Server has restarted, you can start the console again.

3.10 Installing Domino, WebSphere and DB2 in a three-tier environment

In many cases, you will wish to place your Web application server in a protected environment behind your DMZ (“demilitarized zone”) and also place your database servers behind a further level of protection. “Three tier” in this environment means:

1. **DMZ** - Place Domino presentation servers and the Domino server running the HTTP task, plus the necessary plug-in and connectivity software (OSE remote or servlet redirector), in this layer.

2. **Application servers** - Place WebSphere Application server (or servers) and any Domino servers used for managing content but not directly accessed by clients in this layer.

3. **Database servers** - This layer is where you would locate your database servers with high levels of protection and hardening for reliability and security.

One possible way to configure Domino and WebSphere in a three-tier environment is shown in Figure 64 on page 108.
In this configuration, we need to have two ports opened up between Domino and WebSphere; one of these is used for remote configuration and could be closed if one manually configures the properties files on the Domino server.

3.11 Summary

In this chapter we have covered the steps necessary to install Domino R5 and WebSphere V3.5 on different Window NT machines. We have also discussed how to verify correct installation.
Chapter 4. WebSphere - Domino security and single sign-on

In this chapter we describe how to set up and test single sign-on (SSO) between Domino and WebSphere. However, we need to give some background first to ensure that the basic concepts are clear. If you are already comfortable with SSO and security as implemented in WebSphere and Domino, you may wish to turn directly to our SSO example in 4.3, “Example application” on page 120.

4.1 What single sign-on is

Single sign-on means the ability to move between applications without being prompted for a userid and password (or certificate) when moving from one application to another. The applications could be on the same or different physical servers. Our discussion of SSO implies only that a user will not be prompted for authentication credentials more than once during a session.

Single sign-on is a description of a user experience; there are multiple technical approaches to ensure the experience meets the user's expectation. Among the major components in an SSO context are:

- Single authentication directory: Usually an LDAP server, this keeps a central record of a user's credentials (userid/password, distinguished name and other attributes) which can be used to validate the user's identity to authenticate them. Generally, it is assumed that the user will have a single entry in the directory and therefore a single userid/password or certificate, but this is not absolutely necessary. This is desirable to ensure that a user has the same identity (user ID) and authorization credentials (password or certificate) in all applications. However, simply using a single authentication directory does not in itself provide SSO; a user visiting multiple applications could receive authentication challenges from all of them. The advantage of a central directory is that the user should not have to remember multiple sets of user IDs and passwords. Naturally, it is also possible to simply store duplicate information in multiple directories, but it is often difficult synchronize these.

- Persistent authentication: This is often misunderstood to be an example of single sign-on. For example, when a Web browser user is challenged by a Web server 401 return code (access denied to resource), the Web browser will present a dialog box to enter a user ID and password rather than display the 'forbidden access' message. Once the user enters their user ID and password, this is presented to the Web server; if accepted, the browser will automatically present it to the same server and all servers in...
the same realm until the browser session ends. A realm is generally the host name and file system directory accessed, but the Web server can define an arbitrary string to group servers into an extended realm. Although this may superficially appear to be a form of SSO, the user is actually authenticated again for every server access. The limited definition of "realms" (groups of servers or directories within a real or virtual server), plus the fact that the user ID and password are sent base64-encoded but unencrypted, makes this technique difficult to scale and manage. (The communications can, of course, be encrypted by using SSL V2). However, the very persistence of the authentication may be a problem in some environments since the browser will cache the user ID and password until the browser is shut down. There is no concept of a “time-out” and another user could use an authenticated user’s credentials if the first user did not shut down the browser after use. As well, the user cannot force a logout from a site; instead they have to remember to shut down their browser.

Similarly, if a client has a certificate (usually, an x.509v3 SSL certificate) which is acceptable to all servers they access, there will be the illusion of SSO, but the certificate will be reauthenticated if a user’s session ends and is later re-started. This would occur if a user established an SSL session with a server, then visited a second server and finally returned to the first server. It does have the advantage that authentication is in any case transparent to the user (except that initial access to the certificate will likely be password protected by the browser, and the browser may prompt to select a certificate to use each time the server challenges). Notes access to Domino servers follows this model; the user is only prompted for a password to open their Notes ID file; their certificate(s) are presented for authentication without user prompting to each server the user accesses.

Note that persistent authentication does not imply any state information is retained between successive visits to the same site, even in the same browser session.

• Persistent authentication sessions: It is also possible for the first server the user contacts to perform authentication against the user’s security credentials and then create an “authentication token.” Generally, this is a “cookie” stored and presented by the user’s Web browser. When the user again contacts the server or any other server that “trusts” it, the token is presented as proof of authentication. The second server can either accept the authentication or reauthenticate, based on the contents of the token. Generally, the token is issued with a limited lifetime so that its validity will expire after a period of time, either of inactivity or simply from its creation.

The token often also carries state information or a pointer to state information stored by the application. State information could include the
user's last location, contents of a shopping cart, application selections, and the like.

This is the type of SSO we will be illustrating in this section. The trust relationship will exist between Domino and (multiple) WebSphere servers. Trust in this case is assured by encrypting the authentication credentials in the cookie using a “shared secret”: a common (LTPA) key. Only servers with the shared LTPA key can participate in the relationship. (LTPA is the WebSphere abbreviation of Lightweight Third-Party Authentication [LDAP]). In addition, the token is created with a limited lifetime, so it will expire in a fixed period after creation. No state information will be stored in the cookie.

- Reverse proxy or access management applications: These are servers which intercept user requests and pass them on to an application server, retrieving and sending any necessary authentication information on the user's behalf. This technique allows coexistence of applications with inconsistent authentication and state management implementations. We will not illustrate this technique in our examples.

4.2 Security concepts

To illustrate SSO, it is necessary to ensure that the resources the user accesses are protected so that they will receive an authentication challenge. For our example, both Domino and WebSphere resources must be protected so that unauthenticated access is not possible. Thus, at a minimum, it is necessary to ensure that security is implemented in the application or its environment and that the resources are access controlled.

When access to a secure resource is attempted, the user’s authentication credentials are compared to either an access control list or the user’s access control rights are evaluated. If a user has not yet authenticated, obviously their access control rights can only be evaluated against a general profile such as “anonymous”; if an anonymous user does not have adequate access rights, the user will be presented with an authentication challenge.

Note that the authentication mechanisms on the platforms need not be the same; all that is necessary is that each platform trusts the other to authenticate. For our example, we will implement Domino session-based authentication using a custom login prompt; for WebSphere, we will use browser basic authentication. However, both Domino and WebSphere will create a cookie acceptable to the other, so the user will only see one authentication challenge (which one depends on whether the user first accesses Domino or WebSphere) in a session.
Once the user’s identity has been established, their access rights can be evaluated using the security schema implemented in the application runtime environment. The Domino access control model and WebSphere model have rather different implementations which we must explain before implementing security to demonstrate SSO.

4.2.1 Domino security

Domino security is integrated into the product and is always active. The implementation of access control is hierarchical:

- **Server access** (for Web browsers, this is implemented by simply not allowing “Anonymous” access to any database on a Domino server and prohibiting browsing the Domino file system from a browser). For a Notes client, there are two server access lists: “Deny Access,” which lists users or groups who will explicitly be denied access to the server and “Allow Access” which lists groups and users allowed access (provided they are not on the “Deny Access” list).

- **Database access** - This is implemented by an Access Control List (ACL) in the database. The ACL lists users (and groups containing users) and their access rights on a scale of None, Deppositor, Reader, Author, Editor, Designer and Manager. For those levels allowing writing or alteration of database documents (records), users can also be independently controlled from creating or deleting documents. Generally, Web users would have Reader or Author access; the higher levels are applicable to a Notes client. These levels can be further refined by the use of roles. Roles allow creating of subsets of users in the ACL to be granted access rights to the database or individual documents (records) in the database, as well as application defined rights.

- **Document access** - A secondary ACL within a document (a Readers or Authors field) allows refinement of the database access. In other words, it is possible to further restrict access to documents within a database, but it is not possible to increase a user’s access rights in this way. Thus, if a user is in the “Authors” field for a document (implying the right to change the document’s contents), but is only a “Reader” in the database ACL, they will not be granted update access to the document.

- **Section level access** - It is possible to prevent a user from opening and viewing a section of a document. For Notes clients this is not a security feature since they have other methods to view the hidden fields in the section; however, for a browser this is effective since the contents of a section a user does not have access to are not sent to their Web browser.
Further levels of access control are available, but these are all that typically are relevant to a Web-based Domino application. For our purposes, we will simply prevent unauthenticated (“Anonymous”) access to Domino databases on our test server, but allow general authenticated access to databases. This is why during Domino installation we set the default access control on all Domino databases to be “None” for unauthenticated users (Chapter 3.6, “Installing and configuring Domino V5.0.6a” on page 66 and Figure 45, “Setting default database access levels during installation” on page 73).

Domino security implementation is resource based; the objects being protected carry access control lists defining the rights of users to access and alter the object. Thus the security model does not depend on context. However, there is no central repository of user rights since these are stored with each protected resource.

### 4.2.2 WebSphere security

WebSphere security must be enabled; by default, when WebSphere is installed it is not active and no resources are protected.

Global security must be enabled for any application security to be implemented. WebSphere global security also defines the default authentication methodology (none, basic, certificate or custom) to be used by applications. As we describe later in this section, applications can override this default authentication methodology with one of the others if necessary. Global security also defines whether or not the server will support single sign-on; finally, the central repository for authentication credentials (that is, an LDAP directory) is defined to support authentication.

The WebSphere security model is capability-based: a user’s rights are defined as a set of application-method pairs. For example, a user may have the right to “Get” (Read) on application A, but not the right to “Put” (Write) for the application. The WebSphere security repository stores and manages this set of permissions for each user.

We have not defined what is meant by an application; WebSphere defines security access control against enterprise applications. An enterprise application is a grouping of one or more of:

- Enterprise JavaBeans
- Web applications
- Virtual host resources (or, more precisely, individual URIs within a virtual host). The definition of web applications typically defines these resources so they need not be separately selected.
Thus an enterprise application can consist of EJBs (but need not contain these), Web applications (and typically would contain at least one) and, implicitly or explicitly, resources within virtual hosts. This concept is illustrated in Figure 65, in which we show an enterprise application containing three EJBs, a Web application with two servlets and a JSP, and three URLs within a virtual host. A fourth URL within the virtual host is not selected to belong to the enterprise application. The URLs selected will be the external mapping to the objects in the Web application that can be accessed from a Web browser or Java client. There may be other resources within the Web application that are not directly accessed by a client, so their URLs would not be selected in the virtual host.

We show some of the mapping between the URLs and objects within the Web application. In our example, we show URL 1 mapping to JSP 1, URL 2 mapping to servlet 1, and URL 3 mapping to servlet 2.

Note that you have to select the URLs within a virtual host you want to secure; you cannot simply add an entire virtual host to an enterprise application by selecting it.
Note that, although Web applications are contained within an enterprise application, they will not be explicitly protected in the following steps. This is because they can only be accessed through virtual host-URL pairs; these are protected as described later in this section. EJBs must be protected if necessary.

Once an enterprise application is defined, its overall security can be defined as a refinement of global security:

- An application can have a different authentication methodology than the default defined in global security settings and can specify a different realm to be returned to a browser. For example, if the initial global security setting is to require basic authentication, an application can override this default authentication method to be certificate-based (with a default to basic authentication), or to use a custom login prompt.

- It is also possible to have the permissions to access EJBs within the application be tested against an application-defined user ID/password combination. If this is not done, the user ID and password of the user accessing the resource (for example, a servlet) who called the EJB will be passed for permission checking.

Actions that can be performed are defined by method groups. Unlike the Domino access control levels (none, depositor, reader, author, editor, designer, manager), these are not defined in a hierarchy of increasing access levels. In WebSphere the default method groups are:

- Read methods
- Write methods
- Remove methods
- Create methods
- Execute methods
- Finder methods

Thus you assign a method to one or more method groups, and then you assign users to the different method groups. We can also call method groups for roles. For example, when we request data from a Web browser we can use the HTTP GET method. By default WebSphere assigns the read role (or method group) to the HTTP GET method. It is then up to you to specify which users should be attached to the read role (or be part of the read method group).
If you accept default assignment of method groups/roles to your methods, WebSphere will assign them based on the rules shown in Table 3.

**Table 3. Rules for assignment of default method groups or roles**

<table>
<thead>
<tr>
<th>If the Web resource method is ...</th>
<th>Or the object method name starts with ...</th>
<th>The required right (method group) is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET or POST</td>
<td>get</td>
<td>READ</td>
</tr>
<tr>
<td>PUT</td>
<td>set</td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td>create</td>
<td>CREATE</td>
</tr>
<tr>
<td></td>
<td>ejbCreate</td>
<td>CREATE</td>
</tr>
<tr>
<td>DELETE</td>
<td>remove</td>
<td>REMOVE</td>
</tr>
<tr>
<td></td>
<td>ejbRemove</td>
<td>REMOVE</td>
</tr>
<tr>
<td></td>
<td>ejbFindByPrimaryKey</td>
<td>FINDER</td>
</tr>
<tr>
<td></td>
<td>(all other)</td>
<td>EXECUTE</td>
</tr>
</tbody>
</table>

Further method groups (roles) can be defined in the WebSphere security application to fit application-specific needs. For example, a banking application might well have method groups/roles such as transfers, request foreign currency, place standing order, and so forth, that do not exactly fit the above definitions. If there is a need to control these specific methods of the application, then additional security method groups can be defined in the security application.

Resource security can be defined for EJBs and URIs within a virtual host; since the URIs map to Web application components (servlets and JSPs), this implicitly provides security for Web applications. This step simply creates the method groups that can be applied to protect a given resource and, by extension, be granted to users of the application containing the resource. The method groups that will be eligible for protection are selected from the standard method groups plus any new ones that have been defined. Note at this point we have defined the security framework for the resources in the application; we still have to map these to users to implement security.
This concept is shown in Figure 66. We show that EJB 1 has read and write methods defined for protection, EJB 2 has read methods, URL 1 has read and execute methods, URL 2 has read and write methods defined, and finally, URL 3 has only read methods defined. This means that user permissions can be defined to map against the method groups defined, but no others.

Once this is done, we have implicitly defined the method groups for the enterprise application, as shown in Figure 67 on page 118. The method groups defined for the application are the set of all unique method groups defined for the individual resources. User permissions are associated with the method groups for the application. Note, however, that the resource protection definitions are still enforced: a user could have “execute” permission for the enterprise application, but would only be able to use this permission for resources with this method defined. In the current example, the only resource with an execute method group defined was URL 1.

One feature of the model is that resources do not have permissions applied to them directly; therefore a given resource could have different permissions applied in separate enterprise applications. This context does not exist in the
Domino security model, where a resource’s protection and permissions are independent of context.

Once these application/method group relationships have been defined, then it is possible to create permissions for users within an enterprise application by assigning them to method group(s) for the application. We illustrate this in Figure 68 on page 119, in which we show two users, Ann Green and Tom Brown. In this example, we granted Ann Green Read, Write and Execute permissions, while Tom Brown was granted Read and Write permissions. Note that, while assigning permissions, only the method groups of the enterprise application are shown; those for the contained resources are mapped from these.
For a much more detailed discussion of WebSphere application server security, see the IBM white paper *IBM WebSphere, Standard and Advanced Edition, V3.5 Security Overview*. This paper is available at the Web site:


### 4.2.3 The shared secret and the LTPA token

The Domino and WebSphere servers that are part of a SSO trust realm that has a shared secret called Lightweight Third-Party Authentication (LTPA) keys. This shared secret allows each server to verify an authentication token (a browser cookie) issued by the first server the user authenticates with. This LTPA token has the following overall structure:

- **Name**: “LtpaToken” (cookie name). As well domain and path is stored in unencrypted format.
- **User Data**: Name/value pairs separated by the “$” delimiter symbol. For example user id would be represented by “u:<userid>”
- **Expiration Date**: The time and date the token will expire. The format is milliseconds since midnight January 1, 1970.
- **Digital Signature**: A digital signature of the User Data and the Expiration date signed with the private key of the authenticating server (this is the private key in the LTPA keys, not the Domino or SSL private key). The signature algorithm is RSA/SHA1. Other servers can verify this signature.
since they have the corresponding public key; there is no certificate in the signature in this scheme.

The content is encrypted using 3DES (triple DES). The shared secret key used for this encryption is stored in the LTPA keyring file (which is shared between Domino and WebSphere). Once the token is issued by the server a user authenticates with first, all servers sharing the same set of LTPA keys will accept the authentication as valid and can apply access control using the user strings in the token.

The LTPA token expires either after a specified time period from issuing or when the browser exits (that is, it is not stored on the client browser computer). You can recognize the cookie by its name: LtpaToken. If a user does not allow cookies in their browser settings, they cannot participate in single sign-on.

4.2.3.1 Security considerations for the LTPA token
The LTPA token expires as soon as the browser is closed, but there is a theoretical possibility that somebody can steal the token from a "live" connection between the server and the client and make a "replay" attack against the server. You can protect yourself against this possibility by using SSL between the client and the server. Another theoretical possibility if somebody has gotten hold of a LTPA token is an off-line attack against it to try determine the key used to encrypt token. The server administrator can generate new LTPA keys regularly as a safeguard against such attacks.

4.3 Example application
To demonstrate single sign-on between WebSphere application server and Domino, we created a servlet that runs in the WebSphere server and accesses Domino objects. For the purpose of our demonstration, we simply read the People view of the Domino Directory and return the results from the servlet.

Since Domino Directory (with the file name names.nsf) is protected from anonymous or unauthenticated access (recall this was described as part of the Domino configuration process described previously), the servlet needs to be authenticated to access it. The servlet obtains the user's authentication credentials from the browser cookie which contains the SSO authentication token. This allows authenticated access to the Domino Directory on behalf of the user.
The servlet uses the host name of the Domino server as a parameter in the URL calling it to locate the Domino Directory to be read. The servlet session request to Domino uses IIOP so that it can communicate with a remote Domino server. Our test configuration had Domino and WebSphere on separate physical computers. However, the illustration would still be valid if Domino and WebSphere were installed on a single computer since each application server enforces its own security.

We show a high level overview of these relationships in Figure 69.

![Diagram](image-url)

Figure 69. Overview of the ReadNames servlet accessing the Domino Directory

We discuss the file diiop_ior.txt in 4.4.2.1, "Use of another HTTP server and the diiop_ior.txt file" on page 126.
The Java source code for the servlet is listed in Appendix B, “The ReadNamesRemote Servlet” on page 459.

### 4.4 Setup of the Domino-WebSphere single sign-on environment

This section describes how to set up the single sign-on environment in Domino and WebSphere. We will also need to create a WebSphere enterprise application; this procedure is described in the following section (4.5, “Setting up WebSphere application security” on page 147). In addition, in practice we would want to enable multiple Domino servers in order to check that our SSO configuration works with all of them.

#### 4.4.1 Creating a user ID for WebSphere administration

Before starting the configuration of the two servers, create a user ID in Domino that will be used to log into the WebSphere advanced administrative console when security is enabled.

We created a Person document for the user “WebSphere Administration” with the User name field set to *WebSphere Administration*, the Short name/UserID field set to *WASAdmin* and an Internet password set as shown in Figure 70.

![Figure 70. WebSphere administrator person document in Domino Directory](image-url)
Note: Make sure you fill out the User name field. If it is empty WebSphere cannot look up the person using LDAP even though you are specifying the short name.

Ensure that the directory is LDAP-enabled by confirming that the LDAP task is running. In addition, you can use the `ldapsearch` utility supplied with Domino to retrieve the entry using the LDAP protocol. In our case we entered the following on a command line:

```
ldapsearch -v -h odin.lotus.com "uid=WASAdmin"
```

The successful response is illustrated in Figure 71. Note that the `ldapsearch` program does not require that Domino be running to execute, but it does need access to the Domino executable programs. We were unable to get it to execute on a computer on which Domino had not been installed.

```
C:\Domino>ldapsearch -w -h odin.lotus.com "uid=WASAdmin"
ldap_open: odin.lotus.com: krb5: krbtgt/LONGASSWORD@odin.lotus.com:50010
Filler pattern: uid=WASAdmin
returning: ALL
*** Filter is: (uid=WASAdmin) ***
CN=WebSphere Administration
cn=WebSphere Administration
shortName=WASAdmin
uid=WASAdmin
mail=WebSphere_Administration@lotus.com
objectClass=top
objectClass=person
objectClass=organizationalPerson
objectClass= inetOrgPerson
objectClass=dominoPerson
givenName=WASAdmin
cn=Administration
1 matches
C:\Domino>
```

Figure 71. LDAP retrieval of WebSphere administrator

This test confirms that the WASAdmin user ID can be retrieved by the LDAP protocol. Note that the `ldapsearch` utility need not run on the same computer as the LDAP server, and it need not run against a Domino LDAP server; it could, for example, run against an IBM Secureway LDAP server.

4.4.2 Configuring the Domino server to support LDAP and IIOP

While you are working with the Domino server, open the Domino Directory and edit the server document for your LDAP Domino Server. Select the Basics tab, and enter the full DNS name of the server in the “Fully qualified Internet host name” field, shown in Figure 72 on page 124 for our configuration. If the computer has more than one DNS name, make certain this entry matches the one which Domino is using in your configuration.
While you have your server document open, select the tabs **Ports, Internet Ports, Directory**. If the field TCP/IP Port Number has been changed to an port other than 389, either reset it to 389 or record the value in use since you will have to configure WebSphere global security to use the designated port. We did not use SSL to connect to the directory in our testing. For the purposes of our testing, we allowed anonymous (read) access to the LDAP server. It is possible to secure the LDAP directory from anonymous reader access; we discuss how to do this (and the implications) in Appendix D, “Securing the LDAP server from anonymous access” on page 471. The settings we used for the testing in this chapter are shown in Figure 73 on page 125.
Finally, since we will be using IIOP to access Domino from WebSphere, we need to allow this access. Select the Security tab in the server document and scroll to the section (in the extreme right bottom corner) titled “Java/COM Restrictions.” For the purposes of our testing we used “*” (all users in directory) to enable access to Restricted Java/JavaScript/COM and only allowed the Administrators group to run unrestricted Java/JavaScript/COM, as shown in Figure 74 on page 126.

---

**Active Directory warning**

Be careful if enabling Microsoft Active Directory (AD) on the Domino machine. The port for the AD LDAP server cannot be reconfigured to anything other than 389. If you enable AD on a Domino machine where Domino Directory is the LDAP server and you have global security enabled for WebSphere, you will lock yourself out of the WebSphere administrative console. This is because the authentication request will go to AD instead of Domino. The problem is not obvious initially because the Domino LDAP service will not report any errors on the console when loading, even though it cannot bind to port 389. Also, it is not enough to create the same user ID in AD because the two directories use different schemas.
Save and close the server document.

**Be sure to enable Java!**

If you forget to enable access to run Java programs as shown in Figure 74, your applications will be loaded and run in WebSphere. However, as soon as they attempt to open an IIOP session to Domino they will fail with the rather cryptic error message number 4488.

To prevent this, ensure that you have enabled access to restricted Java programs (not agents) for the users you wish to give access to the application.

### 4.4.2.1 Use of another HTTP server and the diiop_ior.txt file

When the IIOP task is loaded on the Domino server it writes the *Initial Object Reference* (IOR) for its session object to a file called diiop_ior.txt in the root directory of the Domino HTTP server. Programs that want to connect to Domino via IIOP needs the IOR to do it. In our ReadNamesRemote servlet this also happens “undercover,” in the code when the NotesFactory.createSession method is executed. The diiop_ior.txt file is read from the Domino server using HTTP.

Therefore, even if you do not plan to access Domino using HTTP, a server must be running on the Domino machine that accepts HTTP requests on port 80 and HTTPS (SSL) requests on port 443 (there may be a way to use other ports, but we did not figure out any such way). This also means that if you use an HTTP server other than Domino’s on the Domino machine you must copy the diiop_ior.txt file to that HTTP server’s root directory. If you change the IIOP port or the DNS name of the Domino server the diiop_ior.txt file will be regenerated with new values and you will have to copy it to the other HTTP server’s root directory again.
If, for some reason, you do not want to have an HTTP server running on the Domino machine, you must get the value of the diiop_ior.txt file to the remote machine by other means and then use the NotesFactory.createSessionWithIOR method to connect to the Domino machine. However, we did not test this approach.

We are now ready to configure SSO for WebSphere and then Domino. They must be configured in this order because the keys used to encrypt the SSO token (or browser cookie) must be generated in WebSphere and then imported into Domino.

4.4.3 Configuring WebSphere V3.5 global security for single sign-on

Start the WebSphere advanced administrative console if it is not already started (the WebSphere server must be running first). Ensure that the Domino server is running with the LDAP server task active.

Select Console -> Tasks -> Configure Global Security Settings from the menu. In the first panel displayed, select “Enable Security” as shown in Figure 75. You can leave the Security Cache Timeout set to the default of 600 seconds.

![Figure 75. Enabling security in WebSphere](image)

Click Next.
On the next panel enter your DNS domain in the Realm field at the top of the panel. For our example, this was lotus.com. This must be done because the SSO token (or browser cookie) can only be shared within a single specified DNS domain. This panel is shown in Figure 76.

![Figure 76. Specifying DNS domain for SSO](image_url)

Leave the Challenge Type set to the default of Basic and do not check the entry "Use SSL to connect client and Web server". Click Next.

The next panel requires you to specify the default authentication mechanism for WebSphere applications (this can be overridden by individual applications if necessary) as shown in Figure 77 on page 129.
Figure 77. Specifying LTPA authentication and SSO for WebSphere

Specify:

- Lightweight Third Party Authentication (LTPA).
- Leave the default token expiration time to be 30 minutes. This means that a user’s authentication credentials will expire after 30 minutes; if your environment has different requirements you can change this value. For example, if people typically required sessions that could last over 30 elapsed minutes, you would likely set the token expiration to a time somewhat greater than the expected session time.
- Select the box marked “Enable Single Sign On (SSO).”
- Enter your DNS domain to be written to the token. In our example it was lotus.com.

We will return to this panel to generate and export the LTPA key; first, we create the association to the Domino LDAP directory so this will be recorded in the generated key.

Select the User Registry tab (or press Next) to specify the LDAP server on the Domino server as the directory to be used to support authentication. Figure 78 on page 130 shows this panel as we filled it out.
Fill in the fields:

- **Security Server ID**
  The administrator ID specified earlier (WASAdmin in our example) in 4.4, “Setup of the Domino-WebSphere single sign-on environment” on page 122. This is the user ID that will have to be entered to start the WebSphere console once security is enabled. The user ID must be the same as the Short Name field in the administrator’s person document in the Domino Directory. (This format can be changed if desired by clicking the **Advanced** button to adjust the LDAP Advanced Properties.) Note that the use of user ID is allowed by Domino, but other LDAP directories (such as SecureWay) require a full distinguished name to be entered.

- **Security Server Password**
  The password you entered in the Internet Password field in the Domino Directory person document for the administrator specified.

- **Directory Type**
  Choose Domino 5.0 from the drop-down list.

- **Host**
  Enter the full DNS name of the Domino server running the LDAP server task (this was odin.lotus.com in our example).
The remaining fields generally should be left blank, but may be needed in special circumstances:

- **Port**
  If you are using an LDAP port number other than the default of 389 in Domino, enter the port number here. Otherwise, leave the field blank. If you do enter a port number remember to update the field “LDAP Realm” in the Domino Web SSO document to include a trailing slash before the colon and port number in this field, as shown on page 142.

- **Base Distinguished Name**
  If you are using distinguished names to sign into WebSphere, you can enter the part of the name which will be the same for all users signing in. For example, a user with a distinguished name of cn=Tom Jones,ou=ITSO,o=Lotus,c=US, could sign in with simply cn=Tom Jones if this field had ou=ITSO,o=Lotus,c=US; the remaining fields would be added when they authenticated. Care should be taken using this field since it will be added to all user ID entries. Our understanding is that this field is required with some LDAP directories; it is not required for the Domino LDAP directory server.

- **Bind Distinguished Name**
  If you have secured your Domino directory to prevent unauthenticated access, then enter in this field the fully distinguished name of an authorized user who can bind to the directory.

- **Bind Password**
  If you entered a Bind Distinguished Name, enter the corresponding password in this field.

Click the **Advanced** button to view the LDAP Advanced Properties dialog box. We did not need to change any of the entries here for our example. We discuss the fields and the implications of changing them in Appendix C, "Using WebSphere advanced LDAP properties" on page 463. Click **Back** to return to the Authentication Mechanism panel shown in Figure 77.

Click the **Generate keys** button to create a pair of LTPA keys. You will be prompted for an LTPA password, shown in Figure 79 on page 132, to protect the keys. Be sure to remember the password since you will need this to import the keys into Domino; if the password is forgotten, you will not be able to retrieve it in the future and will need to generate a new set of keys.
Once you click OK from the password prompt, you will notice the WebSphere administration console message (you may need to move the Security Wizard panel):

Command "Generate LTPA keys" running . . .

Wait until you see the message:

Command "Generate LTPA Keys" completed successfully.

on the WebSphere Administrative Console. When this is done, click Export to File to save the keys for later use in Domino. The conventional File->Save dialog allows you to choose the directory and file name to use. We chose to save the keys to the file DOMWAS.key in the WebSphere directory, as shown in Figure 80, but any other choice would have been acceptable.
Once you have generated and saved the LTPA keys to a file, click Finish to complete the process of defining WebSphere global security. You will be able to observe the progress of the security configuration in the WebSphere administrative console.

---

**LDAP must be running!**

If you did not have the Domino server and its LDAP task running, you will receive an error of the form “Update security configuration failed.” If you receive this message, ensure that the Domino server is running and that the LDAP task is running. You may want to reconfirm that the user ID you entered previously (on the panel shown in Figure 78 on page 130) is a valid user ID in the LDAP (Domino) directory by re-running the ldapsearch utility as we showed in Figure 71 on page 123. Once you have corrected the situation, click Finish again to complete the security configuration process.

The reason the LDAP server must be running is that the WebSphere security service will attempt to bind to the specified user ID and password to verify them. This is to ensure that you do not enter an invalid user ID/password combination that would later prevent you from starting the WebSphere Console.

The reason we generated the LTPA keys and saved them to a file at the end of the process was to ensure that the LDAP host name would be incorporated into the saved file. This information will be used later by Domino when you set up its Web SSO configuration. However, you can manually change the entry in the Domino Web SSO configuration document later if necessary. For example, if you save the keys in the Authentication Mechanism panel and then define the User Registry settings, the LDAP server stored in the exported key file will point to the host WebSphere is installed on rather than the target Domino server. If you do this, simply update the Web SSO configuration document (as shown in Figure 92 on page 142) to reflect the entry made on the User Registry panel as shown in Figure 78 on page 130.

When the process is complete, you will get a confirmation message telling you that Changes will not take effect until the admin server is restarted, as shown in Figure 81 on page 134. Click OK to dismiss the information panel.
Select your host name in the tree on the left pane of the WebSphere Administrative Console and right-click it. Select **Stop for restart** or **Restart** in the resulting context menu. If your WebSphere console is running on the same computer as the WebSphere administration server, you will be warned that the console will close once the server is shut down. Dismiss this warning by clicking **OK** and continue by clicking **Yes**.

Ensure that your Domino server is running and that the LDAP server task is loaded and available for service.

Once the server has shut down, restart the WebSphere server (it will restart automatically if you selected **Restart**) and then the administrative console. You will now be prompted for the Administrator ID you entered previously (see Figure 78 on page 130) because we have enabled global security for the WebSphere server. The challenge panel is shown in Figure 82. The Realm Name field will be pre-filled with the name you specified in the global security configuration process. In our case this was odin.lotus.com.

![Figure 82. WebSphere administrative console password prompt](image)

Fill in the user ID and password you specified when you enabled Global security in WebSphere.

**Note:** When using Domino and your LDAP directory you can enter the user short name simply as it is. If you use another LDAP directory you normally have to specify the name in hierarchical format. For example, for IBM Secureway you would need to write the User ID as `cn=WASAdmin`. 

134 Domino and WebSphere Together
Click **OK** to start the console.

Note that if you type an incorrect user ID and/or password on this panel, the WebSphere administrative console will not start or prompt you to correct your entry. Instead it will simply exit with the error dialog shown in Figure 83.

![Error dialog](image)

*Figure 83. Incorrect Administrator user ID and/or password entry*

This completes the SSO enablement on the WebSphere server. We will still have to set up WebSphere application security later. We describe how to create a WebSphere enterprise application in 4.5, “Setting up WebSphere application security” on page 147 in order to test the setup we have just created.

You should be able to enable SSO between WebSphere administrative domains by exporting the LTPA keys from one and importing it into the other domains. However, we did not do this in our environment.

We now are ready to configure single sign-on on the Domino server.

### 4.4.4 Enabling single sign-on in Domino R5

Note that this discussion is applicable to Domino R5.0.5 and above only. We used Domino V5.0.6a for our tests. There are two tasks:

1. Configure Domino Web single sign-on configuration. This must be done once for a Domino domain.
2. Configure Domino HTTP Session Support. This must be done for every Domino server in the domain that will support SSO.

#### 4.4.4.1 Configuring Domino Web single sign-on configuration

Ensure that the location document of your Domino administration client is pointing at a server in the domain for which you wish to configure SSO. This is necessary since the public keys of the participating Domino servers need to be accessed and used to encrypt the LTPA key which we will import from WebSphere.
Start the Domino administrator client and select the **Files** tab. Open the Domino Directory for your Domino domain as shown in Figure 84. You could also select **File->Database->Open** from the menu bar to open the directory.

![Figure 84. Domino Directory server document](image)

Double-click the Domino Directory entry (names.nsf) to open the directory.

Select the **Servers** view and click the **Web** button in the action bar as shown in Figure 85 on page 137.
The Web button will display a drop-down list. Select **Create Web SSO Configuration** as shown in Figure 86 on page 138.
Figure 86. Specifying creation of Web SSO configuration document in Domino
Enter the Token Domain; this was lotus.com for our example. This must the same as specified earlier for WebSphere for SSO to work since browser cookies will only be returned to servers in the same DNS domain.

Domino adds a dot in front of the token domain when saving the Web SSO configuration document so it becomes .lotus.com

Select the Domino servers you wish to participate in the SSO relationship. Only those servers selected will be able to participate since the LTPA keys will be stored in the document encrypted with the server's public keys. You can select these from entries in the Domino Directory by clicking the down arrow to the side of the field, or simply enter the server names directly separated by commas. You must later edit the server documents for these servers to
specify multi-server session authentication, as explained in 4.4.4.2, “Configuring Domino HTTP Session Support” on page 144.

Leave the Token Expiration set to the default of 30 minutes unless you changed this value in WebSphere (Figure 77 on page 129). The two values should be the same so that expiration times will be consistent when you access either Domino or WebSphere first in a session. Note that the token expiration time is the time from when it was issued, not a timer for inactivity.

The document as we filled it in is shown in Figure 87 on page 139.

Click the Keys button on the action bar and select Import WebSphere LTPA keys as shown in Figure 88.

![Figure 88. Specifying “Import WebSphere LTPA Keys”](image)

You will be prompted for the full path and file name to locate the key file you exported from WebSphere (see 4.4.3 on page 127). The prompt is shown in Figure 89 on page 141. Note that you need to place the key export file on a disk accessible to your Domino server. We chose to copy it to our Domino server, but we could just as well have used a network drive to access the WebSphere server location where we saved the key export file.
Chapter 4. WebSphere - Domino security and single sign-on

Figure 89. Specifying the path and file name of the WebSphere LTPA key import file

Click OK to import the key file. You will be prompted for the password you specified when you created the LTPA keys (see Figure 79 on page 132). The prompt is shown in Figure 90.

Figure 90. Entering the LTPA password in Domino Web SSO configuration

Click OK. You should receive the confirmation message shown in Figure 91.

Figure 91. Confirmation of successful import of WebSphere LTPA keys into Domino

A new section with WebSphere information from the imported key will now be displayed, as shown in Figure 92 on page 142.
The value for LDAP Realm must be the same as the LDAP Host entered in the User Registry, as shown in Figure 78 on page 130. If it is not, edit it now to hold the fully qualified DNS name of your Domino server running the LDAP task (or other LDAP directory specified). Ensure that this value is the same as the entry made in the User Registry. Our tests showed that the name of the LDAP server must be the same in both WebSphere and Domino; if it is not, the SSO token will not be accepted in both environments.

If you specified a port number in the User Registry panel shown in Figure 78 (and the Domino Directory Internet Ports entry), you need to further edit the field to ensure that the full DNS name ends with a backslash ("\"), a colon (":"), and the port number. For instance, in our example of odin.lotus.com and a nonstandard port of 459, we would enter:

odin.lotus.com\:459

in this field.
Click **Save and Close** to save this Web SSO configuration document. Ensure that your Domino Directory is replicated to the other servers in your Domino Domain so that this configuration document will be available to them. Note that there can be only one Web SSO configuration document in a Domino Directory replica (and thus only one for a Domino domain).

This completes the Web SSO configuration for your domain. You now need to set up each of the Domino servers in your domain to support SSO sessions.

---

**Do not lose the administrator’s ID!**

You will need the ID file of the administrator who created the Web SSO document if you wish to add more servers to the participating servers list or refresh the WebSphere LTPA keys. This is because the LTPA keys are encrypted with the public key of the administrator who created the document.

If the Notes ID of the administrator who created the Web SSO document is not available, it is easiest to simply delete the document and create a new one following the steps we described.

---

**Configuring SSO in more than one Domino domain**

It is possible to enable SSO in multiple Domino domains by following these steps:

1. Create the Web SSO document in your first Domino domain as we have just described.

2. The Administrator who created the Web SSO document should copy it to the clipboard and paste it into the Domino Directory for the other domain (“destination domain”), ensuring that the administrator's home server in the current location document (during the paste) is pointing to a server in the destination domain.

3. Edit the Web SSO domain document just pasted into the destination domain:
   a. Edit the Participating Domino Servers field, removing the server entries from the original domain and add the participating servers in the destination domain.
   b. Save the Web SSO document. You may wish to check, through **Document Properties**, that there is indeed a field “Public Encryption Keys” (not visible on the form) with the administrator's and the server's names. This confirms that the public keys are present in the document.

---
Repeat these steps for all Domino domains you wish to enable in the same SSO group. Note that they must all be in the same DNS domain (in our case lotus.com) since browsers will return cookies only to servers in the DNS domain where they originated.

4.4.4.2 Configuring Domino HTTP Session Support
Using the Domino administrator client, open the server document for your Domino server and navigate the tabs to **Internet Protocols -> Domino Web Engine**. Select the Session authentication field, as shown in Figure 93.

![Figure 93. Specifying session authentication](image)

Click the down arrow beside the Session authentication field and select “Multi-server” in the resulting keyword selection dialog shown in Figure 94 on page 145.
Click **OK** after selecting Multi-server. Click **Save and Close** on the server document to save your changes. Note that you must choose Multi-server even if there is only one Domino server since this setting forces the cookie to be shareable with WebSphere.

Make the same changes to the server documents for all other Domino servers that will participate in your SSO domain. If you do this in a replica of the Domino directory on one server, ensure that the Domino directory is replicated to all of the SSO-enabled servers before testing SSO on them.

Load the HTTP task on each of the Domino servers you specified to participate in Domino Web SSO, or, if they are already running, stop and then restart them. In each case you should get the message

```
HTTP: Sucessfully loaded Web SSO Configuration
```

if the configuration was successful.

### 4.4.4.3 Confirmation of Domino session-based authentication

In this test we simply point a browser at the Domino server we just configured using a URL like:

```
http://odin.lotus.com
```

This is to verify that, instead of the browser (Basic) Authentication Challenge (with realm “/” and host odin.lotus.com) shown in Figure 96 on page 147, we get a Domino session-based authentication document as shown in Figure 95 on page 146. Once you receive this challenge, you have enabled session-based authentication in Domino.
Note: You can change the appearance of the Domino session-based logon screen. To do this you must create a Domino configuration database based on the template named DOMCFG5.NTF and then modify the form named $$LoginUserForm, or tell Domino to use another login form in the database. The steps to do this are provided in the Domino Administrators online help. Search for something like Customizing the HTML log-in form to find the help description.

We still have to test single sign-on with WebSphere; this will be described in the following section after we set up a WebSphere enterprise application. You can, of course, confirm that SSO works with other servers in your Domino domain once they are configured to use multi-server session authentication.
Chapter 4. WebSphere - Domino security and single sign-on

Figure 96. Basic authentication challenge from a browser without SSO enabled

We have now set up the single sign-on environment for Domino and WebSphere. Now we need to create a WebSphere enterprise application containing our small test servlet called ReadNames (described in 4.3, “Example application” on page 120). For our enterprise application we must define its protected methods and assign permissions. Once this is done, we can test single sign-on between Domino and WebSphere.

4.5 Setting up WebSphere application security

We need to follow the steps outlined in 4.2.2, “WebSphere security” on page 113 to protect the ReadNames servlet. The steps are:

1. Create a Web application.
2. Add the ReadNames servlet to the Web application, specifying the URL that will be used to access it.
3. Specify the method groups to be assigned permissions for the resources in the Web application; in this example, only the URL pointing to the ReadNames servlet needs to have its read methods selected for protection.
4. Create an enterprise application and add the Web application and URL created in the previous steps to it.
5. Assign permissions to the enterprise application

The ReadNames servlet Java source is listed in Appendix B, “The ReadNamesRemote Servlet” on page 459. It is also available for download.
from the IBM Redbooks Web site in Java source as well as a class file. See Appendix K, “Using the additional Web material” on page 521 for information about how to get the download files.

4.5.0.1 Creating the example Web application

The following steps show creation and configuration of the Web application. For our example, we will create this on the Thor-2000 host.

![WebSphere Advanced Administrative Console](image)

Figure 97. Specifying Create a Web Application from the tasks menu

From the WebSphere console select Tasks -> Create a Web Application as shown in Figure 97.
Specify a name for the Web application as shown in Figure 98. We initially called our Web application *redbook example*, but the Web application name is also used as the default value for class and application paths, some of which have problems handling blanks. Therefore, we decided to call our Web application *Redex* (one word) instead.

Select to enable JSP 1.1 and click **Next**.

**Choose JSP version 1.1!**

For all Web applications in our examples, we specified JSP 1.1 so that JSP tag libraries will be supported.

The default is JSP 1.0, which does not support tag libraries.

The current ReadNamesRemote servlet does not require this change, but it will not hurt to specify it here.

You must specify JSP 1.1 to enable the banking application we describe in later chapters.

You now have to specify a servlet engine in which to run the Web application. Expand the tree under the server (Thor-2000 in our example), pick an application server (Default Server in this case), and then select a servlet.
engine within the selected application server (Default Servlet Engine), as shown in Figure 99

Figure 99. Choosing a servlet engine to run the Web application

Click **Next**. The next panel will allow you to specify the Web path within a virtual host to access the contents of the Web application. This is shown in Figure 100 on page 151. The name of the Web application is displayed on the first line and could be changed if desired. In our examples, we simply added a description; the other settings were already filled in based on our earlier selections.
Figure 100. Specifying virtual host and Web path to access the Web application components

Press Next.

---

Do not put blanks in Web path!

WebSphere will accept any Web path you input. However, you should not create Web paths with imbedded blanks since this will generate URLs that browsers cannot follow. This is because blanks and certain other characters are treated as "end of URL" characters in some browsers.

---

You will be able to change any of the advanced settings for the Web application, as shown in Figure 101 on page 152. The settings as entered should be satisfactory.
Click Finish. The Web application is complete. Now we need to create and add the ReadNames servlet definition to it.

4.5.0.2 Creating the ReadNames servlet
For our example, we created the ReadNameRemote.class file using VisualAge for Java (creating a package com.lotus.wastest for this test class) and placed it manually in the directory:

C:\WebSphere\AppServer\hosts\default_host\Redex\servlets\com\lotus\wastest

It is necessary to specify the exact directory path for the servlet to load. If you downloaded the class file from the IBM Redbooks Web site you must manually create the folders in the path shown above.

The first part of the path (hosts\default_host\Redex\servlets, under the WebSphere installation directory of \WebSphere\AppServer) is the classpath for the Web application Redex. The second part (com\lotus\wastest) is the package that contains the class file. If the directory structure does not match the package definition, the servlet will not load.

Select Tasks->Add a Servlet from the WebSphere administrative console. You will be prompted whether you already have a class file for the servlet.
Specify the information necessary to name and locate the servlet as shown in Figure 102.

Choose **Yes**, and click **Next**. You will now be prompted to select a Web application to which the servlet will be added, as shown in Figure 103 on page 153. Choose the Web Application just defined (*Redex* in our example).
Click **Next**. You will receive a panel allowing you to search the file system for your class file, as shown in Figure 104. Click **Browse**.

![Add a Servlet dialog box](image)

*Figure 104. Initiating a file system search for class or jar files*

Navigate the file system **Open** dialog box until you locate the directory in which you placed the class file. Make sure that files of type All Files (*.*) is shown in the dialog box (initial selection is **Zip Files only**) or you will not be able to see the file of type class.

Select the ReadNamesRemote.class file and click **Open**. This is shown in Figure 105 on page 155.
Chapter 4. WebSphere - Domino security and single sign-on

You will be returned to the panel shown in Figure 104, with the full file system path to the class file shown. You may have to scroll the display to see the entire path. Click **Next**.

On the next panel, shown in Figure 106, you will be prompted to specify the type of servlet to be configured. Accept the default of Create User-Defined Servlet and click **Next**.
Finally, you need to fill out the servlet definitions, as shown in Figure 107.

![Add a Servlet](image)

Figure 107. Specifying servlet name, class name, and Web path

You need to specify the following items:

- **Servlet Name.** This is necessary to identify the servlet in the console tree under the Web application. It need not be the same as the actual class file. We chose `ReadNames`.

- **Web application to which the servlet will belong.** This is a drop down list of Web applications created in this WebSphere domain. We chose the Web application just created (`Redex`).

- **Servlet class name.** This is the path to the servlet and its class name. In our example, this was `com.lotus.wastest.ReadNamesRemote`. This is the package name that it was created using VisualAge for Java; it must be specified exactly.

- **Servlet Web path list.** This is the Web path that will be used to access the servlet from a Web browser. The path is entered by clicking the **Add** button and typing in the final part of the path, as shown in Figure 108 on page 157. We chose `default_host/webapp/Redex/ReadNames`

  The first part of the path was filled in since it was inherited from the Web application (`webapp/Redex`) and you cannot change this part of the path.
Figure 108. Filling in the servlet Web path

Click **OK** to accept the new Web path, and **Finish** to accept the definition of the servlet. This completes the creation and configuration of the Web application *Redex* and adding the ReadNames servlet to it. You can, if you wish, start your Web application and test the servlet with a browser. If you want to do so, ensure that you have added the Domino ncsow.jar file (described in 3.9, “Adding the Domino ncsow.jar file to the WebSphere classpath” on page 106) to the WebSphere administrative server classpath before proceeding. To test the servlet, you need to ensure that you authenticate with Domino first to generate the SSO token that the servlet needs to access the Domino directory. You can then request the servlet via the IBM HTTP server on the WebSphere computer.

We used the following URL to test the servlet:


Note that our ReadNames servlet requires a host parameter like ?Host=odin.lotus.com to work correctly.

You can also test using the Domino HTTP server, but if you want to do this you first have to make sure that the properties files on the Domino computer have been updated with the new Web paths you just defined. See 3.7.8, “Configuration of OSE remote transport for Domino” on page 93 for more information about manual or automatic update of the properties files. Remember the refresh interval for the queues properties file is approximately five minutes.

**Note**: If you do not authenticate with Domino before attempting to execute the servlet you will get an error about missing the token; you may also get a message saying the the token has expired. In this case you simply need to authenticate with Domino again. Finally, if you get the cryptic error message “4488” it probably means that you haven’t allowed your user to run Java code on your Domino server, as described in 4.4.2, “Configuring the Domino server to support LDAP and IIOP” on page 123.
We now need to create the enterprise application so we can assign WebSphere permissions.

**4.5.0.3 Creating an enterprise application**

From the WebSphere Administrative Console select Tasks->Create Enterprise Application. On the first panel, shown in Figure 109, specify the name of the enterprise application. For our example, we named the application *Redbook Enterprise Example*.

![Create Enterprise Application Wizard](image)

**Figure 109. Naming an enterprise application**

Click **Next**. You now need to select the resources to be included in the enterprise application. Figure 110 on page 159 shows selection of the Web application Redex. Highlight the entry, then click the **Add** button to add the Web application. Note that this will also add associated Web resources (URLs under virtual hosts) automatically.
It is not sufficient to simply highlight the entries to be added to the enterprise application. You must also click **Add**; if you don't, there will be no error prompt, but your enterprise application will not contain the selected resources.

You will be able to configure security but will not receive any indication that there are no Web applications or Web resources in your enterprise application. By default, the resources will then be completely unprotected.

Be sure to click **Add!**

Click **Next**. You will be able to review the contents of the enterprise application and to remove any resources that were added in error. We will not make any changes here.

Once you are satisfied with the the contents of the enterprise application, click **Finish**. This completes the definition of the enterprise application; you should see it added to the WebSphere console tree in the left pane, as shown in Figure 111 on page 160.
Note that the enterprise application just added has Web Resources and Web Applications contained within it; this confirms the successful addition of the Web application and the corresponding Web resources.

Note that the enterprise application appears at the bottom of the tree under default_host (and any other virtual hosts you have defined). It will remain in this position until the console is stopped and restarted; when the console builds the tree again, the enterprise application will be located above your host (Thor-2000 in our case) in the tree, as shown in Figure 112 on page 161. (Figure 112 shows method groups; these will be present only after we have configured application security as described in the next section.) As well, not shown in our illustration, the Web application will have a cross-reference in the tree to enterprise applications. The reason this is in the plural is that the Web application could, in principle, belong to more than one enterprise application.
We are now ready to protect resources within the enterprise application and add permissions.

4.5.0.4 Defining security for the enterprise application

From the WebSphere administrative console, select **Tasks->Configure Application Security**. You will be prompted to select one or more enterprise applications, as shown in Figure 113. We chose the enterprise application just created, Redbook Enterprise Example.
Click **Next**. You will be prompted to select any application-specific realm and authentication challenge type options. This panel will be prefilled with the global security authentication specifications you entered previously, as shown in Figure 76 on page 128. For our example, we accepted the defaults as displayed and clicked **Finish**. (Clicking **Next** would have given us a panel on which to specify a user ID and password to be used to access EJBs; we did not need this in our simple sample since there are no EJBs).

We now need to configure resource security. From the WebSphere administrative console, select **Tasks->Configure Resource Security**. You can select EJBs (“Enterprise Beans”) and Virtual Hosts. Expand the Virtual Host node and then the default_host node to see a list of the URIs to be protected. For our example, we only need to select the URI used to access the ReadNames servlet, which is `/webapp/Redex/ReadNames` as shown in Figure 114.

![Create Enterprise Application Wizard](image1.png)

*Figure 114. Selecting a Web resource (URL) to apply security*

Click **Next**. You will be prompted whether you wish to assign default method groups, as shown in Figure 115 on page 163.
Click Yes and WebSphere will assign method groups according to the rules in Table 3 on page 116. Your screen should now look similar to Figure 116.

WebSphere has assigned the default method groups grouped by the standard HTTP directives. If you expand one of the HTTP methods you can see the corresponding method group (HTTP_DELETE - remove methods, HTTP_GET - read methods, HTTP_POST - read methods, and HTTP_PUT - write methods). By default, only read, write, and remove methods will be present.

If you for some reason want other methods groups than the defaults assigned to one of your HTTP directives, you can select the HTTP method and click the Add ... button. This opens a dialog that allows you to add additional method groups, as shown in Figure 117 on page 164.
You can also define your own method groups (for example, Red Hot Methods) if you want, but here we will stick with the default groups.

Our selection screen is shown in Figure 118, with the default methods selected.

Click Finish. We can now assign permissions to the method groups that we selected. Recall from our discussion in 4.2.2, “WebSphere security” on
page 113 that, although we have selected resources to be protected, we will assign permissions to the selected method groups to the enterprise application containing the resources.

Select **Tasks->Configure Security Permissions** from the WebSphere administrative console and select the enterprise application (Redbook Enterprise Example) created earlier. The selection panel is shown in Figure 119.

![Figure 119. Selecting an enterprise application to apply permissions](image)

Click **Next**. On the next panel, select one or more method groups to apply permissions to. As shown in Figure 120 on page 166, we chose to apply permissions to the read methods since this is all that is necessary for our example.
Figure 120. Configuring permissions for the enterprise application

Once you have selected the method groups to apply permissions, click Next. Note that, although you can select method groups that were not selected to be part of the protected resources and therefore the enterprise application, such selection will not have any effect.

You can now select who will be permitted to use the selected method groups. As shown in Figure 121 on page 167, we chose to allow all authenticated users to access the read method groups since we simply need to force an authentication prompt to test single sign-on.
If we wished to select specific users or groups we could also have chosen the **Selection** option, choosing Users or Groups to search, entering a search string (* returns all) and clicking **Search**. This would initiate an LDAP query; the results would be returned in the Search Results window, as shown in Figure 122 on page 168. One or more user entries could be selected. (Multiple selection is accomplished by pressing the Shift key and clicking on entries with the left mouse button to add a contiguous group of entries or Ctrl and the left mouse button to add individual entries.) Similarly, you can search for groups to which to apply permissions from this panel.

Note that the format of the entries on this panel will depend on the settings you specify in the User ID Map field of the LDAP Advanced Properties panel. This is discussed in Appendix C, “Using WebSphere advanced LDAP properties” on page 463. The LDAP Advanced Properties panel is shown in Figure 302 on page 463.
We will stick with our choice of All Authenticated Users.

Click **Next**. You will be presented with a panel which allows you review the permissions for the application and to remove any permissions added in error.
As shown in Figure 123 on page 168, we have only added All Authenticated Users to the read methods of the enterprise application.

Once you are satisfied with your selection (you can click Back if you need to add more permissions), click Finish.

This completes configuring security for the enterprise application. We are now ready to test single sign-on between Domino and WebSphere.

You can confirm that the method groups are used in the enterprise application. Ensure that the enterprise application is refreshed by highlighting it in the tree view of the WebSphere console and pressing the refresh icon in the toolbar. You should see an item labelled Method Groups under the enterprise application. If you highlight it, you will see the method groups now used in the enterprise application, as shown in Figure 124.

Figure 124. Displaying method groups in the enterprise application

You can also confirm your overall application security settings by locating the enterprise application in the WebSphere console (refreshing it as necessary or stopping and restarting the console) and highlighting the item Application Security under the enterprise application. The security configuration just entered will be displayed in the information pane of the console, as shown in Figure 125 on page 170.
We are now ready to test our SSO setup.

4.6 Testing single sign-on between WebSphere and Domino

The actual test to verify that SSO is working between our computers is very simple.

Note: Make sure the properties files for the DSAPI plug in on the Domino machine are up to date. You can either use manual or automatic update as described in Chapter 3.7.8.1, “Configuring WebSphere OSE transport to use INET sockets” on page 94. Otherwise the plug in will not know to forward URIs of the form /webapp/Redex to WebSphere.

Start the enterprise application by selecting it in the WebSphere console, right-clicking and selecting Start from the context menu, as shown in Figure 126.
Chapter 4. WebSphere - Domino security and single sign-on

The Web application will also start automatically (if it was stopped) when you start the containing enterprise application.

Access your Domino server, specifying the Domino Directory in the URL like this:

http://odin.lotus.com/names.nsf

You will receive the session-based authentication prompt shown in Figure 127.

Once you click **Login**, you will be shown the (non-hidden) views in the Domino Directory on your Domino server, as shown in Figure 128 on page 172.
Figure 128. Listing of views in Domino Directory after SSO sign in

Your browser will now have a cookie that will be sent to other servers in the same DNS domain and can be used to prove authentication to other Domino servers and WebSphere servers which share the same shared secret (LTPA keys). If there is more than one WebSphere server, they all must be in the same administrative domain and, therefore, share the same administrative repository (the WAS database by default).

To test this, simply specify a URL to access the ReadNames servlet on the WebSphere server, like this:


In our example, we had Domino and WebSphere installed on separate hosts and accessed the servlet through the Domino HTTP server (using OSERemote and the thin servlet redirector in separate tests); we could also
use the IBM HTTP server on the WebSphere server. In either case, the cookie would be accepted for authentication. The proof that SSO is active is negative: you will not be prompted for your user ID and password (or client certificate).

Once the servlet is invoked, it can access the cookie to get the authentication credentials the user entered. In turn, the servlet uses the authentication credentials to access Domino. The results returned from the servlet are shown in Figure 129.

![Persons in Domino directory](image)

**Figure 129. Results returned from the ReadNames servlet**

You do not need to sign into Domino first; you can instead contact the same path (.../webapp/Redex/...) at your WebSphere server running the IBM HTTP server; in this case you will get a basic authentication prompt, as shown in Figure 130 on page 174. (You will have to refresh the browser after the authentication to get a reply, though because of the simplicity in our test program.) Note that both the realm (lotus.com) and the host name (Thor-2000.lotus.com) are shown in the challenge for confirmation. This challenge also confirms that we successfully configured security for the enterprise application in the previous steps.
Once you have successfully logged in, you will get the display of the Domino Directory shown in Figure 129 on page 173. Note that you will have to click "refresh" on your browser because, although the servlet is loaded when you logged in, it does not yet have the cookie, so it has no credentials to authenticate with Domino. Once you press "refresh" (often it will refresh when you adjust the size of the browser window), the cookie is sent by the browser.
and is accessible to the servlet. The response of the servlet immediately after logging in (but before refreshing) is shown in Figure 131, but the URL will reflect the host name of your WebSphere server (Thor-2000.lotus.com in our example).

Figure 131. Servlet response when SSO token is not available

This completes the configuration and testing of single sign-on between Domino and WebSphere.

4.7 Troubleshooting single sign-on

If you followed our instructions earlier in this chapter in the sequence we described, you should not have any difficulty in enabling SSO. However, there are many subtle differences between environments that may cause problems. If you do have a problem, we suggest checking the following points:

- Check that the Web SSO document in the Domino Directory has been replicated to all servers participating in the SSO relationship. You should also do this if you have modified this document.

- Ensure that you are using the same LDAP directory for both Domino and WebSphere. This need not be the Domino Directory since Domino can authenticate Web users with a remote LDAP directory using Directory Assistance. If you use this configuration, ensure that people are authenticating correctly through Domino before attempting to enable SSO.

If you are using another LDAP directory with Directory Assistance, entering the line:

```
webauth_verbose_trace=1
```
to the server’s notes.ini file will show trace messages in the Domino server’s console. These can also be sent to a file by entering a pathname to a file in the notes.ini entry:

```
Debug_Outfile=<Drive>:\Path\filename.ext
```

The actual drive, path, and filename are arbitrary, but the path should exist before starting the Domino server. This entry will take effect the next time you start your server; the filename will be incremented with a sequential number each time you restart the server subsequently. The current version will have the name you specified in the notes.ini parameter.

- If you are using the Domino Directory as your LDAP directory for your SSO realm, ensure that the field “Fully qualified Internet host name” in the Basics tab of the Server document is set to the full DNS name of the host computer on which Domino is installed. If the computer has more than one DNS name, make certain this entry matches the one which Domino is using in your configuration.

To check and correct this, edit the server document, select the Basics tab, and enter the full DNS name of the server in the “Fully qualified Internet host name” field.

You should ensure that the LDAP port number in use (default: 389) matches your environment.

While you have your server document open, select the tabs Ports-> Internet Ports->Directory. If the field “TCP/IP Port Number” has been changed to a port other than 389, either reset it to 389 or ensure that WebSphere is configured (in global security user registry) to use the designated port. If you change this in WebSphere you will have to stop and start the WebSphere administrative server for the change to take effect.

- When you set up the Web SSO document in Domino, ensure that the administration client’s location is pointing at a (home) server in the same Domino domain. This is because the client must be able to find Domino server documents for the participating SSO Domino servers to retrieve their public keys. The Web SSO configuration document is encrypted for the servers specified in the document (and the administrator who created it) so they can open it and retrieve the LTPA keys from it. During setup, if you get a message box that states that one or more of the participating Domino servers cannot be found, then those servers will not be able to decrypt the Web SSO document and will not be able to decrypt an LTPA key and thus perform SSO. (Instead, they will report Error loading Web SSO Configuration and will revert to single server session authentication.)
Ensure that the entries in the LDAP directory you are using support both a ShortName (with an alias of uid) as well as a fully distinguished name, which must be the first entry in the "FullName" field in the Domino Directory. In our example, the user ID we created had a ShortName (uid) of WASAdmin and a fully distinguished name (FullName) of cn=WebSphere Administrator,o=DOMWAS. The uid field is used by WebSphere (in its default configuration) to look up the person; when the person's data is returned from the LDAP directory, the fully distinguished name is used to bind (along with the password entered); if the bind is successful, the distinguished name is stored in the LTPA token. Thus both entries must be present in the person documents (for a Domino Directory) or person records (for another LDAP directory).

The default LDAP query format that WebSphere uses to locate a person in a Domino Directory, given a user ID is:

```
(&(uid=<userid>)(objectclass=dominoPerson))
```

where `<userid>` is the user ID string entered for authentication.

Note that this also implies that you cannot use flat user names (or common names) if you are using WebSphere with this default LDAP query since these will not be recognized as valid user IDs or shortnames by Domino. You can change this behavior so that a user can use their common name, or force them to enter a fully distinguished name (in LDAP format) if you wish, by changing the query filter as we describe in Appendix C, “Using WebSphere advanced LDAP properties” on page 463.

If you only have Domino servers in your SSO realm, flat names are tolerated (but discouraged) because Domino will use all entries in the $Users view to authenticate.

If you do not allow anonymous access to the Domino LDAP server, all user ID strings must be in fully distinguished name format (LDAP format). Note that you can always specify fully distinguished names to authenticate. Furthermore, unless you are using Directory Assistance, you can log into the Domino HTTP server with any string that Domino can identify in the Domino Directory since it does not use the LDAP task to authenticate by default.

URLs sent to servers for single sign-on must specify the full DNS server name. This is because browsers use the DNS domain part of the URL to decide whether to send a cookie to a host. Thus, otherwise acceptable variations on the URL such as `hostname`, `localhost`, and the server's IP address will not work. Note that this also implies the requirement that all servers (Domino and WebSphere) participating in an SSO realm must be in the same DNS domain.
If you are using the Domino Internet Cluster Manager (ICM) to load balance between servers in a Domino cluster participating in an SSO realm, you must ensure that the full DNS names of the servers in the cluster are in the servers’ Server documents (as we described previously for the LDAP Directory server) in the Domino Directory. This is because ICM will use the contents of this field, by default set only to the hostname of the server, to redirect URLs to cluster servers. A browser would not send the LTPA cookie to the redirected URL since it would not recognize that it belongs to the DNS domain for which the cookie was intended.

If the server is not in a cluster, it is not necessary to set this field (but it is good practice). Once again, you must set it for the server which runs the LDAP service used for SSO.

- If you are specifying an LDAP port in the WebSphere global security configuration user registry panel (see Figure 78 on page 130 for a sample of this panel), the field “LDAP Realm” in the Domino Directory Web SSO configuration document must be edited and a trailing backslash (“\”) must be added to the DNS name before the colon and port number. We illustrate this panel (with the “LDAP Realm” field circled) on Figure 92 on page 142. For example, if you specified the default LDAP port of 389 in the WebSphere global security user registry panel, you would have to add \:389 to the DNS name. For our example using:

  odin.lotus.com

we would change this to:

  odin.lotus.com\:389

The default of odin.lotus.com:389 which would be exported in the LTPA key from WebSphere will not work.

- If you are using the Domino Directory for LDAP and allow anonymous (reader) access to it, its default configuration will work without any changes. However, if you have altered the LDAP configuration, you should check the LDAP configuration settings. If this document exists, either for all servers in your Domino domain or for the server with the LDAP task, you should ensure that the field “Anonymous users can query” contains the “ShortName” field and the “FullName” fields; if it does not, edit the document and add the missing entry or entries using the button marked <<>>. Again, if you change the default filter WebSphere uses to search for people in the LDAP directory, you should ensure that any fields you add to the filter can be queried by anonymous users. The field “Maximum number of entries returned” should be left at its default setting of 0 (no limit). In addition, ensure that the “Timeout:” field is left at its default of 0 (no timeout).
If you do not have a configuration document (either for all servers in your Domino domain or for your LDAP server), the default settings in use will be adequate provided you are using the WebSphere default User Registry configuration.

• Ensure that all entries you are using can be accessed via the ldapsearch tool, as illustrated in Figure 71 on page 123, with all of the necessary fields returned. You should also ensure that you specify all the options on the command line that you forced when you configured WebSphere global security, as described in 4.4.3, “Configuring WebSphere V3.5 global security for single sign-on” on page 127. The command line options are described in the Domino Administration Help database or (in abbreviated format) by simply entering `ldapsearch` with no arguments on a command line.

You can pipe this output to a file to examine it at your leisure. For our example using the LDAP server odin.lotus.com, you would simply issue the command:

```
ldapsearch -v -h odin.lotus.com "uid=*" > ldapsearch.txt
```

in a command window to store the results for all entries in the Directory (with a uid or ShortName field) into the file ldapsearch.txt. You can print or edit this file to check whether any entries are missing either a distinguished name or a shortname entry. If so, correct the person’s entry in the Domino Directory.

### 4.8 Summary

In this chapter we introduced the general concept of single sign-on and we discussed how WebSphere secures resources. We described in step-by-step fashion how to enable global security in WebSphere using Domino Directory as the LDAP directory. Finally, we created an enterprise application with servlets that access a Domino server from a WebSphere server to verify that SSO is working properly.
Chapter 5. Introduction to the example application

Part 2 of this book is about the sample application we developed.

To show how to build an application that takes advantage of the combination of Domino and WebSphere, we created a content management system that includes a navigator for all content documents, a customized full text search over all Domino documents, and a personalization page for every user.

We selected the banking example that comes with WebSphere as an example for accessing enterprise systems via Enterprise JavaBeans (EJBs). An example of our application’s user interface can be seen in Figure 132.

The account information is stored in a DB2 database and accessed through an EJB. All other information is stored in Domino databases and either accessed directly or through servlets and a personalization EJB.

When a new user accesses the system for the first time, their personalization information is created automatically. In addition, a checking account and a savings account are created for the user. The user can then perform one-time and recurring transfers between his accounts or to the accounts of other users.

In this chapter we provide a brief overview of the elements in our sample application and in the following chapter we go into greater detail.

Figure 132. The Redbook Banking main panel
This chapter presents an overview of the following aspects of our application:

- The main framesets and how they are populated
- The database architecture
- The JSPs and servlets
- The EJBs
- The Domino application design
- The flow of control for Web users

At the end of the chapter we explain how the other chapters in this part of the book are organized.

5.1 Framesets

Our main frameset contains four frames. At the top is a JSP that displays a static logo, the current account balances of the connected user, a quick search text box, and a combo box that includes special internal and external links. The content of the combo box is collected from Domino.

Below the top frame are three frames. On the left side a global navigation JSP is displayed, in the middle the current Domino document is visible, and the right frame is used for the PanelBuilder personalization servlet and the search servlet. We describe the servlets in 5.3, “Servlets and JSPs” on page 186.

The main frameset is shown in Figure 133 on page 185.
Chapter 5. Introduction to the example application

5.2 Database architecture

The example application contains three Domino databases, one for published documents, one for the personalization information, and one for the authoring process. The authoring is performed using a Notes client. The account data is stored in a DB/2 database.
All approved documents are copied from the authoring database to the database containing the published documents using an agent. If a version of the copied document already exists, the old version is deleted from the database that contains the published documents. All versions are kept in the authoring database.

The first time a user connects to the systems a checking account and a savings account are created (with zero balances!) In addition, a document is created in the personalization database for this user containing default information.

All access to the account database and the personalization database is encapsulated into two EJBs. The EJ Bs are called from servlets and JSPs. For a recurring transfer from one account to another the EJ Bs also are used from a Domino agent.

To navigate through the published documents, a JSP that reads the Domino data dynamically is used.

5.3 Servlets and JSPs

When connecting to the application, the user first calls the Login servlet to create the application information. We discuss the basic flow of control to do this in 5.6, “Flow of control for Web users” on page 192. All servlets and JSPs of the application use this application information, so the Login servlet always must be called in every session before any other servlet or JSP is called.
Apart from the Login servlet, the application contains two JSPs and three servlets. The JSPs are displayed in the top frame and in the left frame. On the right side either the PanelBuilder servlet or the Search servlet's output is displayed. To save the query string the PanelBuilder servlet needs, the SaveQuery servlet is used to capture and store a query for a user.

5.3.1 The JSPs of the example application

The JSP in the top frame displays the current account balances of the user that is logged in, as well as some links to Domino documents. The JSP in the left frame displays a tree of all defined areas. When the user clicks on an area, it opens a Domino view that is categorized by the area and displays all content documents of the area the user has clicked on.

We use a “utility” servlet named CallDominoJsp to perform required checking and setup before invoking the JSP pages.

5.3.2 The PanelBuilder servlet

The PanelBuilder servlet reads the user's personalization information using the personalization EJB. Part of the personalization information is a full text query string the user can define.

Using this query, the PanelBuilder performs a full text search in the database containing the published documents and displays the result.

In our example application each user can only save one query string and creates one panel by doing this. In a real world application each user would have multiple panels. Other possibilities to create a panel could be implemented, for example displaying the contents of a view or building a list of links to documents actively, like in the bookmarks of the browser.

5.3.3 The Search servlet

The Search servlet also performs a full text search of the published documents, but it does not use a saved query string. Instead, it builds the query just before performing the full text search.

This technique enables the use of a customized search form. The query can be built by combining the contents of the different fields. Our example servlet retrieves the information on the fields it will use from the search form.

Although we did not do this in the example application, it would also be possible to search in multiple databases using a servlet.
5.3.4 The SaveQuery servlet

The SaveQuery servlet is used to create the query string that is saved in the personalization document. It accepts the same parameters as the Search servlet. It does not perform a full text search, but it creates a query string and passes this string to the personalization EJB. The EJB then saves the query string in the personalization document of the current user.

5.4 Enterprise JavaBeans

In our example we are using the Enterprise JavaBeans (EJBs) from the banking example that comes with WebSphere. See 10.3.1, “Installing the EJBs” on page 414 for instructions on installing these EJBs.

5.4.1 Using Account & Transfer EJBs of the IBM banking example

When a new user connects to the system for the first time, a checking and a savings account are created for them. This operation is performed by the PanelBuilder servlet that calls the Account EJB. Both accounts are created with zero balances.

To store an amount in these accounts, you create one source account with a large initial amount using the CreateAccount servlet that comes with the original WebSphere banking example (see Appendix E, “Installing the IBM WebSphere 3.5 banking example” on page 479). You then can use the TransferFunds servlet that uses the Transfer EJB to transfer amounts from the source account to the checking and savings accounts of users. The IBM example does not provide the function of depositing money in an account. Instead you must transfer it from another account.

The Transfer EJB also is used to move amounts from a user’s checking account to his savings account periodically by the agent ag_TransferFunds.

5.4.2 The Personalization EJB

To support our personalization we developed an EJB to handle the read and write access to the personalization documents in a Domino database. When no document for the current user exists, it is created by the EJB.

The access to Domino is from the EJB performed using an administrative ID. The main advantage of this access is performance. The Domino session is kept open and does not have to be initialized each time. In addition, only the administrative ID needs to have access to the Domino database.
The disadvantage is that no Domino user rights can be used during the access via a single administrative ID. You should enable security for EJBs in any case to prevent use by unauthorized clients, even though EJB security does not apply to single Domino documents or design elements, but only to methods of the EJB.

If you need more granular access control (that is, to the document level) you either have to initiate a Domino session for every access or you need to handle the security in the EJB methods programmatically.

In our example the security targets are quite easy to achieve because we only have to make sure that each user only modifies their own personalization document. We decided in favor of performance and our methods ensure that we only access Domino documents that contain the name of the currently connected user.

### 5.5 The Domino application design of the example

The Domino application consists of three databases that have the names and content shown in Table 4.

**Table 4. Domino database file names**

<table>
<thead>
<tr>
<th>File name</th>
<th>Content of the database</th>
</tr>
</thead>
<tbody>
<tr>
<td>intra_01.nsf</td>
<td>Published documents</td>
</tr>
<tr>
<td>intra_02.nsf</td>
<td>Personalization</td>
</tr>
<tr>
<td>intra_03.nsf</td>
<td>Authoring</td>
</tr>
</tbody>
</table>

#### 5.5.1 The database for published documents

The database for published documents is designed for read access by a Web browser and creates the frameset shown in Figure 133 on page 185. It also contains all the data the application uses to display the frame contents. The frame sets are defined in setup documents. Since all content documents are copied into this database by an agent, users only need read access in this database. Only the user who signed the agent and an administrator, who needs to update the frameset and setup information, should have editor or manager access.
The database has the forms shown in Table 5.

**Table 5. Forms in the database for published documents**

<table>
<thead>
<tr>
<th>Form name</th>
<th>Purpose of the form</th>
</tr>
</thead>
<tbody>
<tr>
<td>fo_ContentArea</td>
<td>Defines an area within the Internet or Intranet. All documents are assigned to one or more areas. The navigation tree on the left side is built using these areas.</td>
</tr>
<tr>
<td>fo_Content</td>
<td>Contains an internal content document and contains the text of it. The content document can use different subforms as Web document templates.</td>
</tr>
<tr>
<td>fo_AreaContentList</td>
<td>Used to display the list of content documents for a single area. The form contains a single embedded view and is not used to create documents. It is called by the navigation tree in the left side frame.</td>
</tr>
<tr>
<td>fo_Link</td>
<td>Defines a link to an external document. Populates a drop down box with link in our application.</td>
</tr>
<tr>
<td>fo_Files</td>
<td>Can be used to store files that are referenced in the content documents.</td>
</tr>
<tr>
<td>fo_Search</td>
<td>Form to define an advanced search. The contents of this form are posted to the Search servlet.</td>
</tr>
<tr>
<td>Setup and frameset forms</td>
<td>These forms are used by the administrator to define the overall setup of the application and the framesets.</td>
</tr>
</tbody>
</table>

Most of the views of this database are used for lookups only. The JSP navigation page only uses the view viref_foContent$ByAllCat.

### 5.5.2 The personalization database

The personalization database contains one document per user. This document contains the personalization information as well as bank account numbers for the user. When a Web user connects to the system for the first time, a new personalization document is created. This is done by a servlet that in turn calls an Enterprise JavaBean. Since all access to the personalization database is performed by this EJB, only one administrative Domino user needs to have more access than “depositor” to this database, and the database does not have an interface visible to the user.
The database also contains an agent called ag_TransferFunds that transfers funds between the accounts periodically to illustrate the use of an EJB from a Domino Java agent.

### 5.5.3 The authoring database

The authoring database is used to create or modify the Domino documents the Internet or intranet site hosts. This database is designed to be used by a Notes client.

It contains the same forms as the database for published documents. In addition, it contains an example form that is used to display one example document for every subform to show the author how a new document they are about to create will appear. The content form has category fields (not visible to the Web reader) to ensure a consistent categorization of the internal documents.

The views of this database always display the document created with one form and in one status, as shown in Figure 135.

![Figure 135. The user interface of the authoring database](image)

The documents can be saved in four different statuses. When a document is created it is in Draft status. As soon as the author has finished working on the
document, the action button **Ready for approval** can be clicked to set this new status for the document.

A user that has the role **CM** (for Content Manager) followed by a division name (e.g. [CM Marketing]) can approve or reject the new document.

If a user with the content manager role is creating a new document, the document can be approved directly by the same user without setting the status to **Ready for approval**.

If a document is rejected, the status is reset to **Draft**; if it is approved, the status is set to **Approved** and the document can no longer be edited.

A periodic agent then copies all documents in status **Approved** to the database for published documents and sets the status to **Published**.

To change an existing document that already is published an author can use the action button **Create new version**. To delete a published document the action **Delete** can be used. The document is replaced or deleted by the same agent that publishes documents.

Older versions of documents have the status **Archived**.

In 10.4.1, “Creating content” on page 438 we briefly walk through the flow of creating, approving, and publishing a content document.

### 5.6 Flow of control for Web users

When a Web user connects to our example site, a Domino resource is called first. This ensures that a valid session token is available because Domino displays its login screen until the user supplies a user ID and password and it can create a valid token.

If you are working with servlets that have no security applied to them, you should always use this technique to force creation of the authentication token. If you have enabled security for a servlet or JSP, you could call it instead of this Domino resource. However, a servlet that creates a session token cannot use it immediately to connect to Domino.

It would be possible to write a servlet that reloads itself if it does not get a valid session token, but this would create a risk of a permanent loop in the case that no cookie can be created. That is why we recommend letting Domino create the session token in most cases.
The Domino resource we are calling to ensure that the user is logged in is a setup document. If the user has logged in successfully, it redirects the browser to a login servlet and passes the path to the Domino database, the type of the browser that is connecting, and the next URL that will be called within the database as URL parameters.

The Login servlet then creates a session object for each user containing the information that was passed to it. The session object also contains information about errors that occurred earlier in the same servlet session. If an error occurred, it displays the appropriate error message; if not, it redirects the browser to the next URL that was passed to it as a parameter.

All other servlets and JSPs use the session information created by the Login servlet. If an error occurs, they redirect the URL to the first protected Domino resource. This is done because, if the session token expires, it is detected as an error by a servlet or JSP. In a production environment this is the most frequent error detected because the token can expire while the user is working. When it expires (based on the token expiration time specified in the Domino R5 server document and in the WebSphere global security configuration), it must be renewed. By default the token expiry time is 30 elapsed minutes from the time of its creation.

After the protected Domino resource has recreated the token, it calls the Login servlet again. If the error it received has the ID 1213, which means “Token expired,” it redirects to the first application URL again, otherwise it displays the error message.

The flow of control for the application login is displayed in Figure 136 on page 194. This technique and the Login servlet could be reused for any Domino application that contains servlets or JSPs.
In our example application the frameset is a Domino document. We did not use a Domino R5 frameset design element because of the greater flexibility of the code we can create in a document. For example, it can contain servlet or JSP parameters.

5.7 Ready to develop

The remaining chapters in this book are about the development and deployment of our sample application.

In Chapter 6, “Setup of the development and test environment” on page 197 we walk the reader through the installation and setup of VisualAge for Java.

In Chapter 7, “Servlets” on page 233 we discuss how to access servlets from Domino. We then describe the creation of a common class (BankingServletTemplate) to use as base for all our servlets in the example, and then we discuss the individual servlets:

- DominoLogin - where we show how to handle the expiration of tokens and reauthentication.
- PanelBuilder - where we show how to perform a full text search in a Domino database from a servlet and format the result for output.
- Search - also searches a Domino database, but here we show how the search arguments are picked up from values the user enters directly in a customized search form.
- SaveQuery - where we show how to pass search arguments over to our EJB for saving instead of actually performing a search.
In Chapter 8, “JavaServer Pages” on page 269 we first discuss JSPs in general. We show how to create a JavaBean with information from a Domino database and then pass it on to a JSP for presentation. JSP v1.1 supports the concept of tag libraries that helps even further in separating the application logic from the presentation logic in the JSP. We walk through a simple tag library example and then we describe the creation of the:

- ViewNavigator tag library

We show how we have used the ViewNavigator tag library in the two JSPs that are part of our user interface:

- Navigation Tree JSP
- Top Frame JSP

We also describe how we developed the servlet that performs the required checking and setup before invoking the JSPs:

- CallDominoJsp servlet

This servlet can be used as a generic servlet to forward to any JSP that requires a Domino session.

In Chapter 9, “Enterprise JavaBeans” on page 367 we first look at how to develop EJBs in VisualAge for Java and how to deploy them. We describe the implementation of the:

- Personalization EJB

We discuss how a Java client can call an EJB and we also discuss how a Domino agent can invoke the banking example Transfer EJB.

Finally, in Chapter 10, “Installation of the banking application” on page 401, we walk through installing the application files on the server, setting up the Web application, the EJB container and enterprise application, and securing the application.

5.8 Summary

In this chapter we have introduced you to the sample application we developed to illustrate Domino and WebSphere integration techniques.
Domino and WebSphere Together
Chapter 6. Setup of the development and test environment

This chapter describes how to install VisualAge for Java Enterprise Edition Version 3.5 and configure it to support development for the WebSphere and Domino Environment.

6.1 Installing VisualAge for Java Enterprise Edition V3.5

Insert the CD with the VisualAge for Java installation code. It should start automatically. If it does not, locate the setup program (usually in the VAJava directory or the VAJInstall directory, depending on how the product was packaged) and start it. You will be presented with a panel of options as shown in Figure 137.

Figure 137. VisualAge for Java welcome panel

Click the menu item Install Products. You will then be able to select the VisualAge product you wish to install, as shown in Figure 138 on page 198.
Click **Install VisualAge for Java**. You will be prompted for the language for the installation. The suggestion will be the language you have as default in your operating system setup. We kept the default suggestion of *English(United States)*. The Windows installer VisualShield program will start as shown in Figure 139 on page 199. Note that the installation panel (Figure 138) will still remain active, so you can return to it to install the Distributed Debugger or select other options later if you wish.
Chapter 6. Setup of the development and test environment

Click **Next**. You will be presented with the software license agreement. You can confirm that you are installing the selected product at the top of the licence agreement as shown in Figure 140.

![License Agreement](image)

**Figure 140. License agreement for VisualAge for Java with product version circled**
If you accept the terms for the VisualAge for Java Enterprise Edition, select the radio button marked **I accept the terms in the licence agreement** and click **Next**.

You will then be able to select the installation type, as shown in Figure 141.

![Installation type choices for VisualAge for Java](image)

*Figure 141. Installation type choices for VisualAge for Java*

Select **Custom** and click **Next**. You will then be able to select the features you want to install, as shown in Figure 142 on page 201.
Chapter 6. Setup of the development and test environment

Figure 142. Feature selection Panel showing selection of Domino access

When you select the button beside the feature, you can select to install the feature, install the feature and all subfeatures, or not to install the feature. You need to select:

- **Application Access Builders** - install the feature.
- **Domino Access Builder** - install the feature (You can only select this after you have selected Application Access Builders).
- **EJB/JSP Development Environment**.

**Note:** Domino Access Builder provides generic beans, based on the Domino Java classes from Lotus, plus a SmartGuide to create user-defined beans. The generic beans include wrapper classes for databases, forms, views, and other Domino design elements. It is not strictly necessary to install the Domino Access Builder to work with our examples in the book. However, installing it now saves you from a re-install in case you need it later on.

If you are building Java agents to run in the Domino server, you may also want to select the **Domino AgentRunner** feature. We did not use this for our examples.

Once you have made your selections, the panel should look like Figure 143 on page 202.
The features that will be installed have a disk image next to the selected feature; those that will not be installed will have a red X next to them. Click the button marked **Describe** to get an overview of each of the components if desired.

The button marked **Change** allows you to select an alternate destination for the installation. We retained the default (C:\Program Files\IBM\VisualAge for Java) installation directory, but you can select another location if you wish.

Once you have confirmed your selections, click **Next**. You will be prompted for the location of your development repository. The **Local** option should be used if you wish a stand-alone installation on a single computer. Select **Remote** if you are working in a team environment and need to share code on a server. Since we will illustrate stand-alone development, you should accept the default selection of **Local** and click **Next**.

You will be presented with an Installation confirmation panel. Click **Install** to start the copying of programs and files to the installation directory. If you are unsure of your choices or wish to confirm them again, click **Back** to view and possibly change them, and return to this panel to start the installation. You will see the installation progress panel shown in Figure 144 on page 203.
It will take several minutes to copy all the necessary files. When the installation is complete you will get an installation confirmation panel as shown in Figure 145.

Press **Finish** to complete the installation. The installation program will exit. If you left the original installation panel open (Figure 138 on page 198) you can exit it now or select other features.

It is not necessary to reboot your computer: VisualAge for Java V3.5 is ready for use.
6.2 Configuring VisualAge for Java V3.5 to support our examples

You may, if you wish, install the FixPak for VisualAge for Java. However, we found that it left the classes relevant to our installation (the servlet and Domino classes) unaffected. Thus for our installation the FixPak installation is optional. You should install it if you are using other class libraries and are unsure of whether they have been upgraded by the FixPak.

Open the VisualAge workspace by selecting Programs ->IBM VisualAge for Java for Windows V3.5 -> IBM VisualAge for Java.

The first time you start VisualAge, a dialog box will ask for the Administrator ID for VisualAge as shown in Figure 146. Simply enter the ID you usually use for your workstation.

![Figure 146. Administrator ID prompt when starting VisualAge](image)

The network login name is used to identify the owner of the code being developed and is relevant for the team edition of VisualAge. It is less relevant for us since we used the stand-alone version.

Again, once you have entered the product, you will be presented with a welcome panel as shown in Figure 147 on page 205.
Select **Go to the Workbench** and uncheck **Show this Window at startup**. All of the features listed are accessible from the menu or toolbar of the workspace.

Once you have started the workspace, select **File -> Quick Start** from the menu. From the Quick Start panel shown in Figure 148 on page 206, select **Features** in the left panel and then **Add Feature** from the right panel.
Click **OK** to view the list of features that you can add to the workspace. The Selection Required panel will be displayed, as shown in Figure 149 on page 207.
Figure 149. Selecting features to add to the VisualAge for Java workspace

From the Selection Required panel, select (by holding the Ctrl key and left-clicking the mouse):

1. **IBM EJB Development Environment 3.5**. This is necessary to create EJBs which we use in our sample.

2. **IBM WebSphere Test Environment 3.5**. This is necessary to test our code in the VisualAge Integrated Development Environment.

3. **Lotus Domino Java library 5.0.5**. This is to support our use of Domino classes in our example.

Do not select Sun Servlet API 2.1 since we need to import the Servlet 2.2 classes, as we explain in 6.2.1, “Upgrading the servlet API to V2.2” on page 209. Click OK to add the three selected features. VisualAge for Java will start to add the classes while checking their relationships with the existing class structure as shown in Figure 150 on page 208.
Once the process is complete, the new features will be added to your workspace as shown in Figure 151.

There are two more configuration changes we need to make before we are ready to create our Java code for the examples:

1. First, we need to update the installed servlet code (at the servlet 2.1 API level) to Servlet API level 2.2. This is necessary to install support for tag
libraries for servlets which was added at the 2.2 API level. This is explained in the section immediately following, 6.2.1, “Upgrading the servlet API to V2.2” on page 209.

2. Second, we need to set up a project to contain our classes for the example, as described in 6.2.2, “Creating a project for the Redbook example” on page 213.

Further configuration (for example, to create servlets, classes, and their methods) will be done in the chapters describing the components.

6.2.1 Upgrading the servlet API to V2.2

This step is necessary to support tag libraries. You can obtain the servlet V2.2 API classes from the Sun Web site or from a WebSphere V3.5 installation (with at least FixPak 2 installed). Since we had already installed WebSphere V3.5, we chose to use this installation as a source.

Ensure that the VisualAge for Java workspace is opened and locate the Servlet API Classes Project Folder. Highlight the folder and select File -> Import from the menu, as shown in Figure 152.

Figure 152. Updating the Servlet API Classes
On the next panel, shown in Figure 153 on page 210, select an import source.

![Figure 153. Selecting a jar file as an import source](image)

Select **Jar File** and click **Next**. Enter or select the specific jar file you wish to use. We chose the `servlet.jar` file in the WebSphere\AppServer\lib directory. Since we had previously installed FixPak 2 to WebSphere, this jar file is already at the V2.2 API level. The selection panel as we filled it in is shown in Figure 154 on page 211.
Chapter 6. Setup of the development and test environment

Figure 154. Specifying the WebSphere V3.5.3 servlet jar file to import to VisualAge for Java

Click **Finish** to start the import of the servlet V2.2 API. Again, you will see a progress dialog telling you that the classes are being analyzed. You may get prompts warning you that classes will be updated. Reply **Yes to all** to these warning messages. Reply **Yes to all (or Yes)** to any warnings that your old servlet API will be versioned since this is what you want in any case. Once the 2.2 API has been successfully imported, you will see the tag library package `javax.servlet.jsp.tagext` added to the Servlet API Classes project folder as we show in Figure 155 on page 212.
The Servlet API Classes folder is shown in Figure 156. This is a zoom in on the folder shown in Figure 155.

This completes the update of the Servlet API to 2.2.
6.2.2 Creating a project for the Redbook example

You can create a new Project folder to be added to the workspace using any of the following methods:

- Right-click on the project folders pane and select **Add -> Project ...**
- Select the menu item **Selected -> Add -> Project ...**
- Click the “**New project**” icon in the toolbar, as shown in Figure 157

Each of these selections will result in display of the SmartGuide Add Project panel shown in Figure 158.

![Figure 157. The 'new project' icon in the VisualAge for Java Toolbar](image)

Simply select **Create a new project named:** and fill in the name you wish to use for your project. We used Redbook Example for our code.

![Figure 158. Creating the example project](image)

Click **Finish**. Your (empty) project folder is ready for use. Since we will be using the WebSphere Samples banking classes included with the default
installation of WebSphere, we need to import these classes into our project folder.

Highlight the new project folder (Redbook Example) from the workspace and select Import ... from the File menu, as we did for the Servlet 2.2 API import. From the import panel, select Directory as your import source and click Next. Beside the Directory entry box at the top of the menu click the Browse... button and navigate to the WebSphere Banking sample classes. In our (default) installation this was:

```
C:\WebSphere\AppServer\hosts\default_host\WSsamples_app\servlets\WebSphere Samples\AccountandTransfer
```

Part of the file selection dialog is shown in Figure 159. Click OK to accept the path.

![Select a directory dialog](image)

**Figure 159. Selecting the Banking Example to import from WebSphere**

Note that, although we show importing the classes from a WebSphere installation on the same computer we installed VisualAge for Java on, you could also import from a WebSphere installation on another computer using a shared drive.
Make sure that the checkboxes for import of Java and resources are selected. Click **Finish** to start the import. Once again, you may receive messages about analyzing classes. When the importing process is finished you can confirm that the classes are in your project folder by opening it as we show in Figure 160.

**Figure 160. Showing the IBM Banking example imported into the example project**

You should see the four packages from the example:

- com.ibm.ejs.doc.account (expanded in our example in Figure 160).
- com.ibm.ejs.doc.accountBM
- com.ibm.ejs.doc.transfer
- WebSphereSamples.AccountAndTransfer.

This completes the configuration of VisualAge for Java V3.5. The next section is an optional introduction of how to work with VisualAge for Java when creating classes and methods.
6.3 Working with VisualAge

To acquaint you with VisualAge for Java, we now take you through the steps to create a class and a method for that class using the VisualAge SmartGuides (or wizards). If you already are experienced with VisualAge you can skip the rest of this chapter.

We illustrate how VisualAge for Java assists in the creation of Java objects with the creation of the BankingServletTemplate class (which we base our servlets on) and one of its methods. In principle, one could use a text editor to create the Java code, but this would forgo many of the advantages of the VisualAge for Java Integrated Development Environment (IDE). Among these benefits are the following features (this is not intended to be an exhaustive list):

- “Wizards” to automate the creation of Java code. These create code based on selection from higher level constructs, eliminating the error-prone creation of low level code for includes, parameters and the like.
- Real-time compile checking of code while in the editor. This is a common feature of most IDEs, and is one of the most compelling reasons to use one rather than ‘code to the metal.’
- Management of code entities and their grouping into classes, packages and projects.
- Support for team development.
- Integrated test environment.
- Support for code deployment into the production environment.

The reader can easily extend the list of advantages to using the VisualAge for Java IDE; it has to be used to be properly appreciated.

The following examples are intended to illustrate the use of the VisualAge for Java environment, not to show actual code development. For a description of the actual BankingServletTemplate class and its methods, see 7.3.1, “The BankingServletTemplate class” on page 242.

6.3.1 Creating the BankingServletTemplate class

As always, automated tools are not a substitute for a clearly expressed design which is based on a thorough understanding the business challenge; instead they complement a well organized development approach.
In our example, we wish to create a servlet class that will be used as a base class for other servlets. The reason we want to do this is so that we can create common “helper” methods, which will be inherited by the servlet classes we base on the BankingServletTemplate class.

We know in advance that we will base the new class on the existing servlet class; in addition we know that it will need access to Java utilities, Java input/output facilities and, of course, servlet classes. Finally, we may wish to add access to Domino objects from our new class.

Start the VisualAge for Java workspace by selecting Programs -> IBM VisualAge for Java for Windows V3.5 -> IBM VisualAge for Java. Navigate to the Redbook Example project we created in 6.2.2, “Creating a project for the Redbook example” on page 213, and select the project folder. This is shown in Figure 161.

![Figure 161. VA Java workspace - projects view - with our examples project selected](image)

First, we need to create a package to hold the new classes we will create for the example. We have chosen the package name of com.lotus.redbook.banking. To create a new package, we have three options which are equivalent (ensure that the project folder is highlighted). Use one of the following methods:
1. Select the project folder that the package will be added to, right-click and from the context menu, select **Add -> Package** ... as shown in Figure 162.

![Figure 162. Adding a package using the context menu](image)

2. Click on the **Add a package** icon in the toolbar. This icon is shown in Figure 163.

![Figure 163. The “Add a Package” icon on the VisualAge for Java toolbar](image)

3. Finally, we can select **Selected -> Add -> Package** from the menu bar.

Any of these three methods will display the SmartGuide - Add Package panel, as shown in Figure 164 on page 219.
Chapter 6. Setup of the development and test environment

Figure 164. Creating the com.lotus.redbook.banking package

Enter the new package name and click **Finish**. You can click **Next >** if you wish, but this will give you a panel to add users who can add classes to your package. Since we will illustrate with stand-alone development, we do not need this feature in our environment.

The new package will be created and added to your project folder. We are now ready to create the BankingServletTemplate class in this package.

By default, when you create the package, it will be highlighted when you are returned to the workspace. If it is not highlighted, select it now in preparation to adding the new class. Once again, there are three ways to add a class from the IDE:

- Use the context menu for the package and select **Add -> Class ...**
- Click on the **New Class** icon in the toolbar. (This icon is a letter “C” with a sparkle in the upper right; it is immediately to the right of the **New Package** icon.)
• Select **Selected -> Add -> New Class ...** from the menu.

In principle, we could also select Add a servlet (which would create the class based on the Java HTTPServlet class with `doGet` and `doPost` methods), but we wish to create a base class so we will use the class creator instead.

In any case, we are presented with the SmartGuide - Create Class panel shown in Figure 165.

![SmartGuide - Create Class panel](image)

*Figure 165. Creating the BankingServletTemplate class*

Note that the project and package are prefilled based on the selection from the workspace. Fill in the name of the class (BankingServletTemplate), and click the **Browse** button beside the Superclass entry; this will allow us to
select the **HTTPServlet** class to base our new class on, as shown in the Superclass panel in Figure 166.

![Superclass panel](image)

*Figure 166. Choosing HTTPServlet as the superclass for the BankingServletTemplate class*

In the pattern field type the first few letters of the class (we typed “https” in our case; the pattern is not case sensitive). The type name HttpServlet is selected since it is the first match. Click **OK** to select it as our superclass for the BankingServletTemplate class.

We returned to the Create Class panel shown in Figure 165 with the superclass set to `javax.servlet.http.HTTPServlet`. Click **Next >**.

You are presented with the Attributes panel shown in Figure 167 on page 222.
Click the Add Package .. button so you can add standard helper classes. You will be presented with the Import Statement panel shown in Figure 168 on page 223.
We want to add the \texttt{java.io} package and the \texttt{java.util} package. Once again, type the first few letters of the package (we show \texttt{java.i} in Figure 168; once again the pattern is not case sensitive); in our case, only \texttt{java.io} matched so it is automatically selected. Click \textbf{Add} to add an import statement for this package. Do not click \textbf{Close} yet since we still want to add the utility classes. Instead, alter the pattern (we simply backspaced over the terminal “i” and typed “u” in its place) to display \texttt{java.util} in the selection pane. Add this import statement and click \textbf{Close}. When you return to the Attributes panel the two statements:

\begin{verbatim}
    import java.io.*;
    import java.util.*;
\end{verbatim}

will be shown in the top pane. We could import other statements if we needed them, but these are all that we need for our example. Click \textbf{Finish} to create the class with the default attributes:

- Modifier: \texttt{public}
- \textbf{Create methods which must be implemented} (recommended)
- \textbf{Copy constructors from superclass} (recommended)

Your class will be created and added to the com.lotus.redbook.banking package as shown in Figure 169 on page 224.
The class has been created with two methods:
1. A BankingServletTemplate constructor method, and
2. The service method.

This completes the creation of the BankingServletTemplate class. We can now add methods to this class.

### 6.3.2 Creating the getAllServletParameters method

We need to have designed our method and have a clear definition of its function, interfaces, and visibility. For our new getAllServletParameters method, we want to scan the parameters sent to the servlet (either in a URL using the HTTP GET method or in a form using HTTP POST), fold them to lowercase, and return a Hashtable object with the parameters in it. We also want this to be a protected method. It would be possible (and fairly simple) to simply type in these definitions. However, we will use the VisualAge for Java workspace to automate some of this work so we can concentrate on the function.

Once again, we have three ways to create a method once we select and highlight the BankingServletTemplate class:

1. Select Add -> Method ... from the context menu by right-clicking.
2. Click Add Method (the letter “M” with a sparkle) from the toolbar.
3. Press Selected -> Add -> Method ... from the menu bar.
This will open the SmartGuide Create Method panel shown in Figure 170.

![Figure 170. Creating a new method SmartGuide](image)

Leave the (default) selection to Create a new method and click Next >. You do not need to overtype the attributes, insert parameters, or type the method name since we will do this in the next step.

The SmartGuide Attributes panel, shown in Figure 171 on page 226, will allow us to give the method a name, change the attributes of the new method and add properly typed parameters.
Since we want the return type to be `Hashtable` rather than `void`, click the **Browse** button beside the return type entry field (you could also type in the return type, but it is less error prone to select the case sensitive type from a list). This will open a Field type selection panel as shown in Figure 172 on page 227.
We typed “hash” in the entry field; there were three candidates including \texttt{Hashtable}. Select \texttt{Hashtable} and click \texttt{OK} to return to the attributes panel. We could have further narrowed the search by typing “hash”; this would only have shown \texttt{Hashtable}, but selecting from three candidates is just as easy.

Once we return to the attributes panel, the Return Type: entry field will have our selection, Hashtable.

On the same panel, select \texttt{Protected} for the access modifier and click the \texttt{Add} button beside the parameters pane to add parameters. This will open the \texttt{Parameters} panel as shown in Figure 173 on page 228.
Type the name of the parameter (that is, the name you will use in your code) in the Name entry field. Since we want to scan the HTTP request object, we named it `req`. Since an HTTP request object is not a primitive type, select **Reference Types** and type in the first few letters of the reference type. Once we typed `httpser`, we narrowed the search sufficiently to be able to identify and select `HttpServletRequest` as the parameter type. Click **Add** to put this parameter definition in the method code. You can continue this process to add more parameters, but we do not need to do this for the `getAllServletParameters` method. Click **Close** to return to the Attributes panel. The parameter `javax.servlet.http.HttpServletRequest req` will now be added to the parameters pane. If you wish to specify exceptions this method can throw, you can press **Next >** to do so. Again, these can be selected from a list if you wish. We want the default **Exception** exception to be thrown by the method so we do not specify anything here.
When you are satisfied with your changes, click **Finish** to create the new method. You will be returned to the workspace as shown in Figure 174, with the method added to the class and the generated code shown on the programmer’s pane. Note that we have changed the orientation of the panes on the VisualAge for Java workspace by choosing the menu item **Window - Flip Orientation** to place the programmer’s pane beside the project hierarchy pane.

![Figure 174. The getAllServletParmeters skeleton method](image)

We can now extend the code to add its logic before the closing brace. Note that you can format the code so that it has uniform indentation by right-clicking in the programmer’s pane and selecting **Format Code**. In addition, when you attempt to save the code, it will be syntax checked. We typed in the code in Figure 175 on page 230 with two typing errors.
The errors were:

- We mistyped the `String` modifier as `Strin`
- We misspelled the enumeration variable `esAllParameterNames` as `esAppParameter names`. Both errors were caught and potential corrections were suggested as shown in Figure 176 on page 231.
Figure 176. Compilation warnings and suggested corrections

We highlighted the first error in the top pane; the suggested correction was shown in the lower pane. In addition, the offending variable in the code is highlighted in the programmer's pane. If we click Correct, the suggestion will be applied to the code in error. The error will be removed from the error pane and the next error highlighted; clicking Correct again will correct this error and save the method since there are no more errors.

You can also click Save from the warning panel to allow you to save the method with errors. If you do so, the method name will be marked with a red “X” in the project hierarchy pane until you make acceptable changes.

Naturally, this process cannot detect logic errors or guide you in the selection of appropriate objects and their methods.

This completes our discussion of how to use VisualAge for Java to create methods within a predefined class.

We are now ready to create servlets, JSPs, and EJBs; they are described in the next three chapters.
6.4 Summary

In this chapter we have described how to install VisualAge for Java using a custom installation to add additional features.

After the installation we described how to add features for EJB development, WebSphere testing, and Domino access to the VisualAge workspace, and finally we gave a brief introduction to the use of VisualAge SmartGuides for creating classes and methods.
Chapter 7. Servlets

In this chapter we describe methods to invoke servlets from browsers and from Domino applications, and then describe the servlets in the banking example. You may wish to review the high level overview of servlets and their structure in 2.2.1, “Java servlets” on page 19.

In 2.4.2, “What WebSphere can add to a Domino R5 application” on page 33 we discussed the advantages of enhancing your Domino R5 applications with servlets, Java Server Pages (JSPs) and Enterprise Java Beans (EJBs).

There are two basic scenarios of using WebSphere in any Web application. You can access programs that are managed by WebSphere via:

- HTTP URLs from a Web browser
- Java programs/agents

Both ways are possible using Domino R5 design elements. In this chapter we discuss:

- How to invoke servlets and how to set up Domino and WebSphere for servlet support
- How to invoke an existing servlet from a Domino form

Then we describe the development of servlets; our examples also demonstrate how to access Domino objects from servlets.

7.1 Invoking servlets from Domino R5

Servlets can be invoked from a browser using a URL. Therefore, you can call a servlet from a Domino R5 application from any design elements that can call a URL; examples include frame sets, pages, forms, and agents.

7.1.1 Servlet URLs

If WebSphere is installed and uses the Domino R5 HTTP service, Domino R5 sends all URLs that point to defined servlet directories or to their subdirectories to WebSphere. The Domino R5 servlet directory is specified in the server document. However, as we explained in 3.7, “Connecting Domino to WebSphere V3.5 via a network connection” on page 76, the DSAPI plug-in intercepts all HTTP requests and decides based on the contents of the three (vhost, queues and rules) properties files whether to send a request to WebSphere. The setting of the servlet directory in the server document is thus not relevant unless you have an environment where some servlets will
be executed in WebSphere and others in the Domino servlet engine. Although this is technically possible, we do not recommend such a configuration since it would offer no advantages and would be more complex to troubleshoot and program for. This is especially so since the servlet engines are not on the same levels. Note, as well, that the default Domino servlet directory “/servlet” is, by default, in the WebSphere virtual host (“default_host”); one or the other of the directories would have to be changed for servlet references to fall through to Domino.

Servlets intended to be run under WebSphere must be in a defined virtual_host/uri servlet path so that the plug-in can recognize the destination as “WebSphere” and WebSphere itself can locate the resources. Often, you can use one of the pre-defined servlet directories.

WebSphere has a default path setting for the invocation of servlets using their class name. This is the path of the invoker servlet in the default_app Web application. When you run a servlet that you have not explicitly configured in the administrative domain, the WebSphere invoker servlet is called to load the servlet.

You can see how the default servlet path in WebSphere is defined in the administrative console in Figure 177.

![Figure 177. Default Servlet path in the WebSphere administrative console](image-url)
If you want to change the default path for servlets, you can change the setting in the Domino server document (although this will have no effect for WebSphere resources), as well as the setting for the invoker servlet in `default_app` in the WebSphere administrative console.

However, you should only use the default path for servlets when you are experimenting and exploring; in general there should be no need to change the servlet path. In any production application you would organize your servlets in an `enterprise application` in WebSphere. Following is an explanation of the difference between the default invocation of a servlet and a servlet that is defined as part of an enterprise application in WebSphere.

### 7.1.1.1 Default invocation of servlets

To run a servlet under WebSphere you simply need to deploy it to the servlet directory or one of its subdirectories. You do not need to do any configuration via the WebSphere administrative console: WebSphere will invoke the servlet if the URL points to a valid servlet class. Servlet URLs are case-sensitive and have to be spelled exactly like the class that is used to create the object in the servlet.

If the servlet is in a subdirectory of the WebSphere servlet directory, the directories have to be separated by a period (`.`). For example, if the servlet directory is `(virtual_host)/servlet` and you want to call a servlet that is in the `WebSphereSamples/AccountAndTransfer` subdirectory of the WebSphere servlet directory and that contains a class called `CreateAccount`, the URL to call this servlet would be:

```
http://yourhostname/servlet/WebSphereSamples.AccountAndTransfer.CreateAccount
```

After the servlet has performed its task, it can send back the code to display the next page or it can redirect to another URL.

As mentioned, default invocation of servlets should only be used for testing and not in production. Another servlet, called the invoker, has to be involved to load the servlet, thus affecting performance, and default invoked servlet cannot be secured in a WebSphere application.

### 7.1.1.2 Using another location for servlets and assigning aliases

If you do not want to put all your servlets into the standard servlet directory of WebSphere, you must create a Web application using the administrative console in WebSphere. For each Web application you specify the Web application path that is used by the user to call resources of the application. In addition, you can specify one or more aliases for each servlet. These can be used by the user to call the servlet instead of the servlet class name, and different initialization parameters can be passed to the servlet for different
aliases. An example is the showConfig servlet that we called in 3.7.8.4, “Testing OSE remote from Domino to WebSphere” on page 101. The name of the class file for the servlet is `ServletEngineConfigDumper.class`. However, the servlet has been defined as being part of the `examples` application with aliases of `showCfg`, `ShowCfg`, and `showConfig`, and can thus be called using any of these aliases instead of its class name. This set of aliases was created during creation of the WebSphere default application using the XML definition shown in Figure 178.

You will find that, if you search the file path (hosts\default_host\examples\servlets) as we described, there is no class file called `ServletEngineConfigDumper` in the expected directory (hosts\default_host\examples\servlets\com\ibm\websphere\examples\). This is because, rather than deploying it in the servlet directory, it has been included in the IBMWebAS.jar file that is part of the WebSphere classpath at WebSphere server initialization. Similarly, we deployed the Domino class files (in the NCSOW.jar file as shown in 3.9, “Adding the Domino ncsow.jar file to the WebSphere classpath” on page 106) as part of the WebSphere classpath so that we would not need to add this file to each Domino-enabled WebSphere application’s class file directory.

```
<servlet name="ShowConfig" action="create">
  <description>Displays the Current Servlet Engine Configuration</description>
  <code>com.ibm.websphere.examples.ServletEngineConfigDumper</code>
  <load-at-startup>false</load-at-startup>
  <uri-paths>
    <uri value="/showCfg"/>
    <uri value="/ShowCfg"/>
    <uri value="/showConfig"/>
    <uri value="/ShowConfig"/>
  </uri-paths>
</servlet>
```

Figure 178. The XML definition of the ShowCfg servlet for default application creation

### 7.1.2 Passing data to servlets in the URL

The easiest way to pass data to servlets is to create parameters in the URL that calls the servlet. The parameters are separated from the URL by a question mark (?), and the separator between parameters is an ampersand (&). The URL looks like:
If the URL is called from a Domino R5 Form or a Domino R5 Agent, the data that is passed to the servlet can be collected from Domino R5 fields. The actual sending of the data to the servlet can be coded in a button with the formula `document.forms[0].submit()`.

In the servlet class, the URL parameters are collected from the `HttpServletRequest` parameter of the `doGet` method. The URL parameter names are case-sensitive and the URL must contain all parameters you try to retrieve. The `doGet` method of the servlet would contain the following code to retrieve the parameters:

```java
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
...
  // Read input parameters from URL.
  String[] sQuery = req.getParameterValues("Query");
...
}
```

In our example, we implemented a helper method to extract the parameters passed in the URL and fold them to lower case before storing them in a hashtable object. The hashtable object was then used by servlets to extract the parameters:

```java
protected Hashtable getAllServletParameters(HttpServletRequest req) {
    Hashtable hServletParameters = new Hashtable();
    Enumeration esAllParNames = req.getParameterNames();
    while (esAllParNames.hasMoreElements()) {
        String sCurrentPar = (String)esAllParNames.nextElement();
        String[] asCurrentValues = req.getParameterValues(sCurrentPar);
        hServletParameters.put(sCurrentPar.toLowerCase(),
                               asCurrentValues[0]);
    }
    return hServletParameters;
}
```

However, there are three things to consider when passing data as part of URLs:

- The first is that URLs may not contain spaces or certain special characters. This requires the replacement of some characters and all spaces. When doing this, select the replacement carefully; changing the
replacement back to the original characters, which must be performed in
the servlet, is only possible if the original string does not contain the
replacement.

**Note:** JavaScript has an `escape()` function that parses a string and
converts special characters into their %xx equivalents (where, for
example, a blank space is %20) and an `unescape()` function to go the other
way. In Java the method to encode is `java.net.URLEncoder.encode()` and
(from jvm v1.3) there also is an unencode method named
`java.net.URLDecoder.decode()`.

- The second issue is that the parameters are case sensitive and your code
  either has to fold all cases to the same case or deal with variations of
case.

- The third issue with passing data in URLs is the limited length of the URL
  string. This limits the total amount of data that can be submitted in one
  URL. This is dependent on the browser you are using and its version. (For
  example, Netscape 4.04 allows about 2000 characters in one URL.) If the
  URL is too long, most browsers simply truncate it without reporting an
  error.

### 7.1.3 Posting data to servlets from Domino R5 forms

You can use Domino R5 forms to collect data you want to post to servlets.
This enables you to use most of the features of Domino R5 and the Domino
R5 designer in your Web application forms if you have decided to use a data
store other than Domino R5.

This technique makes it very easy to build a consistent user interface for
combined Domino R5 and WebSphere applications. In addition, it helps you
when you migrate parts of a Domino R5 application to WebSphere to achieve
better performance.

Figure 179 on page 239 shows a Domino form designed to be HTTP POSTed
to a servlet. We use a similar form in our sample application when the user
clicks on the Advanced Search button. The data in the fields will be sent in
the HTTP header of the request to the servlet in the form of name-value pairs.
The names are the Domino field names (rendered by the Domino HTTP task
and sent to the browser); the values are the values entered (or computed) in
the fields in the browser.
In our example, we used the methodology of collecting the parameters in a hashtable as shown above to ensure consistent handling of parameters regardless of how the servlet is accessed. Note that the parameters, whether sent in the URL or POSTed (and thus in the HTTP header), are always included as parameter objects of the request object presented to the servlet.

In the servlet class, the URL parameters are collected from the HttpServletRequest parameter of the doPost method. The URL parameter names are case-sensitive and the form must contain all parameters you try to retrieve.

### 7.2 Connecting to Domino using IIOP

If the WebSphere and Domino servers reside on the same physical machine, you can use the local Domino APIs (in Notes.jar) to connect to Domino from a WebSphere servlet. If WebSphere and Domino reside on different machines, you need to use the Domino CORBA/IIOP APIs (in NCSOW.jar).

In our examples we always create an IIOP session to Domino using the NotesFactory.createSession method, specifying the Domino server and the LtpaToken with the user’s authentication credentials. This ensures that the calls will work over a network and will work if Domino and WebSphere are on
the same computer. A prerequisite for these calls to work is to add the Domino NCSOW.jar file to the classpath of the WebSphere server (we showed how to add this in 3.9, “Adding the Domino ncsow.jar file to the WebSphere classpath” on page 106) or the application server.

Another reason to use the remote Domino classes as we have done is to eliminate the need to start and terminate a Notes thread for each thread of execution (that is, a method in our examples). Apart from the overhead of creating and tearing down Notes threads for every invocation of every method that calls a Domino object, this also means that the Domino objects will become inaccessible when the thread terminates. This is in contrast to using the remote methods where the Domino objects remain accessible until they are explicitly demarshalled by a \(<NotesObject>\).recycle() method call. In our Personalization EJB class we exploited this property to open an IIOP session that can be shared by multiple users. Having such a long-life IIOP session requires other considerations which we discuss in 9.3.1.2, “Considerations for keeping IIOP session open” on page 374.

### 7.2.1 SSL enabling of IIOP session

To create an IIOP session to Domino that is SSL-enabled, an extra parameter needs to be supplied for the `NotesFactory.createSession` method:

```java
String args[] = new String[1];
args[0] = "-ORBEnableSSLSecurity";
Session s = NotesFactory.createSession(
    host, args, user, pwd);
```

In addition, the Java classpath must be set up so that the class `TrustedCerts.class` is found. This class file is generated every time the diiop server starts and contains the server's certificate. It is normally found in the server's data directory\(\text{domino}\)/java.

However, at the time of writing there was no `NotesFactory.createSession` method that accepts the SSL argument together with the LTPA token. This is documented in SPR SBRN4XULHF and will be fixed in a future release.

**Note:** You can still SSL-enable the connection from the user's browser to the Domino and WebSphere machines while using SSO. It is the programmatic connection between the two machines that cannot be SSL-enabled if using SSO. Also, if you use the technique we exploit in the Personalization EJB class to open an IIOP session with a fixed user ID and password (and no LTPA token), you can SSL-enable such a session.
7.2.2 Using XML instead of IIOP

In certain situations where you simply need to get data from Domino to your servlets, the performance can be better if you use XML instead of IIOP. This is based on experience from the Lotus K-station development team achieved when reading Domino view data. In general, if you need to work with single documents for read or write, you should still use IIOP. We do not illustrate the use of Domino view data as XML in this book, but the idea is that you use the Domino URL argument ?ReadViewEntries to get a Domino view served as XML and then parse out the values you need. To learn more about using XML with Domino see the IBM Redbook *XML Powered by Domino*, SG24-6207.

We now move from the general discussion of servlets and look at how we implemented the classes we used for the servlets in our sample application.

7.3 Banking example helper classes

When we developed the different servlets for our sample application, we also developed several helper classes that are used by some or all of our servlets to manage their application context and perform housekeeping tasks.

We already touched a bit upon the BankingServletTemplate class when we discussed the use of VisualAge for Java as our development tool. The helper classes we will discuss in this section are:

- **BankingServletTemplate** class - with methods to put passed parameter into hash table, instantiation of an ApplicationInfo object, fetching a cookie and more
- **ApplicationInfo** class - used to store servlet context, like database path or user name for other servlets to access
- **DominoDocumentPanel** class - that performs a Domino full text search and builds a HTML page with the returned result set
- **SearchQuery** class - handles the formatting of input search parameters to the format used by Domino full text search

In 7.4, “The banking example servlets” on page 254 we cover the code of the actual servlets that use these helper classes.
7.3.1 The BankingServletTemplate class

This is the foundation class for all of the other classes. We do not invoke any of its methods directly; they are implicitly invoked as parent methods of the other methods.

7.3.1.1 Class variables for the BankingServletTemplate class

Table 6 lists the private class variables used by this class and its methods.

<table>
<thead>
<tr>
<th>Private class variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_sServletTitle</td>
<td>Current Servlet Title</td>
</tr>
<tr>
<td>m_sTopLevelURL</td>
<td>The URL to the content database (database “01”)</td>
</tr>
<tr>
<td>m.ejbhAccount</td>
<td>EJB object for user’s bank account</td>
</tr>
<tr>
<td>m.ejbhPersonalization</td>
<td>EJB object for user’s personalization data</td>
</tr>
</tbody>
</table>

7.3.1.2 Methods for the BankingServletTemplate class

This class has 15 methods:

1. BankingServletTemplate()
   This is a void constructor method.

2. generateError(HttpServletRequest, PrintWriter, String sModule, int iErrorId, String sMessage)
   This method invokes the setErrorMessage method of the ApplicationInfo class to store an error message along with the error number and the calling module.

Use Zip Files for Code Testing!

We have truncated the display of many of the following code samples since we wish to emphasize the main logic flow. For the most part, we have eliminated standard error “catch” blocks.

The user who wishes to try these samples should use the zip files which are posted on the Redbooks Web site rather than attempting to cut and paste from the code listings in this chapter. The source code for this chapter is in BankingSource.jar that is part of 5955java.zip. You can see how to get this file in Chapter K, “Using the additional Web material” on page 521.
3. `getAllServletParameters(HttpServletRequest)`
   This method creates a hashtable of all of the servlet's parameters. The name-value pairs are the parameter name in the servlet request object and the corresponding values.

4. `getAppInfo(HttpServletRequest)`
   This method instantiates a new ApplicationInfo object and returns it to the caller.

5. `getBrowser(HttpServletRequest, boolean bStdCSSFile)`
   Provided there is an application context in the servlet request parameter, this method gets and returns either the server name in the URL plus a path to the applicable cascading style sheet for the browser in Domino configuration database (if bStdCSSFile is true) or simply the browser type.

6. `getCookie(HttpServletRequest, PrintWriter, String sCookieName)`
   This method simply scans the cookies sent by the browser looking for a match to the input sCookieName parameter. In our application, we used this to search for the LtpaToken Cookie to retrieve the user’s authentication credentials.

7. `getDbPath(HttpServletRequest)`
   This returns the current top level URL for the request, including the Domino database being accessed.

8. `getDebugFlag(HttpServletRequest)`
   This simply retrieves the current debug flag setting. This was originally set by the Login doGet method based on a servlet parameter.

9. `getHomeObject(ApplicationInfo, String sBeanName)`
   This method opens the home method for the EJB supplied in the parameter sBeanName. Figure 180 on page 244 shows the initialization and retrieval of the account home method of the Account EJB.
protected Object getHomeObject(ApplicationInfo objApp, String sBeanName) {
    if (sBeanName.compareTo("Account") == 0 & m_ejbhAccount != null) {
        // account home exists
        return m_ejbhAccount;
    } else if (sBeanName.compareTo("Personalization") == 0 &
              m_ejbhPersonalization != null) {
        // personalization home exists
        return m_ejbhPersonalization;
    } else {
        // Get home object.
        Hashtable hEnv = new Hashtable(2);
        hEnv.put(Context.INITIAL_CONTEXT_FACTORY,
                 "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
        hEnv.put(Context.PROVIDER_URL, "iiop://" + objApp.getEJBServer()
                 + ":900");
        try {
            // Create the initial context.
            Context ctx = new InitialContext(hEnv);
            // Get the home object.
            Object objHome = null;
            if (ctx != null) {
                objHome = ctx.lookup(sBeanName);
            }
            if (sBeanName.compareTo("Account") == 0) {
                if (objHome != null) {
                    m_ejbhAccount =
                    (AccountHome) javax.rmi.PortableRemoteObject.narrow(
                                    objHome, AccountHome.class);
                }
                return m_ejbhAccount;
            }
        } finally {
            // removing home object.
            if (objHome != null) {
                objHome = null;
            }
        }
    }
}

Figure 180. getHomeObject method of PanelBuilder class - getting the account home

Figure 181 on page 245 shows the retrieval of the home object of the Personalization EJB. Note that the objects are stored as class variables so that any future calls can simply return the home objects; the method returns the stored or the located EJB home object requested. If the EJB home object could not be found, null is returned to the requesting method.
10. getPrintWriter(HttpServletResponse)
   This method gets a PrintWriter object for the response and formats the
   header of the output. It also inserts the current servlet’s name into the title
   of the response.

11. getServletParameter(Hashtable hParameters, String sParameterName)
   This method scans the hashtable of servlet parameters created in (3) and
   returns the value of the one matching the input string.

12. getServletPath(HttpServletRequest)
   This returns the current value of the servlet path using the
   ApplicationInfo’s getServletPath method.

13. getServletTitle()
   This returns the servlet’s title (stored as a method variable for the
   BankingServletTemplate class).
14. `getUserName(HttpServletRequest)`
   This gets the username originally stored by the Login doGet method.

15. `printHTML(PrintWriter, String sHTMLCode, boolean bStartBody, boolean bEndBody)`
   This method outputs a line of HTML to the input PrintWriter. It can optionally start and/or end the HTML body based on the two flags.

7.3.2 ApplicationInfo class
This class retains and services requests for servlet context information by the other servlets. This information includes current user, database path, EJB server, error message handling, and control of debugging.

7.3.2.1 Class variables for the ApplicationInfo class
The class variables for this class are shown in Table 7.

<table>
<thead>
<tr>
<th>Private class variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m_sServletPath</code></td>
<td>The URL to the servlet.</td>
</tr>
<tr>
<td><code>m_sDbPath</code></td>
<td>The database part of the URL.</td>
</tr>
<tr>
<td><code>m_sEjbServer</code></td>
<td>The server that was specified as the EJB server.</td>
</tr>
<tr>
<td><code>m_sBrowser</code></td>
<td>Current Browser (IE, Netscape 4 or Netscape 6). Used to select Cascading Style Sheet.</td>
</tr>
<tr>
<td><code>m_sUserFullName</code></td>
<td>The current user’s name as retrieved from Domino.</td>
</tr>
<tr>
<td><code>m_sErrorMessage</code></td>
<td>Error message set by other method.</td>
</tr>
<tr>
<td><code>m_iErrorId</code></td>
<td>Error number corresponding to <code>m_sErrorMessage</code>.</td>
</tr>
<tr>
<td><code>m_bErrorReloaded</code></td>
<td>Set if error due to token expiry to allow login rather than error exit.</td>
</tr>
<tr>
<td><code>m_bDebugFlag</code></td>
<td>Set to true if applications should produce debug information.</td>
</tr>
</tbody>
</table>
7.3.2.2 Methods for the ApplicationInfo class

This class has 14 methods:

1. ApplicationInfo()
   This is a void constructor class that invokes its parent class
   (HTTPSessionBindingListener).

2. activateDebugging()
   This sets the Debug flag – method variable (m_bDebugFlag) – to true.

3. getBrowser(boolean bStdCSSFile)
   This returns the browser ID (stored in the member variable m_sBrowser) if
   the input flag is false; otherwise it returns a path to the Cascading Style
   Sheet that matches the browser definition in the Domino Published
   Documents database (intra_01.nsf).

4. getDbPath(String sDbNumber)
   This returns the path to the current database, qualified by the database
   number. Thus a call with sDbNumber set to “01” would return the path to
   the intra_01.nsf database, including the file name of the database.

5. getDebugFlag()
   This returns the current value of the method variable m_bDebugFlag. If
   true, debugging is enabled in the other classes.

6. getEjbServer()
   This method returns the EJB server.

7. getErrorMessage()
   This returns the current error message (stored in the method variable
   m_sErrorMessage) and clears the currently stored error message and
   error ID. In the special case of error number “1213” (token expired), it sets
   a flag used to force a login of the user.

8. getServletPath()
   This returns the current value of servlet path stored in the method variable
   m_sServletPath. This was stored by the Login doGet method.

9. getUserName()
   This returns the current value of the user name stored by the Login doGet
   method in the method variable m_sUserFullName.

10. setErrorMessage(String sModule, String sMessage, int iErrorID)
    This sets the class variables m_sErrorMessage and m_iErrorId to the
    supplied values of invoking module, message, and error number. The
    message is stored as an HTML string for later retrieval. If there is already
    an error message stored, it is concatenated with the input values.
11. `setPathValues(String sServletPath, String sDbPath, String sEjbServer, String sBrowser)`
   This simply stores the supplied parameters into class variables `m_sServletPath`, `m_sDbPath`, `m_sEjbServer` and `m_sBrowser`.

12. `setUserName(String sFullName)`
   This simply stores the user’s name as retrieved in the Login `getUserName` method in the method variable `m_sUserFullName`.

13. `valueBound(HttpSessionBindingEvent)`
   This method is necessary for servlet session objects.

14. `valueUnbound(HttpSessionBindingEvent)`
   This method is necessary for servlet session objects.

### 7.3.3 DominoDocumentPanel class
This class executes a Domino full text search and builds an HTML panel of database links for the frame on the right side of the frameset containing the user’s database favorite links. It services requests from the `PanelBuilder` class and the `Search` class.

#### 7.3.3.1 Class variables of the DominoDocumentPanel class
This class has four private class variables as shown in Table 8.

<table>
<thead>
<tr>
<th>Private class variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m_sFontStart</code></td>
<td>Definition of font face for panel. Set to Sans Serif (Arial or Helvetica) <code>&lt;FONT FACE=&quot;&quot;MS Sans Serif, Arial, Helvetica&quot; SIZE=1&gt;</code></td>
</tr>
<tr>
<td><code>m_sFontEnd</code></td>
<td>End of font tag (<code>&lt;/FONT&gt;</code>).</td>
</tr>
<tr>
<td><code>m_objApp</code></td>
<td>Current <code>ApplicationObject</code> object.</td>
</tr>
<tr>
<td><code>m_sCurrentHost</code></td>
<td>Current host from URL.</td>
</tr>
</tbody>
</table>

#### 7.3.3.2 Methods of the DominoDocumentPanel class
This class has five methods:

1. `DominoDocumentPanel(ApplicationInfo, String)`
   This constructor method simply initializes the class variables of the class.

2. `doSearch(String sQuery, int iMaxResults, int iSortOptions, String sDisplayOptions, String sLtpaToken, boolean bShowFound)`
   This method creates an IIOP session to the Domino contents database as shown in Figure 182 on page 249.
The method then executes a full text search based on the query passed to it in the invoking message. Each document returned from the full text search is passed to the `getDocumentLink` method to be parsed into a line of HTML (optionally with an abstract and image, if requested by the user options and present in the document). Each line of HTML is created as a list element of an unordered list; the entire unordered list is returned as a string of HTML to the invoking method as shown in Figure 183 on page 250.
Figure 183. doSearch Domino full text search

3. getCurrentDbPath()
   This returns the database path to the Domino contents database in a string.

4. getCurrentHost()
   This returns the current host (from the URL) in a string.

5. getDocumentLink(boolean bAbstract, boolean bImage, Document docLinkTo)
   This helper method resolves each returned link from the full text search of the doSearch method into a line of HTML with a reference to the document.
and, if requested, the associated document abstract and image as shown in Figure 184.

```java
private String getDocumentLink(boolean bAbstract, boolean bImage,
Document docLinkTo) {
    String sHTMLCode = "";
    try {
        if (!docLinkTo.isValid()) {
            return "";
        }
        // get values from document
        String sName = docLinkTo.getItemValueString("tContLinkText");
        if (sName == null) sName = "- Click here -";
        // create HTML link
        sHTMLCode = sHTMLCode.concat("<B><A HREF=" +
            getCurrentHost() + "/" + getCurrentDbPath() +
            "/viref_foContent$ByID/" + docLinkTo.getUniversalID() +
            "?OpenDocument\" TARGET="Middle\">" + sName +
            "</A></B>");
        String sAbstract = docLinkTo.getItemValueString("tContAbstract");
        if (bAbstract && sAbstract != null) {
            // abstract
            sHTMLCode = sHTMLCode.concat("<BR>\n");
            String sImageName =
                docLinkTo.getItemValueString("tContImageName");
            if (bImage && sImageName != null) {
                String sImageDocName =
                    docLinkTo.getItemValueString("tContImageDocName");
                if (sImageDocName != null) {
                    // image in document within db
                    sImageDocName = "http://" + getCurrentHost() + "/" +
                        getCurrentDbPath() +
                        "/viref_foFiles$ByName/" + sImageDocName + "/$File/";
                } else {
                    sImageDocName = "";
                }
                // image
                sHTMLCode = sHTMLCode.concat("<img src=" + sImageDocName +
                    " align=left hspace=10>")));
            }
        }
    }
    sHTMLCode = sHTMLCode.concat(sAbstract);
}
```

Figure 184. The getDocumentLink method of the DominoDocumentPanel class
7.3.4 SearchQuery class

This is a set of helper methods that format the strings used to qualify the Domino full text search for documents.

7.3.4.1 Class variables for the SearchQuery class

There are two private class variables as shown in Table 9.

Table 9. Private class variables for the SearchQuery class

<table>
<thead>
<tr>
<th>Private class variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_sQuery</td>
<td>The current query string being executed.</td>
</tr>
<tr>
<td>m_bOr</td>
<td>If true, use “OR” to specify any one search entry for full text search, otherwise “AND” them to specify that all must be found.</td>
</tr>
</tbody>
</table>

7.3.4.2 Methods of the SearchQuery class

This class has five methods:

1. `SearchQuery(boolean bOr)`
   
   This constructor method takes an input boolean and stores it in the private method variable m_bOr. This flag determines whether the user wants any of the search fields or all of the search fields in the result.

2. `addDateField(String sFieldName, String sYear, String sMonth, String sDay, boolean bBefore, String sLtpaToken)`
   
   This method takes the input date parameters, edits their validity and formats a date search string according to the Internationalization settings of the target Domino server. The date parameters are stored in the m_sQuery private method variable by calling the addField method. Part of the method’s code is shown in Figure 185 on page 253; the remaining code takes the internal date string sDate and calls addField to add the input field sFieldName, the date string, and an operator (greater than (>) or less than (<)). Any exceptions result in the method returning a false value.
public boolean addDateField(String sFieldName, String sYear, String sMonth, String sDay, boolean bBefore, String sLtpaToken) {
    try {
        // test if parameters are valid
        if (sFieldName.compareTo("") == 0) {
            // cannot search for date without field name
            return true;
        }
        int iYear = convertStringToInt(sYear);
        int iMonth = convertStringToInt(sMonth);
        int iDay = convertStringToInt(sDay);
        if (((iYear == 0) || (iMonth == 0) || (iDay == 0)) {
            // no date search added
            return true;
        } else if (((iYear == -1) || (iMonth == -1) || (iDay == -1)) {
            // invalid date parameters
            return false;
        }
        // get domino session
        Session sesCurrent = NotesFactory.createSession((String)null, sLtpaToken);
        International international = sesCurrent.getInternational();
        // calculate date string according to server settings
        String sDateSep = international.getDateSep();
        String sDate = "";
        if (international.isDateDMY()) {
            sDate = sDay + sDateSep + sMonth + sDateSep + sYear;
        } else if (international.isDateMDY()) {
            sDate = sMonth + sDateSep + sDay + sDateSep + sYear;
        } else if (international.isDateYMD()) {
            sDate = sYear + sDateSep + sMonth + sDateSep + sDay;
        } else {
            // cannot find date settings of the server
            return false;
        }
        DateTime ndtDate = sesCurrent.createDateTime(sDate);
    }
}

Figure 185. addDateField method of the SearchQuery class

3. addField(String, String, String)
   This method takes fields passed from the other methods and adds them to
   the Query string held in m_sQuery.

4. convertStringToInt(String sValue)
   This method simply takes an input string and converts it to an integer
value. If the string does not contain a valid integer, it returns -1. This method is used by the addDateField method to parse the values the user entered for year, month and date.

5. getQuery()

This method is used to return the Query string built in m_sQuery by the other methods; it is qualified by a default query that ensures that only documents created using the form fo_content are requested from the contents database.

This completes our discussion of the helper classes. We are now ready to look at the servlets in our application.

### 7.4 The banking example servlets

In this section we describe the actual servlet code. See 5.3, “Servlets and JSPs” on page 186 for a description of the overall flow between the servlets.

You may also want to refer to 9.3, “The Personalization EJB” on page 373 for a description of the Personalization methods referred to in this section.

The servlets we discuss in this section are:

- DominoLogin
- PanelBuilder
- Search
- SaveQuery

#### 7.4.1 The DominoLogin servlet

This servlet manages the initial login and is used to intercept errors so that a person whose session has expired will be able to login again without getting an error message. For an overview of the DominoLogin servlet’s context on the overall application, see 5.6, “Flow of control for Web users” on page 192.

#### 7.4.1.1 DominoLogin servlet methods summary

This class has two methods:

1. `doGet(HttpServletRequest, HttpServletResponse)`

   This servlet logs a person into the Domino environment and stores user values in a servlet session object. This in turn can be queried by the other application components to retrieve context information. The application code is shown in Figure 186 on page 256. We show only the main processing flow.
The method updates or creates an ApplicationInfo object based on the servlet context. In the block commented “Set application settings”, we store the servlet path (that is, URL without any parameters: the host name plus path), the Domino database name and the user’s browser type.

We then connect to Domino using the method getUserName (shown in Figure 187 on page 257). If successful, we then either return with a success message or redirect to the actual URL the user was trying to open.
if (httpSes != null) {
    objApp = (ApplicationInfo) httpSes.getAttribute ("AppInfo");
    if (objApp == null) {
        objApp = new ApplicationInfo();
        httpSes.setAttribute ("AppInfo", objApp);
    }

    // Set application settings
    Hashtable hParameters = getAllServletParameters (req);
    objApp.setPathValues(req.getServerName() + req.getRequestURI(),
                        getServletParameter (hParameters, "Db"),
                        getServletParameter (hParameters, "EjbServer"),
                        getServletParameter (hParameters, "Browser");

    // Connect to Domino
    String sUserName = getUserName (req, out);
    objApp.setUserName (sUserName);

    String sErrorMessage = objApp.getErrorMessage();
    String sDebugFlag = getServletParameter (hParameters, "Debug");
    if (sDebugFlag.compareTo (""") != 0) objApp.activateDebugging();
    if (sErrorMessage != null) {
        // Display error message
        printHTML (out, sErrorMessage, true, true);
    } else if (sDebugFlag.compareTo ("1") == 0) {
        // Display values (for debugging)
    } else {
        String sNextURL = getServletParameter (hParameters,
                                             "NextURL");
        if (sNextURL.compareTo ("") == 0) {
            printHTML (out, "Login was successful.", true, true);
        } else {
            printHTML (out, "Redirecting ...", true, true);
        }
    }
}

Figure 186. The body of the doGet Method of the DominoLogin class

Note that, apart from setting error messages, this is the only method that
sets values in the ApplicationInfo object.

2. getUserName (HttpServletRequest, PrintWriter)

This helper method simply opens an IIOP session to the Domino server
using the LTPA token to authenticate. If the session is successfully
created, the users’s identity is read from the Domino session method getUserName as shown in Figure 187. We then destroy the Domino IIOP session and return the user’s Name.

```java
protected String getUserName(HttpServletRequest req, PrintWriter out) throws IOException {
    String sLtpaToken = getCookie (req, out, "LtpaToken");
    Session sesCurrent = null;
    try {
        sesCurrent = NotesFactory.createSession (req.getServerName(), sLtpaToken);
        String sUserName = sesCurrent.getUserName();
        if (sesCurrent != null) sesCurrent.recycle();
        return sUserName;
    } catch (NotesException e) {
        generateError(req, out, "DominoLogin: doGet", e.id, e.text);
        return "";
    } catch (Exception e) {
        generateError(req, out, "DominoLogin: doGet", 0, e.getMessage());
        return "";
    }
}
```

*Figure 187. The getUserName method of the DominoLogin class*

### 7.4.2 PanelBuilder servlet

This servlet builds a panel of document links based on the user’s stored personal criteria, like content in title or abstract, category, or age of document. It uses the DominoDocumentPanel class to execute a query and format the panel. For an overview of its function, see 5.3.2, “The PanelBuilder servlet” on page 187.

#### 7.4.2.1 Class variables for the PanelBuilder servlet

This class has no private class variables.

#### 7.4.2.2 Methods of the PanelBuilder servlet

This class has three methods:

1. `doGet(HttpServletRequest, HttpServletResponse)`
   
   This servlet instantiates the personalization EJB (see 9.3, “The Personalization EJB” on page 373 for the class files in the personalization
EJB) as shown in Figure 188. We have removed the debug statements in this and all other examples for simplicity.

```java
public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
    PrintWriter out = getPrintWriter(res);
    HttpSession httpSes = req.getSession(true);
    ApplicationInfo objApp = null;
    try {
        if (httpSes != null)
            objApp = (ApplicationInfo) httpSes.getAttribute("AppInfo");
        if (objApp != null) {
            Hashtable hParameters = getAllServletParameters(req);
            String sLtpaToken = getCookie(req, out, "LtpaToken");
            if (sLtpaToken == null) {
                generateError(req, out, "PanelBuilder:doGet", 1213,
                              "User not logged in");
            } else {
                PersonalizationHome ejbhPersonalization =
                        (PersonalizationHome) getHomeObject (objApp, "Personalization");
                if (ejbhPersonalization != null) {
                    Personalization ejbPersonalization =
                            ejbhPersonalization.create();
                    Hashtable hPersonalization =
                            ejbPersonalization.getPersonalization(
                                    req.getServerName(), objApp.getDbPath("02").getTrue());
                    if (hPersonalization == null) {
                        // try again, because null is also returned if the session was
                        // dropped by the Domino server
                        hPersonalization =
                                ejbPersonalization.getPersonalization(
                                        req.getServerName(), objApp.getDbPath("02"),
                                        true);
                    }
                    // Note that, if we get a null returned from the getPersonalization method of
                    // the Personalization EJB, we retry the method invocation a second time.
                    // This is because the EJB will return null if the Domino session is no longer
                    // valid; calling it again will force an IIOP reconnection to the Domino server.
                    // The session could have dropped because it timed out (by default, after an
                    // hour Domino recycles IIOP sessions), or because the DIIOP task or
                    // Domino itself was restarted.
                }
            }
        }
    }
}
```

Figure 188. The doGet method of the PanelBuilder Class - accessing personalization EJB
The returned personalization information is then checked to see whether the user's personalization information was created; if so, we create account keys in preparation for creating their checking and savings accounts as shown in Figure 189.

```java
// the hashtable now has values we can use to search
// if the user has never set up a personalisation document
// these will be defaulted read the values from the returned hashtable
// to start the search and return the results to the browser
DominoDocumentPanel objPanel = new DominoDocumentPanel(objApp,
req.getServerName());
String sPanelContent = objPanel.doSearch(
(String) hPersonalization.get("sstring"),
((Integer) hPersonalization.get("maxresults")).intValue(),
getSortBy ((String) hPersonalization.get("sortoptions")),
(String) hPersonalization.get("displayoptions"),
LtpaToken,
false);
if (((Boolean)hPersonalization.get
("docreateaccounts")).booleanValue()) {
    // creating bank accounts
    Integer iChecking =
        (Integer)hPersonalization.get("checking");
    Integer iSaving =
        (Integer)hPersonalization.get("savings");
    AccountKey objKeyChking =
        new AccountKey(iChecking.longValue());
    AccountKey objKeySvgs =
        new AccountKey(iSaving.longValue());
}
```

Figure 189. Use personalization results to start document search and create accounts

We then use the account key objects to call the Account EJB and create the user's bank accounts (defaulted to zero balances). Then, whether or not the accounts needed to be created, we output the results of the search query returned from the doSearch method of the DominoDocumentPanel class using the printHTML method (a method of the BankingServletTemplate) to send them to the browser, as shown in Figure 190 on page 260.
AccountHome ejbhAccount = (AccountHome)
getHomeObject (objApp, req.getServerName(),"Account");
if (ejbhAccount != null) {
    // Create the customer's Checking and Savings Account
    ejbhAccount.create(objKeyChking, 2, 0);
    ejbhAccount.create(objKeySvgs, 1, 0);
} else {
    generateError(req, out, "PanelBuilder:doGet", 0,
    "Could not create home object.");
}
if (sPanelContent == null) {
    // error during FT search
    generateError(req, out, "PanelBuilder:doGet", 0,
    "No documents found!");
} else {
    printHTML(out, getDocStart(req), false, false);
    printHTML(out, sPanelContent, false, true);
}
} else {
    if (getDbPath(req).compareTo("") != 0) {
        generateError(req, out, "PanelBuilder:doGet", 1213,
        "User not logged in");
    } else {
        printHTML(out, "No application info available!", true,
        true);
    }
}
} catch (Exception e) {
    generateError(req, out, "PanelBuilder:doGet", 0, e.getMessage());
    System.out.println(e.getMessage());
}
} } catch (Exception e) {
    generateError(req, out, "PanelBuilder:doGet", 0, e.getMessage());
    System.out.println(e.getMessage());
}

Figure 190. Returning the results of the search query to the browser
2. `getDocStart(HttpServletRequest)`
   This method is used to initialize the HTML output. It simply inserts the appropriate style sheet for the user's browser using the `getBrowser` method of the `BankingServletTemplate` class.

3. `getSortBy(String sSortBy)`
   This method simply translates the sort criteria (“NEW”, “OLD” or “RELEVANCE” – the default) into the values used by the Domino database full text property values.

### 7.4.3 Search servlet

This servlet class performs a search using the `doSearch` method of the `DominoDocumentPanel` class, using the servlet's parameters (created either from a form or directly entered) to qualify the search. It is analogous to the `PanelBuilder` servlet; the difference is that this servlet executes a search query for the user’s favorites based on input while the `PanelBuilder` servlet uses the user's stored settings. This servlet has no private class variables.

For an overview of the Search servlet's functions, see 5.3.3, “The Search servlet” on page 187.

#### 7.4.3.1 Methods of the Search servlet

This class has six methods:

1. `doGet(HttpServletRequest, HttpServletResponse)`
   This servlet method retrieves a query (as a single string) from its URL and executes it using the `doSearch` method of the `DominoDocumentPanel` class as shown in Figure 191 on page 262.
public void doGet(HttpServletRequest req, HttpServletResponse res)  
throws ServletException, IOException {  
  PrintWriter out = getPrintWriter (res);  
  HttpSession httpSes = req.getSession (true);  
  ApplicationInfo objApp = null;  
  if (httpSes != null) objApp = (ApplicationInfo)httpSes.getAttribute  
  ("AppInfo");  
  if (objApp != null) {  
    Hashtable hParameters = getAllServletParameters (req);  
    String sLtpaToken = getCookie (req, out, "LtpaToken");  
    getDbPath(req);  
    if (sLtpaToken == null) {  
      generateError (req, out, "Search:doGet", 1213,  
      "User not logged in");  
    } else {  
      // build the search string  
      SearchQuery objQuery = new SearchQuery (false);  
      objQuery.addField ("", getServletParameter (hParameters,  
            "Query"), "");  
      // display search results  
      DominoDocumentPanel objPanel = new DominoDocumentPanel  
          (objApp, req.getServerName());  
      String sPanelContent = objPanel.doSearch(objQuery.getQuery (),  
          getMaxResults (hParameters), getSortBy (hParameters),  
          "1", sLtpaToken, true);  
      if (sPanelContent == null) {  
        generateError (req, out, "Search:doGet", 0,  
            "Error during full text search");  
      } else {  
        printHTML (out, getDocStart(req), false, false);  
        printHTML (out, sPanelContent, false, true);  
      }  
    }  
  }  
}

Figure 191. The doGet method of the Search servlet

2. doPost(HttpServletRequest, HttpServletResponse)

This method is very similar to the doGet method except that it retrieves its
search query string from the HTTP header as a result of a POST from a
Domino form. It also extracts the query as a set of tokens so that the field
content can be parsed and validated by the methods of the SearchQuery
class described in 7.3.4, “SearchQuery class” on page 252.
3. `getDocStart(HttpServletRequest)`
   This method simply initializes the HTTP output with the cascading style sheet appropriate to the user's browser.

4. `getMaxResults(Hashtable hParameters)`
   This method retrieves the user's specification of the maximum number of documents they wish. If the user did not specify this value, it returns a value of 100. Note that the user can specify more than 100 documents; it is only if they do not specify the maximum number or they specify a non-numeric value that 100 is used.

5. `getQueryString(Hashtable hParameters, String sLtpaToken)`
   This method parses the input query string from the (tokenized) list of search parameters from the servlet parameters and renders them in a format suitable to be passed to the Domino full text search.

6. `getSortBy(Hashtable hParameters)`
   This method parses the ASCII sort parameters (“NEW”, “OLD”, or “RELEVANCE”) supplied in the servlet's parameters and returns them in integer format ready to be submitted to the Domino full text search.

### 7.4.4 SaveQuery servlet

This servlet class accepts a query posted from a form and saves it to the user's personalization data using the Personalization EJB (described in 9.3, “The Personalization EJB” on page 373). It is analogous to the Search servlet; the difference is that this servlet parses and stores a search query rather than executing it; the PanelBuilder servlet will use these stored settings. When the servlet has stored the settings, it redirects the browser to the PanelBuilder servlet to display the results of the query. This servlet has no private class variables.

For an overview of the Search servlet's functions, see 5.3.4, “The SaveQuery servlet” on page 188.

### 7.4.4.1 Methods of the SaveQuery servlet

This class has three methods:

1. `doPost(HttpServletRequest, HttpServletResponse)`
   This method parses the query using the `getQueryString` method and, if it can successfully be parsed to a valid query, stores it in the user's personalization document using the `writePersonalization` method of the Personalization EJB. We show the code for this method in Figure 192 on page 264 to the point of writing the personalization data; the remaining code (shown in Figure 193 on page 265) simply rebuilds the URL to
redirect the user’s browser to the PanelBuilder servlet to test and display the stored query data.

```java
public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    PrintWriter out = getPrintWriter (res);
    HttpSession httpSes = req.getSession (true);
    ApplicationInfo objApp = null;
    if (httpSes != null) objApp =
        (ApplicationInfo)httpSes.getAttribute ("AppInfo");
    try {
        if (objApp != null) {
            Hashtable hParameters = getAllServletParameters (req);
            String sLtpaToken = getCookie (req, out, "LtpaToken");
            getDbPath(req);
            if (sLtpaToken == null) {
                generateError (req, out, "SaveQuery:doPost", 1213,
                    "User not logged in");
            } else {
                // build the search string
                String sQuery =
                    getQueryString (hParameters,
                        req.getServerName(), sLtpaToken);
                if (sQuery == null) {
                    generateError (req, out, "SaveQuery:doPost", 0,
                        "Error building query string");
                } else {
                    // get the personalization EJB
                    PersonalizationHome ejbhPersonalization =
                        (PersonalizationHome) getHomeObject (objApp,
                            "Personalization");
                    if (ejbhPersonalization != null) {
                        Personalization ejbPersonalization =
                            ejbhPersonalization.create();
                        int iChecking =
                            ejbPersonalization.writePersonalization (req.getServerName(),
                                objApp.getDbPath("02"),
                                sQuery, getMaxResults (hParameters),
                                getServletParameter (hParameters, "tSortBy"),
                                "1");
```

Figure 192. the doPost method of the SaveQuery class - saving the user’s query information

Note that if we did not get a checking account number returned from the writePersonalization method of the Personalization EJB, we call one more time to allow it to try to set up a new Domino IIOP session since the session
may have expired or interrupted, as we explained in 7.4.2.2, "Methods of the PanelBuilder servlet" on page 257.

```java
if (iChecking == 0) {
    // try again, because null is also returned if the session was dropped
    // by the Domino server
    iChecking =
        ejbPersonalization.writePersonalization {
            req.getServerName(), objApp.getDbPath("02"),
            sQuery, getMaxResults
                (hParameters),
            getServletParameter (hParameters, "tSortBy"), "1";
        }
    String sServletPath = getServletPath (req);
    sServletPath = sServletPath.substring (0,
        sServletPath.lastIndexOf ('/'));
    printHTML (out, "<meta http-equiv="refresh"
        content="0; URL=http://" +
        sServletPath + "/PanelBuilder"">", false, false);
    printHTML (out, "Loading PanelBuilder ...", true, true);
}
else if (getDbPath (req).compareTo ("") != 0) {
    generateError (req, out, "SaveQuery:doPost", 1213,
        "User not logged in");
} else {
    printHTML (out, "No application info available!", true, true);
}
} catch (Exception e) {
    generateError(req, out, "SaveQuery:doPost", 0, e.getMessage());
    System.out.println(e.getMessage());
}
```

**Figure 193. SaveQuery doPost method - retrying and sending the query results to the browser**

2. `getMaxResults(Hashtable)`
   
   This method checks the input maximum documents setting; if it is a valid numeric value, parameter is returned as an integer. If this parameter is absent or invalid, a default value of 100 documents is returned.
3. `getQueryString(Hashtable, String, String)`

This method simply parses the tokenized input parameter string using the `addDateField` and `addField` methods of the `SearchQuery` class (described in 7.3.4, “SearchQuery class” on page 252). The code is very simple, as shown in Figure 194. Not shown in the code listing is the final return of the Query string built in the `objQuery.getQuery()` object.

```java
private String getQueryString(Hashtable hParameters, String sCurrentHost, String sLtpaToken) {
    StringTokenizer tokFields = new StringTokenizer
    (getServletParameter (hParameters, "tListOfSearchFields"), ";");
    boolean bSuccess = true;
    String sField = "";
    SearchQuery objQuery = null;
    if (getServletParameter (hParameters, "tOperator").compareTo("OR") == 0) {
        objQuery = new SearchQuery (true);
    } else {
        objQuery = new SearchQuery (false);
    }
    while (tokFields.hasMoreTokens()) {
        sField = tokFields.nextToken();
        if (sField.compareTo("") != 0) {
            if (sField.startsWith("dt")) {
                // date field
                boolean bDateBefore = true;
                if (getServletParameter (hParameters, sField + "_OPERATOR").compareTo(">") == 0)
                    bDateBefore = false;
                bSuccess = objQuery.addDateField (sField,
                getServletParameter (hParameters, sField + ".YEAR"),
                getServletParameter (hParameters, sField + ".MONTH"),
                getServletParameter (hParameters, sField + ".DAY"),
                bDateBefore,
                sCurrentHost,
                sLtpaToken);
            } else {
                // text field
                bSuccess = objQuery.addField (sField,
                getServletParameter (hParameters, sField), "");
            }
        }
    }
    return (bSuccess == true) ? objQuery.getQuery() : "";
}
```

**Figure 194. The getQueryString method of the SaveQuery servlet**
There is one more servlet in our application. It is the *CallDominoJsp* servlet that we use as a generic servlet to forward to any JSP that requires a Domino session. Therefore we will discuss this servlet in Chapter 8, “JavaServer Pages” on page 269.

### 7.5 Summary

In this chapter we first discussed the different ways to invoke servlets. We then looked at how you can use VisualAge for servlets development and went on to describe the helper classes we developed for use in our servlets. Finally, we went through the code of the individual servlets we developed.
Chapter 8. JavaServer Pages

JavaServer Pages (JSP) is a page template technology that is an integral part of the J2EE specification. It allows page designers to easily combine static and dynamic content in an HTML page without having to know a lot of Java.

In this chapter we examine why the JSP technology exists and how it works, and the syntax of JSP.

We will then show how to display Domino data embedded in a Java Bean in a JSP.

Version 1.1 of the JSP specification introduced the use of custom tag libraries for JSPs. We discuss how to program a tag library, and show how to code some simple custom tags before creating the view navigator tag library that we use in our sample application. Finally, we discuss how we built the JSPs that use our tag library, as well as the servlet to handle the forwarding to the JSPs.

8.1 Overview of JSP

In this section we discuss the rationale behind using JSP and the underlying architecture of JSP.

8.1.1 Why use JSP?

Servlet technology is very flexible and allows you to perform many different tasks at once. For example, you can have a servlet respond to a request for a bank account balance by querying a backend database for the data and then sending an HTML response containing the requested information.

One problem with such a servlet is that the Java code to query the database and the code to create the HTML presentation are mixed together in a single location. This will make maintenance of the servlet much more complex later on. If a page designer wants to change the layout or static content of the HTML results page, then it would be necessary to update the servlet code, recompile the servlet, and restart the application server. If a backend Java programmer needs to change the data access code, the programmer must be careful not to change the HTML response code.

In practice, coding all but the simplest HTML responses in a servlet is very cumbersome. Ideally, we would like to separate the data access and business logic Java code from the HTML presentation code. JavaServer Pages technology accomplishes this separation for us.
In a JSP, you create a page template for the presentation of dynamic data. This template combines static HTML with dynamic content and looks very much like a regular HTML page. The difference is in additional tags that specify the dynamic content.

One significant benefit of JSP is that page developers can develop the presentation layer of a Web application without having to be experts in Java. Thus the labor of building a WebSphere application can be divided between client-side page developers and server-side Java developers. The page developers can focus on presentation components such as JSP, client-side scripting, content, page layout, and graphics. The Java developers will then deal with business logic and data access in the form of servlets, JavaBeans, EJBs, and custom tags.

Using JavaBeans, the servlet developer can pass data to the JSP page for presentation. Combining servlets and JSP pages in this way is an example of the robust Model/View/Controller programming model. This process is described in 8.2.4, “JSP and JavaBeans” on page 277.

As of the JSP 1.1 specification, you can extend the JSP syntax by creating your own custom JSP tag libraries. Using custom tags, a Java developer can “package” a large amount of complex Java code into a single tag. This technique is discussed in 8.4, “Custom tags” on page 295.

8.1.2 JSP architecture

In the preceding section we mentioned JSP as an alternative to coding HTML responses in servlets. In fact, JSP is an extension of servlet technology. Behind each JSP page is a servlet that does the actual work. You can think of a JSP page as a convenient method for creating a servlet that outputs a lot of HTML.

The first time a JSP resource is invoked, the JSP processor translates the JSP source code into a Java source code file which defines a servlet class. This special servlet class implements the javax.servlet.jsp.HttpJspPage interface which itself extends the Servlet interface. Similar to the service() method in Servlet, HttpJspPage requires the implementation of the _jspService() method, which is invoked when the JSP page is requested.

This new servlet is then compiled, the class is loaded, and the _jspService() method is invoked. Once the HttpJspPage servlet has been compiled, it stays loaded in memory to await further requests. However, every time the JSP page is requested the timestamp on the JSP source file is checked to determine if it has been updated. If so, the Java source file for the HttpJspPage
servlet will be generated again and the servlet class will be compiled and loaded again.

As an example, we created the following JSP file in a text editor and saved it with the file name simple.jsp:

```html
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Simple JSP Page</title>
</head>
<body>
<p>The time on the server is: <%= new java.util.Date() %></p>
</body>
</html>
```

This simple page includes the current time on the server as its dynamic content. Notice that for the most part the source code resembles HTML. The one new section is the the fragment `<%= new java.util.Date() %>` This tag represents a JSP expression that displays the time on the server at that specific position in the HTML. JSP expressions are explained in the next section, 8.2, “JSP syntax” on page 273.

To try it out yourself you can create the file in any text editor and save it to the Web directory of a WebSphere Web application. For example, if you use the Web application Redex we created in 4.5, “Setting up WebSphere application security” on page 147 you must put the JSP file in this directory:

`\WebSphere\AppServer\hosts\default_host\Redex\web`

Then use an URL similar to this to invoke it:

`http://odin.lotus.com/webapp/Redex/simple.jsp`

When we invoked the JSP page for the first time, WebSphere automatically generated a .dat and a .class file, as seen in Figure 195 on page 272.
However, in our example a Java file also is generated, which is good for debugging purposes. To keep the Java code generated when the JSP file is processed, do the following from the Administrative console:

1. Stop the application server (in our case Default Server).
2. Click the JSP listed under your application.
3. Click the Advanced tab.
4. Under Init. Parameters, add “keepgenerated” with a value of “true.”
5. Click the Apply button and restart your server.

In our case the output files are created in the C:\WebSphere\AppServer\temp\Redex directory where Redex is the name of the Web application.

If you open and inspect the Java source file, you will find it defines a class that extends the HttpJspBase class. This class, which is in the com.sun.jsp.runtime package, extends HttpServlet and implements HttpJspPage. Inside the _jspService() method you will find the Java code that outputs both the static and dynamic response data.

More information on creating and deploying JSP pages is given in 8.3, “Using a bean in a JSP to display Domino database properties” on page 280.
8.2 JSP syntax

The elements that make up the JSP 1.1 specification can be divided into three categories:
- Directives
- Scripting
- Actions

8.2.1 JSP directives

JSP directives inform the JSP processor servlet about the structure and characteristics of the JSP page. There are three types of JSP directives: page, include, and taglib.

8.2.1.1 Page directive

A JSP page directive describes certain characteristics of the JSP page at translation time. It has the following format:

```jsp
<%@ page attribute1="value1" attribute2="value2" ... %>
```

Table 10 lists some of the important attributes associated with the JSP page directive.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>Imports Java packages or classes for use in scripting elements of the JSP page.</td>
</tr>
<tr>
<td>session</td>
<td>Indicates whether the JSP page will access the HttpSession. If not included, the default is true. For increased performance, you should specify false if the JSP page does not use the HttpSession object.</td>
</tr>
<tr>
<td>errorPage</td>
<td>Relative URL of an error page to forward to if an uncaught exception occurs at runtime. If not specified, an uncaught runtime exception will be raised to the Servlet Engine level.</td>
</tr>
<tr>
<td>isErrorPage</td>
<td>Indicates whether the JSP page is an error page. An error page is a JSP page that is referred to by another JSP page's errorPage attribute. The default is false.</td>
</tr>
</tbody>
</table>

The following is an example of a JSP page directive:

```jsp
<%@ page import="java.util.*" session="false" errorPage="error.jsp" %>
```
The preceding directive indicates that:

- The classes in the package java.util are available to scripts in the page.
- The JSP page will not access the HttpSession object.
- If an exception is encountered, forward to error.jsp in the same directory.

For a complete list of page directive attributes, visit Sun’s JavaServer Pages Web site at http://java.sun.com/products/jsp/

8.2.1.2 Include directive
The include directive inserts the contents of a text file into the JSP source at translation time. Note that this include happens only once, when the JSP page is translated to servlet source code. Therefore, any changes made to the included file will not be picked up at runtime, even if the application server is restarted.

The include directive has the form `<%@ include file="/include.jsp" %>`. The file attribute is a relative URL pointing to the file to be included.

To include content at runtime you would use the `<jsp:include>` action discussed in 8.2.3, “JSP actions” on page 276.

8.2.1.3 Taglib directive
The taglib directive indicates what custom tag libraries are available in the JSP page. Custom tags and this directive are discussed in detail in 8.4, “Custom tags” on page 295.

8.2.2 JSP scripting elements
JSP uses Java as its server-side scripting language. The following scripting elements are available for the inclusion of Java code in a JSP page:

- Declarations
- Scriptlets
- Expressions

8.2.2.1 Declarations
The declaration tag allows the declaration of class level methods and variables. This is most useful when you want to create a jspInit() method that is called when the corresponding JspPage servlet is first loaded or a jspDestroy() method called when the JspPage is unloaded.

The tag has the form `<%! declarations %>`.
8.2.2.2 Scriptlets
The scriptlet opening and closing tags (<% ... %>) contain actual Java code that is included as is in the JspPage servlet source code. Consider the following JSP fragment:

```jsp
<h2>Loopy</h2>
<% for (int i = 0; i < 5; i++) { %>
<p>looping...</p>
<% } %>
```

This roughly translates to the following servlet source code in the _jspService() method:

```java
out.write("<h2>Loopy</h2>");
for (int i = 0; i < 5; i++) {
    out.write("<p>looping...</p>");
}
```

Notice that the static HTML lines were translated into out.write() method calls while the for statement and closing brace appeared exactly as it did in the JSP source. The output of this fragment is the <p>looping...</p> line repeated five times.

8.2.2.3 Expressions
Expressions provide an easy way to insert a value into the HTML response. They have the form `<%= someExpression %>`. The expression someExpression is evaluated and its value is written out to HTML. If someExpression evaluates to an object, then its toString() is used to print the value.

For example, the following JSP code:

```jsp
<b><%= obj1.returnSomeString() + obj2.returnSomeObject %></b>
```

roughly translates to:

```java
out.write("<b>");
out.write(obj1.returnSomeString() + obj2.returnSomeObject().toString());
out.write("</b>");
```

8.2.2.4 Implicit objects
In scriptlets and expressions you can use a set of objects that are instantiated automatically. Some important ones are listed in the following table.
Table 11. Implicit objects in JSP

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Object type</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>javax.servlet.http.HttpServletRequest</td>
</tr>
<tr>
<td>pageContext</td>
<td>javax.servlet.jsp.PageContext</td>
</tr>
<tr>
<td>session</td>
<td>javax.servlet.http.HttpSession</td>
</tr>
<tr>
<td>application</td>
<td>javax.servlet.ServletContext</td>
</tr>
<tr>
<td>config</td>
<td>javax.servlet.ServletConfig</td>
</tr>
<tr>
<td>out</td>
<td>javax.servlet.jsp.JspWriter</td>
</tr>
<tr>
<td>exception</td>
<td>java.lang.Throwable</td>
</tr>
</tbody>
</table>

The request object gives you access to information about the HTTP request including any parameters sent using an HTML form or query string. The following scriptlet extracts a parameter value from the request:

```jsp
<% String sParam = request.getParameter("param"); %>
```

The pageContext, request, session and application objects represent different scope levels where you can store and retrieve objects using the `setAttribute()` and `getAttribute()` methods. Objects stored in `pageContext` can only be referenced in the current JSP page. If you forward to another JSP page, then the object is lost. The request object allows you to preserve data when forwarding to another JSP. The session scope allows sharing with other JSP pages and servlets in the same session while application scope allows data to be referenced across HTTP sessions.

The config object gives you access to servlet configuration information such as initialization parameters. The out object allows you to call methods on the JspWriter including buffer actions such as `clear()` and `flush()`. For JSP pages with a page directive of `isErrorPage="true"`, the exception object is available which represents the exception thrown by the previous JSP page.

### 8.2.3 JSP actions

JSP action tags have the form `<jsp:someAction>`. The `<jsp:forward>` and `<jsp:include>` actions allow different JSP pages, HTML pages and servlets to work together to create a single HTTP response. As the name implies, `<jsp:forward page="forward.jsp" />` forwards the client request to another Web resource. The current response is cleared before the forward action occurs. Using `<jsp:include page="include.jsp" />` includes the HTML generated by the included Web resource into the current response.
Another action, `<jsp:plugin>`, directs the Web browser to download a Java plugin to execute an applet or JavaBean.

The remaining actions deal with using JavaBeans in JSP pages and are discussed in the next section.

### 8.2.4 JSP and JavaBeans

The use of JavaBeans allows data to be shared between servlets and JSP pages. A typical model for handling an HTTP request in a WebSphere application is as follows:

1. The request is sent to a servlet.
2. The servlet processes the request, including accessing backend data and EJBs.
3. The servlet creates JavaBeans to store the resulting data.
4. The servlet forwards the request to an appropriate JSP page to display the data stored in the JavaBeans.

This sequence is an example of the Model/View/Controller programming model. The servlet acts as the controller. It validates the request, collects the required data, creates JavaBeans to store the data, and selects the appropriate JSP page to view the results. The model is the JavaBean that represents the business data. The JSP page provides the view or presentation of the data in the model.

The Model/View/Controller framework provides clean separation between the business logic and presentation layers. The key to this architecture is the use of JavaBeans to share data between the servlet and the JSP page. After the servlet has created the JavaBeans, it will typically store them as attributes in the HttpRequest or HttpSession objects for use by the JSP page.

In the JSP page, you could write Java code in scriptlets to access the JavaBeans. However, the goal of JSP is to minimize the use of Java code and to allow page authors with little or no Java experience to create presentation views. To this end, JSP includes the `<jsp:useBean>`, `<jsp:getProperty>`, and `<jsp:setProperty>` actions for use with JavaBeans.


#### 8.2.4.1 `<jsp:useBean>` action

The `<jsp:useBean>` action obtains a reference to a JavaBean from a specified scope. If the JavaBean cannot be located, a new one may be created.
depending on the parameters specified. Table 12 lists the parameters of
\texttt{<jsp:useBean>}
.

\begin{tabular}{|l|l|}
\hline
Parameter & Description \\
\hline
id & The identifier for the JavaBean. This identifier will be used by other tags to refer to the JavaBean. Also, this identifier is used as the key when locating the object. \\
\hline
scope & Indicates at what scope level the JavaBean has been stored. \\
\hline
type & The static type of the JavaBean's identifier. \\
\hline
class & The class of the JavaBean. \\
\hline
\end{tabular}

Only one of the type or class attributes is required, although they may both be present. The parameters are best explained by inspecting the Java source code in the generated servlet:

\begin{verbatim}
<jsp:useBean id="key" scope="session" type="TypeName" class="ClassName" />

TypeName key = null;
synchronized (session) {
    key = (TypeName)
    pageContext.getAttribute("key",PageContext.SESSION_SCOPE);
    if (key==null) {
        // attempt to create new instance using ClassName
    // store new instance in specified scope
    }
}
\end{verbatim}

Note that in this scenario, where both \texttt{type} and \texttt{class} parameters are specified, \texttt{TypeName} must be the same class as \texttt{ClassName} or a superclass of \texttt{ClassName}. Or, if \texttt{TypeName} is an interface, \texttt{ClassName} must implement it.

If only \texttt{class} is specified, it becomes the \texttt{type} as well. If only \texttt{type} is specified, then the JavaBean will not be created if not located in the specified scope. Instead, an exception will be thrown by the JSP processor.

This behavior is summarized in Figure 196 on page 279.
There is another parameter called *beanName* which is used to instantiate serialized JavaBeans from files. For more information on the *beanName* parameter, visit Sun’s JavaServer Pages Web site at: http://java.sun.com/products/jsp/.

8.2.4.2 *<jsp:setProperty>* action

As the name suggests, the *<jsp:setProperty>* tag sets property values in a previously declared JavaBean. You can set explicit values or you can use request parameters as property values.

You use the form `<jsp:setProperty name="id" property="prop" value="val" />` to set an explicit value. You may also use a JSP expression for the value, such as `value="<%= expr %">`.

It is also possible to set bean properties directly from HTTP request parameters. `<jsp:setProperty name="id" property="prop" param="pname" />` will set the bean property to the value of the “pname” request parameter. If the
property name and parameter name are the same, you can omit param. If you specify `<jsp:setProperty name="id" property="*" />` then all properties with matching request parameter names will be set.

The `<jsp:setProperty>` action is most commonly found in the body of the `<jsp:useBean>` tags.

### 8.2.4.3 `<jsp:getProperty>` action

The `<jsp:getProperty>` action displays the value of a bean property in the response HTML. If the property is an object, then its `toString()` method will be invoked. In essence, `<jsp:getProperty name="id" property="prop" />` is another form for the JSP expression `<%= id.getProp() %>`.

Note that `<jsp:getProperty>` does not handle indexed (multi-valued) properties. WebSphere provides the `<tsx:repeat>` and `<tsx:getProperty>` custom JSP tags for displaying indexed properties. Refer to the product documentation for more information on these custom tags.

### 8.3 Using a bean in a JSP to display Domino database properties

As an example of using JSP pages, we will display certain properties of a Domino database in an HTML page. The user will fill out an HTML form, entering a user ID, password, server name, and database file name. The application will then respond by listing a few properties of the database.

Figure 197 and Figure 198 on page 281 illustrate how this application will work.
View Database Properties

User Id: ThorJensen
Password: ********
Server: odin.lotus.com
Database: names.nsf

Get Database

Figure 197. JSP example: input form

names.nsf on odin.lotus.com

Title: ITSO-DOMWAS's Address Book
Created On: Mon Apr 02 11:03:49 EDT 2001
Size (bytes): 5242580.0

Figure 198. JSP example: display results
8.3.1 Displaying Domino data in a JSP page

There are basically three methods we can use to display Domino data in a JSP page:

1. Import the lotus.domino package into the JSP page and access the Domino data in scriptlets.
2. Use a servlet to access Domino and store the required data in JavaBeans. The JSP page will then use the JavaBeans to display the data.
3. Access the Domino data using custom JSP tags.

The first option goes against our goal of separating data access logic from presentation. We want to minimize the amount of Java code in JSP pages, so the first option should not be used.

We used custom JSP tags in the Banking example. Custom tags are described in detail in 8.4, “Custom tags” on page 295.

For the Database Properties example we used the second method, which follows the Model/View/Controller programming model.

This example consists of the following four elements:

- HTML input form
- JavaBean class to store database properties
- Servlet to process request
- JSP page to display results

We will discuss each of these elements in the following sections.

8.3.1.1 HTML input form

The following is the HTML source for the input form:

```html
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Accessing Domino</title>
</head>
<body>
<h2>View Database Properties</h2>
<form name="frm" action="GetDatabase" method="post">
<table border="1">
<tr>
    <td>User Id:</td>
    <td><input type="text" name="user"></td>
```
The important thing to notice is the action attribute of the form. This will be the servlet Web path we specify when we create the servlet in the WebSphere Administrator's console.

### 8.3.1.2 DbProperties JavaBean class

We must create a JavaBean class which will store the database title, creation date, and size for the database specified in the input form. The following is the source code for this class:

```java
package com.lotus.redbook.banking.jsp;

import java.io.Serializable;
import java.util.Date;

public class DbProperties implements Serializable {
    private String title;
    private Date created;
    private double size;

    public DbProperties(String title, Date created, double size) {
        this.title = title;
        this.created = created;
        this.size = size;
    }

    public Date getCreated() {
        return created;
    }
}
```
To be able to store the JavaBean in request, session, or application scope we must implement the `java.io.Serializable` interface. This is because WebSphere must be able to write the object to disk. This can happen when it must be swapped out of memory or when the application server is stopped and persistent sessions are enabled.

In the `DbProperties` class we did not create public setter methods for the properties. This will indicate to the JSP page that the properties are read-only.

### 8.3.1.3 GetDatabaseServlet servlet
The controller servlet will use the user ID and password provided to connect to the Domino database. Then it will create the `DbProperties` object, store it in the request, and forward to the JSP page for presentation. The following is the Java source code for the servlet:

```java
package com.lotus.redbook.banking.jsp;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import lotus.domino.*;

public class GetDatabaseServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        try {
            // get HTTP parameters
            String sUser = req.getParameter("user");
            String sPwd = req.getParameter("pwd");
            String sServer = req.getParameter("server");
            String sDb = req.getParameter("db");

            // connect to Domino server and get db reference
```
Session nses = NotesFactory.createSession(sServer, sUser, sPwd);
Database db = nses.getDatabase("", sDb);

// create JavaBean to store data for JSP page
DbProperties dbprops =
    new DbProperties(db.getTitle(),
                    db.getCreated().toJavaDate(),
                    db.getSize());
req.setAttribute("dbprops", dbprops);

// tell Domino we are finished with the NotesSession
// Domino can then reclaim the memory
nses.recycle();

// prevent caching of the response
res.setHeader("Pragma", "no-cache");
res.setHeader("Cache-Control", "no-cache");
res.setDateHeader("Expires", 0);

// forward to JSP page to present information
ServletContext sc = getServletContext();
RequestDispatcher rd =
    sc.getRequestDispatcher("/dbprops.jsp");
rд.forward(req,res);
}
} catch (NotesException ne) {
    ne.printStackTrace();
    throw new ServletException("NOTES ERROR #" + ne.id + " " +
    ne.text);
}

First we get the form parameters from the HTTP request and use them to
create a Domino session object over IIOP with the NotesFactory.createSession
method.

Note: If we want to support SSO, we must pass the LTPA token as a
parameter in the NotesFactory.createSession method. However, to keep things
simple and let this example stand on its own with minimal setup, we do not
use SSO here.

Next we obtain a reference to the specified database and use the database
properties to create a new instance of the DbProperties bean class. Note that
we are using the toJavaDate() method to pass the created property to the
bean constructor. We could not use a property of the Domino type `DateTime` because it does not implement `Serializable`. Not only must the JavaBean class be `Serializable`, but any objects it refers to must also be `Serializable` for storage in request, session, or application scopes.

After creating the `DbProperties` bean, we store it as an attribute of the request object. Since we will not be using the information after the response is sent, request scope is sufficient. If the data needs to be shared with other resources, we would then save the bean in session or application scope.

The next line, `nses.recycle()`, releases the session's memory in Domino. We must explicitly tell Domino we are finished with the session, otherwise it will remain in memory, even after the reference is garbage collected in Java. Recycling the Domino session also recycles all the Domino objects created from the session, in this case the `Database`.

Finally, we set the response so it will not be cached and we forward to the JSP page to display the results. Preventing the caching of responses is a good practice when dynamic data is involved.

To forward to a JSP page, we must first obtain the current `ServletContext` object. There is one `ServletContext` object per Web application and it allows resources in the same Web application to share data and to cooperate. When an object is saved in application scope, it is in the `ServletContext` object that it is stored. The `ServletContext` is also used to obtain a `RequestDispatcher` object which allows forwarding to another Web resource. The `<jsp:forward>` and `<jsp:include>` actions use a `RequestDispatcher` in the underlying servlet.

### 8.3.1.4 JSP results page

The final element in the example is the JSP page to display the database properties:

```html
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<jsp:useBean id="dbprops" scope="request"
type="com.lotus.redbook.banking.jsp.DbProperties" />
<html>
<head>
<title>Accessing Domino - Result</title>
</head>
<body>
<h2><% request.getParameter("db") %> on
<%= request.getParameter("server") %></h2>
<table border="1">
<tr>
```
8.3.2 Deploying the Database Properties example

We now outline the steps involved in deploying and testing the Database Properties example. We assume that you have completed the installation detailed in Chapter 3, “Installation and setup” on page 39 and can reproduce the Web application setup detailed in 4.5.0.1, “Creating the example Web application” on page 148. Also, the following instructions assume you have mapped a network drive from your development workstation to the WebSphere server so you can copy files directly to the Web application’s class path.

8.3.2.1 Create and deploy the Java classes

In this step we create the servlet and bean classes in VisualAge for Java and export them as a JAR file to WebSphere. For your convenience, we have created the JAR file (DbProps.jar) together with the associated JSP and HTML file for you. Refer to Appendix K, “Using the additional Web material” on page 521 for information on finding the file 595jsp-dbprops.zip that contains DbProps.jar and the other files used in this example.

Thus, if you don’t want to try using VisualAge for Java to create and export DbProps.jar, you can just copy the file to your Web application’s servlets
directory and continue with step 8.3.2.2, “Create and deploy the page files” on page 291.

For your convenience, we also have created a JAR file (DbPropsSource.jar) with the required source code and included it as part of the additional material. Instead of typing out the source code, you can import it from DbPropsSource.jar (also part of 5955jsp-dbprops.zip) into VisualAge for Java.

To use VisualAge for Java to work with the servlet code, do the following.

1. In VisualAge for Java, create a project and the package com.lotus.redbook.banking.jsp for this example (if you import DbPropsSource.jar the package will be created automatically).

   For simplicity, we created the JavaBean and the servlet in the same package. In a production environment you would probably separate servlet files from JavaBean files.

2. Create the DbProperties and GetDatabaseServlet classes using the source code on pages 283 and 284, or import DbPropsSource.jar.

3. Export the DbProperties and GetDatabaseServlet classes in a JAR file to the Web application’s class path:
   a. Right-click the package in VisualAge for Java and choose Export...
b. In the SmartGuide Export window, select Jar file as the export destination and click Next >.

c. In the Jar file box, enter the path to the Web application’s servlets directory and a name for the JAR file. In our case, we entered \M:\AppServer\hosts\default_host\Redex\servlets\DbProps.jar where the
M: drive is mapped to the WebSphere directory on the WebSphere server. You can give any name to the JAR file.

Select to export class files as shown in Figure 201.

d. Click **Finish** to create the JAR file and export it to the WebSphere server.

Note that you do not have to explicitly list the JAR file in the Web application's classpath. If the JAR file is located in the servlets directory, then WebSphere will search the JAR file for the correct class. For maintenance purposes you may wish to store your JavaBean class files in a separate directory from your servlet class files. In this case, you must update your Web application's classpath with the new directory.
8.3.2.2 Create and deploy the page files

1. In a JSP, HTML, or a text editor, create the HTML file listed on page 282 and the JSP file listed on page 286. Save them with the names getdb.html and dbprops.jsp or get the copies supplied with the additional Web material.

2. Copy the two files to the document root directory. To determine the document root of the Web application, refer to the screen shown in Figure 202. In our case the absolute path to the Web application’s document root is `C:\WebSphere\AppServer\hosts\default_host\Redex\web`.

8.3.2.3 Add the servlet in WebSphere

1. Since we have updated classes in the Web application’s classpath, we must restart its application server for our changes to take effect. To do this, right-click on the application server name (in our case Default Server) in the administrator’s console and select Stop. Or you can use the Stop button in the toolbar.
2. Add a new servlet to the Web application:
   a. Right-click the Web application and select **Create -> Servlet**.
b. Enter the servlet properties as shown in Figure 205 on page 294. Click the **Add** button to enter the Web app path. Note that the path must match the *action* attribute of the HTML input form on page 282. Click **OK** to save the path and **OK** again to confirm all servlet settings.
c. Start the application server (in our case Default Server) by choosing Start from its context menu.

8.3.2.4 Test the example

1. Once the administrator’s console reports that the application server has started successfully, open a Web browser and navigate to the getdb.html page. In our case, our host name is mjollner.lotus.com and the Web application’s Web path is webapp/Redex/ so we entered http://mjollner.lotus.com/webapp/Redex/getdb.html.

2. Fill in the four fields and submit the form. This example does not employ SSO, so you can connect to any Domino server that is visible to WebSphere and running the DIIOP task. The user you specify must be present in the Domino Directory on the remote Domino server.
Refer to page Figure 198 on page 281 for a screen shot of sample output.

8.4 Custom tags

The key new feature in JSP 1.1 is the ability to create libraries of custom tags for use in a JSP page. One problem that can occur with JSP pages is that they can become “polluted” with complex Java code. Ideally, JSP code should resemble HTML and have very little Java code in scriptlets. Using custom JSP tags, large blocks of Java code can be encapsulated into a single tag.

The use of custom tags supports the separation of the presentation layer from the data access and business logic layers. This allows a page developer to create dynamic HTML responses without knowing a lot of Java. A Java developer can support the JSP developer by creating custom tags to represent complex behaviors. The JSP developer can then easily include these complex behaviors in a JSP page.

8.4.1 Custom tag library example: ViewNavigator tag library

As an example of using custom JSP tags to access Domino data, we will develop tags that emulate the functionality of the ViewNavigator and ViewEntry Domino Java classes.

The ViewNavigator class provides programmatic access to all entries or a subset of entries in a Domino view. Figure 206 on page 296 shows where the ViewNavigator and ViewEntry classes are located in the Domino object hierarchy.
A few important points about a ViewNavigator object are:

- **ViewNavigator** returns a ViewEntry object that contains information such as SiblingCount, view position, and so on.
- It lets you access categories and totals as well as documents.
- Users can create navigators from subsets of views; for example, all children of an entry.

You can read more about the methods of the ViewNavigator and ViewEntry classes in the Domino Designer online help.

The following JSP code excerpt shows how the custom tags that encapsulate the Java code will work:

```xml
<redbook:viewNav database="db.nsf" view="viewname">
  <redbook:viewEntry>
    JSP for all entries here
    First Column: <redbook:viewEntryColumn column="1" />
    Indent: <redbook:viewEntryIndent />
  </redbook:viewEntry>

  <redbook:viewEntry type="category">
    JSP for category entry here
  </redbook:viewEntry>

  <redbook:viewEntry type="document">
```

Figure 206. ViewNavigator and ViewEntry classes in the Domino Object Model
The outer tag `<redbook:viewNav>` represents the `ViewNavigator` class for the view specified by the database and view parameters. The JSP code inside the tag's body, which contains nested custom tags, will be repeated for each `ViewEntry` in the `ViewNavigator`.

The `<redbook:viewEntry>` tag at the next level represents a single view entry. Notice that the tag appears three times. For the first instance, the JSP text inside the tag is processed for every `ViewEntry`. The text inside the second `<redbook:viewEntry>` tag appears only for a `ViewEntry` that is a category, as indicated by the `type` parameter of “category”. Similarly, the content of the third tag only appears when the current `ViewEntry` is a document.

Inside the `<redbook:viewEntry>` tags is the `<redbook:viewEntryColumn>` tag. This tag will output the value of a single column of the current `ViewEntry`. The `column` attribute of the tag specifies which column to retrieve the value from.

The `<redbook:viewEntryIndent>` tag outputs the indent level of the current `ViewEntry`. This tag will be useful in laying out a hierarchical structure in the HTML.

Using these custom tags, a page developer is free to display view data in an HTML page using any layout and formatting desired. Contrast this with the rigidity of the embedded view.

Next we discuss the tasks involved in creating a custom JSP tag.

### 8.4.2 Developing custom JSP tags

There are three elements involved in developing and using custom tag libraries:

1. A tag handler class written in Java
2. A tag library descriptor file in XML format
3. The actual JSP page

#### 8.4.2.1 Tag handler classes

The actual behavior of a custom tag is implemented using a Java class. The Java Servlet 2.2 specification includes various interfaces and classes which support the creation of custom tag libraries.
There are two types of custom tags: those that process their body contents and those that do not. A tag that processes its body contents can, for example, be one that converts all text between its start and end tags to lowercase characters, like this:

```
<custom:lowerCase>
WILL DISPLAY LOWER CASE
</custom:lowerCase>
```

A tag that does *not* process body contents can be one that simply outputs a new string with content like current date:

```
<custom:Date />
```

For developing tags that do not process their body, you must create a class that implements the `javax.servlet.jsp.tagext.Tag` interface. To perform processing on the text between the opening and closing tags, you must implement additional methods found in the `javax.servlet.jsp.tagext.BodyTag` interface. Note that the `BodyTag` interface is a subclass of the `Tag` interface.

### The Tag interface

For custom tags that do not have to process their body contents, a tag handler class that implements the `Tag` interface should be created. For your convenience, a support class called `javax.servlet.jsp.tagext.TagSupport` already exists with the appropriate method stubs. You merely have to extend this class and override the required methods. The method of most interest is `doStartTag()` which is invoked when the beginning tag is encountered.

Here is an example of a tag handler class:

```java
package com.lotus.redbook.banking.taglib.example;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/**
 * Simple JSP tag that inserts a string into the output
 */

public class SimpleTag extends TagSupport {
    public int doStartTag() {
        try {
            JspWriter out = pageContext.getOut();
            out.print("WebSphere and Domino work together");
        }
        catch (Exception e) {
```
System.out.println("Error in SimpleTag: " + e);
e.printStackTrace();
}

return Tag.SKIP_BODY;
}
}

The first line in the try block obtains a JspWriter from the implicit pageContext object. This is the same implicit object that is available to scriptlets in the JSP page. You can also get and set attributes in the pageContext object to allow sharing of data with other JSP elements (including other custom tags). The next line inserts a string into the response. Note that unlike java.io.PrintWriter, the JspWriter print methods can throw exceptions which must be handled.

Finally, the method returns a value SKIP_BODY which is defined in the Tag interface. This indicates that any body text should be ignored and not sent out with the response. While we do not process the tag’s body contents when implementing the Tag interface, we can still decide to include the body in the HTML response. To do this, we would return a value of EVAL_BODY_INCLUDE.

The other method you may wish to override is doEndTag(), which is invoked after the tag and its contents have been processed. In this method, you have the option to cancel processing the rest of the JSP page by returning SKIP_PAGE. If you want processing to continue normally after the tag then you would return EVAL_PAGE which is the default.

The flow of events for a tag handler implementing the Tag interface is shown in Figure 207 on page 300.
Figure 207. Flowchart for tag handler implementing the Tag interface
Note that the Output Body process can include other custom tags, thus creating a nested structure of these flowcharts.

**The BodyTag interface**

If you need the ability to read and perhaps manipulate the body contents of a custom tag, then you must create a class that implements the `BodyTag` interface. The easiest way to do this is to extend the `BodyTagSupport` convenience class that has already been provided. The `BodyTag` interface is a subclass of `Tag`, so the `doStartTag()` and `doEndTag()` methods still work the same way. However, by extending `BodyTagSupport` we are now given another method called `doAfterBody()`, which is invoked after the tag's body contents have been evaluated. In the `doAfterBody()` method we can invoke another new method called `getBodyContent()`, which returns a `javax.servlet.jsp.tagext.BodyContent` object.

The `BodyContent` class is a subclass of `JspWriter` and represents the text between the opening and closing tags. If obtained in the `doAfterBody()` method, a `BodyContent` object represents the text after it has been evaluated and before it has been inserted into the response. We say "evaluated" because the body contents can contain dynamic JSP elements, such as scriptlets, expressions, and even other custom tags. You then have the option of changing the text before outputting it or not outputting it at all.

The `BodyContent` class includes the methods described in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getEnclosingWriter()</code></td>
<td>Returns a reference to the parent JspWriter</td>
</tr>
<tr>
<td><code>getReader()</code></td>
<td>Returns a Reader that allows the evaluated body contents to be read</td>
</tr>
<tr>
<td><code>getString()</code></td>
<td>Returns the evaluated body contents as a string</td>
</tr>
</tbody>
</table>

The following example class outputs the body contents of the tag all in lower case:

```java
package com.lotus.redbook.banking.taglib.example;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/**
 * Custom tag that outputs its body contents in lower case
 */
```
public class LowerCaseTag extends BodyTagSupport {
    public int doAfterBody() {
        BodyContent body = getBodyContent();
        try {
            JspWriter out = body.getEnclosingWriter();
            out.print(body.getString().toLowerCase());
        }
        catch (Exception e) {
            System.out.println("Error in LowerCaseTag: " + e);
            e.printStackTrace();
        }

        return Tag.SKIP_BODY;
    }
}

The first line of the doAfterBody() method obtains a reference to the current body contents represented by a BodyContent object. The getString() method is used to get the body as a String, which is converted to lower case before being output.

Returning SKIP_BODY from doAfterBody() indicates that you are finished with the body contents of the tag and to continue JSP processing. You may also return BodyTag.EVAL_BODY_TAG, which will cause the body to be evaluated again and another call to doAfterBody(). This allows us to create looping structures as follows:

package com.lotus.redbook.banking.taglib.example;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/**<*
 * Custom tag that outputs its body contents five times
 */

public class LoopTag extends BodyTagSupport {
    private int i = 5;

    public int doAfterBody() {
        if (i-- > 0) {
            BodyContent body = getBodyContent();
            try {
                JspWriter out = body.getEnclosingWriter();
                out.print(body.getString());
            }
        }

        return BodyTag.EVAL_BODY_TAG;
    }
}
body.clearBody(); // clear body for next evaluation
}
catch (Exception e) {
    System.out.println("Error in LoopTag: " + e);
    e.printStackTrace();
}
return BodyTag.EVAL_BODY_TAG;
}
else {
    return Tag.SKIP_BODY;
}
}
}

Here we have a member variable that controls the number of repetitions. This variable is decremented each time doAfterBody() is invoked. To continue looping we return EVAL_BODY_TAG and when we have finished our loop we return SKIP_BODY.

You may notice that we did not implement the doStartTag() method in the previous two examples. In the BodyTagSupport class, doStartTag() returns EVAL_BODY_TAG by default. If you were to override doStartTag() and return SKIP_BODY, then the body contents would not be evaluated and doAfterBody() would never be called.

The flow of events for a tag handler implementing the BodyTag interface is shown in Figure 208 on page 304.
Again, the Output Body process can include other custom tags, creating a nested structure of these flowcharts.

**Tag attributes**

JSP allows us to specify attributes for tags in the following format:

```xml
<lib:tag attr1="value1" attr2="value2" ... />
```
Handling tag attributes is similar to using properties in JavaBeans. You must declare a member variable representing the attribute, as well as a public setter method that is called automatically by the JSP 1.1 processor. The member variables are then available to you in the various Tag methods, such as doStartTag(). The following is an outline of a tag handler class for the above custom tag:

```java
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/**
 * Simple JSP tag that has attributes
 */

public class AttributeTag extends TagSupport {
    private String sAttr1;
    private int iAttr2;

    public void setAttr1(String a1) {
        sAttr1 = a1;
    }

    public void setAttr2(String a2) {
        try {
            iAttr2 = Integer.parseInt(a2);
        } catch (NumberFormatException nfe) {
            iAttr2 = 0;
        }
    }

    public int doStartTag() {
        // do something with sAttr1 and iAttr2
        return Tag.SKIP_BODY;
    }
}
```

Similar to JavaBean properties, the names of the setter methods must conform to the attribute names. That is, for the tag `<lib:tag value="4" />`, the setter method must have the signature `setValue(String s)`. Notice that we attempt to convert the second attribute to an integer and that the appropriate exception is handled in the setter method.
In a previous example we created a tag that output its body contents five times. We will now modify the example to accept a parameter representing the number of repetitions:

```java
package com.lotus.redbook.banking.taglib.example;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/**
 * Custom tag that outputs its body contents x times
 * where x is a tag parameter named "reps"
 */

public class Loop2Tag extends BodyTagSupport {
    private int iReps;

    public void setReps(String s) {
        try {
            iReps = Integer.parseInt(s);
        }
        catch (NumberFormatException nfe) {
            iReps = 0;
        }
    }

    public int doAfterBody() {
        if (iReps-- > 0) {
            BodyContent body = getBodyContent();
            try {
                JspWriter out = body.getEnclosingWriter();
                out.print(body.getString());
                body.clearBody(); // clear body for next evaluation
            }
            catch (Exception e) {
                System.out.println("Error in Loop2Tag: " + e);
                e.printStackTrace();
            }
            return BodyTag.EVAL_BODY_TAG;
        } else {
            return Tag.SKIP_BODY;
        }
    }
}
```
**Nesting custom tags**

Let us review the ViewNavigator custom tag library example from 8.4.1, "Custom tag library example: ViewNavigator tag library" on page 295:

```xml
<redbook:viewNav database="db.nsf" view="viewname">
    <redbook:viewEntry>
        JSP for all entries here
        First Column: <redbook:viewEntryColumn column="1" />
        Indent: <redbook:viewEntryIndent />
    </redbook:viewEntry>

    <redbook:viewEntry type="category">
        JSP for category entry here
    </redbook:viewEntry>

    <redbook:viewEntry type="document">
        JSP for document view entry here
    </redbook:viewEntry>
</redbook:viewNav>
```

In this example the custom tags follow a precise nesting structure. For example, a `<redbook:viewEntryIndent />` tag only makes sense when inside `<redbook:viewEntry> ... </redbook:viewEntry>` tags. Also, to support this hierarchy, there must be a way for outer tags to share data with inner tags. In this case, the `<redbook:viewEntryIndent />` tag must know which `ViewEntry` the enclosing `<redbook:viewEntry>` tag refers to.

To accomplish this nesting, the `TagSupport` class contains a method called `findAncestorWithClass()`, which can obtain a reference to the enclosing tag handler object. Once you have this reference you have access to any visible member properties and methods of the enclosing tag handler object.

The following excerpt of a `doStartTag()` shows how to obtain a reference to the enclosing tag handler class:

```java
public int doStartTag() throws JspTagException {
    OuterTag outer =
        (OuterTag) findAncestorWithClass(this, OuterTag.class);
    if (outer == null) {
        throw new JspTagException("Nesting Error");
    }

    outer.someMethod();
    // and so on
}
```
In this example, if the JSP author does not nest the tag properly, we throw a 
`JspTagException`. Until now, we have handled all exceptions locally using `try -
catch` blocks. However, if a severe error occurs, we may want to cancel the 
processing of the JSP page. Recall that JSP specifies a page directive called 
`errorPage` that allows the author to specify an error page to forward to if an 
unhandled exception occurs. To support this functionality, you can throw a 
`JspTagException` from a tag handler class method, which will then be handled 
by the JSP processor servlet.

### 8.4.2.2 Tag library descriptors

In the previous section we dealt with the creation of the Java tag handler 
classes that define the behavior of custom tags. When the JSP 1.1 processor 
encounters a custom tag, it needs a mechanism to map the custom tag to the 
appropriate tag handler class. The JSP 1.1 processor also needs information 
about what custom tags exist and how they are structured. A tag library 
descriptor (TLD) file in XML format is used to provide this information.

The following is the TLD file which describes the custom tag defined by the 
`SimpleTag` tag handler class on page 298.

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC 
"-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN" 
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tag library descriptor -->

<taglib>

<!-- after this the default space is 
"http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd" -->

    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>example</shortname>
    <urn></urn>
    <info>Custom tag examples</info>

    <tag>

        <name>simple</name>
        <tagclass>com.lotus.redbook.banking.taglib.example.SimpleTag</tagclass>
        <info>Simple example, display static text</info>

        <bodycontent>EMPTY</bodycontent>
    </tag>
</taglib>
```
In the TLD file, you specify information about a tag library and all the custom tags that are contained in the library. The important elements are displayed in bold and are explained in the following table.

### Table 14. Elements of a TLD file

<table>
<thead>
<tr>
<th>TLD tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;shortname&gt;example&lt;/shortname&gt;</code></td>
<td>Name the JSP author will use to refer to this custom tag library. Used in the <code>&lt;%@ taglib %&gt;</code> directive.</td>
</tr>
<tr>
<td><code>&lt;name&gt;simple&lt;/name&gt;</code></td>
<td>Name the JSP author will use to insert the custom tag.</td>
</tr>
<tr>
<td><code>&lt;tagclass&gt;com.lotus.redbook.banking.taglib.SimpleTag&lt;/tagclass&gt;</code></td>
<td>The name of the tag handler class.</td>
</tr>
<tr>
<td><code>&lt;bodycontent&gt;EMPTY&lt;/bodycontent&gt;</code></td>
<td>Indicates whether the custom tag expects text between the opening and closing tags. A value of “JSP” is used to indicate the tag should have a body.</td>
</tr>
</tbody>
</table>

In the TLD file, you also have to specify what attributes (if any) a tag may accept and whether the attributes are required or not. You can also specify whether an attribute may be a JSP runtime expression such as `<lib:tag value="<%= someExpression %>>" />`. If the TLD file specifies that runtime expressions are not allowed, the literal text “<%= someExpression %>” will be passed as the parameter.

The following TLD file includes tag definitions for the `SimpleTag`, `LowerCaseTag` and `Loop2Tag` example tag handlers:

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<!-- a tag library descriptor -->
<taglib>
  <!-- after this the default space is "http://java.sun.com/j2ee/dtds/jsp.taglibrary_1_2.dtd" -->
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>redbook</shortname>
  <urn></urn>
  <info>Redbook example custom tags</info>
</taglib>
```
8.4.2.3 Using custom tag libraries in a JSP page

In the previous two sections we created tag handler Java classes that implement the behavior of a custom tag and we created TLD files which describe a custom tag library to the JSP 1.1 processor. Now we examine how to include these custom tags into a JSP page.

Before using a custom tag, you must declare its custom tag library using the JSP TagLib directive. This directive takes the following format:

```
<%@ taglib uri="/tld/example.tld" prefix="redbook" %>
```

The uri attribute contains a mapping to the appropriate TLD file. Like other Web resources, this URI is relative to the Web application’s Web path directory. In this case, we created a new directory at the root level to store the TLD files. This allows JSP files from anywhere in the Web application’s
directory structure to access the TLD file and also allows us to move JSP pages without changing the `uri` attribute.

The `prefix` attribute must correspond to the `<shortname>` of the tag library in the TLD file. It is used in the name of a custom tag, as shown in the following JSP code fragment:

```html
<p><redbook:simple /></p>
```

Custom JSP tags follow a `<prefix:tagname>` naming convention where `prefix` refers to the `prefix` attribute of the TagLib directive and `tagname` corresponds to `<name>` in the tag definition in the TLD file.

The following sample JSP file uses the three custom tags found in the `example` custom tag library:

```html
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ taglib uri="/tld/example.tld" prefix="redbook" %>
<html>
<head>
<title>Custom Tag Library</title>
</head>
<body>
<h2>Redbook Custom Tag Library Examples</h2>

<h3>Simple Tag</h3>
<p><redbook:simple /></p>

<h3>Lower Case Tag</h3>
<p><redbook:lowerCase>
THIS TEXT WAS ALL UPPER CASE IN THE JSP SOURCE!!!
</redbook:lowerCase>
</p>

<h3>Loop2 Tag</h3>
<redbook:loop2 reps=' '<%= request.getParameter("reps") %>'>
<p>JSP is fun!</p>
</redbook:loop2>
</body>
</html>
```
8.4.3 Deploying custom tag libraries

In the previous section we discussed the elements that make up a JSP custom tag library and the structure of those elements. We will now deploy the previous examples to the WebSphere environment for testing.

8.4.3.1 Initial setup

Before you deploy the custom tag examples, you must have completed the setup detailed in Chapter 6, “Setup of the development and test environment” on page 197.

Specifically, to deploy these examples you must ensure the following:

1. You have imported the Servlet 2.2 API classes into VisualAge for Java.

![Figure 209. The Servlet 2.2 API classes in VisualAge for Java. The javax.servlet.jsp.tagext package must be present to create the tag handler classes.](image)

2. In WebSphere, you have created a Web Application that uses JSP 1.1.
Chapter 8. JavaServer Pages

8.4.3.2 Create and deploy the tag handler classes

In this step we create the tag handler classes in VisualAge for Java and export them as a JAR file to WebSphere. For your convenience, we have created the JAR file (TagLibExample.jar) for you, and included it as part of the additional material in the file 5955jsp-example-taglib.zip. So, instead of using VisualAge for Java to create and export TagLibExample.jar, you can just copy the file to your Web application’s servlets directory and continue with step 8.4.3.3, “Create and deploy the TLD file” on page 317.

Refer to Appendix K, “Using the additional Web material” on page 521 for information on finding 5955jsp-example-taglib.zip that contains TagLibExample.jar and the other files in this example.

For your convenience, we also have created a JAR file (TagLibExampleSource.jar) with the required source code and included it as part of the additional material. Instead of typing out the source code, you can import from this file into VisualAge for Java.

1. In VisualAge for Java, create the SimpleTag, LowerCaseTag and Loop2Tag tag handler classes in VisualAge for Java using the source code examples from 8.4.2, “Developing custom JSP tags” on page 297, or import the TagLibExampleSource.jar file.
You can use the same project as in the previous example in 8.3.2, “Deploying the Database Properties example” on page 287. The package name is com.lotus.redbook.banking.taglib.example.

2. Export the tag handler classes in a JAR file to the Web application’s class path:
   a. Right-click the package in VisualAge for Java and choose Export...
b. In the SmartGuide Export window, select **Jar file** as the export destination and click **Next >**.

c. In the Jar file box, enter the path to the Web application’s servlets directory and a name for the JAR file. In our case, we entered `M:\AppServer\hosts\default_host\Redex\servlets\TagLibExample.jar` where the M: drive is mapped to the WebSphere directory on the WebSphere server. You can give any name to the JAR file.

Make sure to choose export only **class code**, as shown in Figure 213 on page 316. De-select export of Java code if necessary.
Figure 213. Entering a file path and name for an exported JAR file

<table>
<thead>
<tr>
<th>Jar File</th>
<th>Server\hosts\default_host\Redox\services\TaglibExample.jar</th>
<th>Browse...</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Details...</td>
<td>3 selected</td>
</tr>
<tr>
<td>java</td>
<td>Details...</td>
<td>3 selected</td>
</tr>
<tr>
<td>resource</td>
<td>Details...</td>
<td>0 selected</td>
</tr>
<tr>
<td>beans</td>
<td>Details...</td>
<td>0 selected</td>
</tr>
</tbody>
</table>

Select referenced types and resources

- Deselect BeanInfo and PropertyEditor

Do you want to create .html files to launch applets?
- html Details... 0 selected

Do you want to seal the contents of the jar file?
- Seal the jar Details... Unseal packages...
- Do not seal the jar Seal packages...

Options
- Include debug attributes in class files.
- Compress the contents of the jar file.
- Overwrite existing files without warning.
- Automatically open a web browser on created .html files.

- Click **Finish** to create the JAR file and export it to the WebSphere server.

For the Web application to use custom tags, their tag handler classes must be in the Web application’s classpath. Note that you do not have to explicitly list the JAR file in the Web application’s classpath. If the JAR file is located in the servlets directory then WebSphere will search the JAR file for the correct class. For maintenance purposes you may wish to store your tag handler class files in a separate directory from your servlet class files. In this case, you must update your Web application’s classpath with the new directory.

If more than one Web application will use the custom tag library, you should then export to the *application server’s* classpath.
8.4.3.3 Create and deploy the TLD file
1. In an XML or text editor, create the TLD file listed on page 309 and save it with the file name example.tld, or get the file from our additional Web material.

2. In the document root directory of the Web application, create a new directory called tld. To determine the document root of the Web application, refer to the screen shown in Figure 214. In our case, the document root is C:\WebSphere\AppServer\hosts\default_host\Redex\web.

3. Copy example.tld to this new directory.

8.4.3.4 Create and deploy the JSP file
1. In a JSP or text editor, create the JSP file listed on page 311 and save it with the name example.jsp, or get the file from our additional Web material.

2. Copy example.jsp to the document root directory. In our case the absolute path to the Web application's Web path is C:\WebSphere\AppServer\hosts\default_host\Redex\web.
8.4.3.5 Test the deployment

1. Since we have updated classes in the Web application’s classpath, we must restart its application server for our changes to take effect. To do this, right-click the application server name in the administrator’s console and select Stop. Then repeat and select Start. Or you can use the Stop and Play buttons in the toolbar.

2. Once the administrator’s console reports that the application server has started successfully, open a Web browser and navigate to the example.jsp page. In our case, our host name is mjollner.lotus.com and the document root is webapp/Redex/ so we entered:

http://mjollner.lotus.com/webapp/Redex/example.jsp?reps=2

Remember to include a query string parameter indicating the number of repetitions for the <redbook:loop2> tag.

![Redbook Custom Tag Library Examples](http://cdn.lotus.com/webapp/Redex/example.jsp?reps=2)

To investigate custom tags further, you can compare the static HTML source code with the original JSP code. Also, you can inspect the Java source code that the JSP 1.1 processor generated from example.jsp. You can locate
where the tag handlers are instantiated and where their various methods are called. In our case the servlet source code and class files generated by the JSP processor are located in the WebSphere\AppServer\temp\default_host\Redex directory where Redex is the name of our Web application.

### 8.5 Custom tag example: ViewNavigator tags

Recall the ViewNavigator custom tag library example from section 8.4.1:

```xml
<redbook:viewNav database="db.nsf" view="viewname">
    <redbook:viewEntry>
        JSP for all entries here
        First Column: <redbook:viewEntryColumn column="1" />
        Indent: <redbook:viewEntryIndent />
    </redbook:viewEntry>

    <redbook:viewEntry type="category">
        JSP for category entry here
    </redbook:viewEntry>

    <redbook:viewEntry type="document">
        JSP for document view entry here
    </redbook:viewEntry>
</redbook:viewNav>
```

This section describes how to develop the tag handler classes and the TLD file to support these tags.

The source code for the tag handler classes discussed here is available in the file BankingSource.jar that is part of 5955java.zip in the additional Web material for this book. See Appendix K, “Using the additional Web material” on page 521 for instruction on how to get the file.

Note that when we develop custom tag libraries, we want to make them as flexible and general as possible. This will allow the custom tag libraries to be reused in other applications. Application-specific details should be dealt with in the JSP page that uses the tags and with attributes in the custom tags.

In our example, the custom tags make no assumptions about layout or formatting. The details of how the view data will be displayed are left to the enclosing JSP page. Also, attributes for the database and view allow these custom tags to be used in any application.
8.5.1 `<redbook:viewNav>` tag

The top-level `<redbook:viewNav>` tag is responsible for creating the appropriate `ViewNavigator` and iterating over the entries in the `ViewNavigator`. We will design the tag so it assumes that a Domino session has already been created and has been stored as an attribute of the `HttpServletRequest`.

The creation of the Domino session would most likely be done by a controller servlet. We use the DominoLogin servlet as described in 7.4.1, “The DominoLogin servlet” on page 254.

To allow the tag to locate the Domino session, we will add a tag attribute called `sessionKey`. We will also assume that the JSP author will invoke `recycle()` on the Domino session after the custom tags have executed.

The documentation for the Domino Java classes specifies the methods shown in Table 15 for creating a `ViewNavigator` from a `View` object.

<table>
<thead>
<tr>
<th>View method name</th>
<th>Description of <code>ViewNavigator</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createViewNav()</code></td>
<td>All entries</td>
</tr>
<tr>
<td><code>createViewNavFrom()</code></td>
<td>All entries starting with a specified entry</td>
</tr>
<tr>
<td><code>createViewNavFromCategory()</code></td>
<td>All entries under a specified category</td>
</tr>
<tr>
<td><code>createViewNavFromChildren()</code></td>
<td>Entries which are immediate children of a specified entry</td>
</tr>
<tr>
<td><code>createViewNavFromDescendants()</code></td>
<td>Entries which are descendants of a specified entry</td>
</tr>
<tr>
<td><code>createViewNavMaxLevel()</code></td>
<td>All entries in a view down to a specific level</td>
</tr>
</tbody>
</table>

For the banking example, we needed the ability to obtain a `ViewNavigator` for a single category and for the first level entries only. To accomodate this, we added two more optional attributes to the ones listed in the preceding example: `category` and `maxlevel`. If neither attribute is specified, we will create a `ViewNavigator` with all view entries. If `category` is specified, we will call `createViewNavFromCategory()` and get entries for the category only. If `maxlevel` is given, we will use `createViewNavMaxLevel()` and only retrieve entries down to the specified level. If both attributes are present, we will throw an exception.
8.5.1.1 ViewNavTag tag handler class

The following is the source code for the ViewNavTag class:

```java
/**
 * File: ViewNavTag.java
 * Defines the behavior of the <redbook:viewNav>
 * custom tag
 */

package com.lotus.redbook.banking.taglib;

// Java classes
import java.io.*;
import java.util.*;

// Servlet classes
import javax.servlet.*;
import javax.servlet.http.*;

// JSP classes
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

// Domino classes
import lotus.domino.*;

public class ViewNavTag extends BodyTagSupport {

    // Notes objects
    private Session m_nsesCurrent;
    private ViewNavigator m_vwnav;
    private ViewEntry m_vwentCurrent;

    // tag attributes
    private String m_sDatabase;
    private String m_sView;
    private String m_sSessionKey;
    private String m_sCategory;
    private String m_sMaxLevel;

    /*************** tag handler methods ***************/
    public int doStartTag() throws JspTagException {
        ViewEntry vwent = null;
        try {
            // validate category and maxlevel parameters (both cannot be present)
```
if (m_sCategory != null && m_sMaxLevel != null) {
    throw new JspTagException("Error in redbook:viewNav tag: " +
        "Both category and maxlevel parameters may not be present");
}

// parse max level parameter
int iMaxLevel = 0;
if (m_sMaxLevel != null) {
    try {
        iMaxLevel = Integer.parseInt(m_sMaxLevel);
    } catch (NumberFormatException nfe) {
        throw new JspTagException("Error in redbook:viewNav tag: " +
            "Cannot convert maxlevel parameter to an integer");
    }
}

// get Domino session which was stored as a Request attribute
HttpServletRequest req =
    (HttpServletRequest) pageContext.getRequest();
m_nsesCurrent = (Session) req.getAttribute(m_sSessionKey);
if (m_nsesCurrent==null) {
    throw new JspTagException("Error in redbook:viewNav tag: Notes session does not exist");
}

// get database and view specified in (required) tag parameters
Database ndbToRead =
    m_nsesCurrent.getDatabase(m_nsesCurrent.getServerName(),
        m_sDatabase, false);
if (!ndbToRead.isOpen()) { 
    ndbToRead.open();
}

View vwToRead = ndbToRead.getView(m_sView);
if (vwToRead == null) {
    throw new JspTagException("Error in redbook:viewNav tag: View does not exist");
}

// create ViewNavigator to be used by doAfterBody()
if (m_sCategory != null) {
    m_vwnav = vwToRead.createViewNavFromCategory(m_sCategory);
} else if (m_sMaxLevel != null) {

Domino and WebSphere Together
m_vwnav = vwToRead.createViewNavMaxLevel(iMaxLevel);
}
else {
    m_vwnav = vwToRead.createViewNav();
}

//initialize current ViewEntry
m_vwentCurrent = m_vwnav.getFirst();
return BodyTag.EVAL_BODY_TAG;
}

try { 
    if (m_nsesCurrent != null)
        m_nsesCurrent.recycle();
}
catch (Exception e) {
    e.printStackTrace();
    throw jte;
}

catch (JspTagException jte) {
    jte.printStackTrace();
    try {
        if (m_nsesCurrent != null)
            m_nsesCurrent.recycle();
    }
catch (Exception e) {
        e.printStackTrace();
        throw jte;
    }

    catch (NotesException ne) {
        ne.printStackTrace();
        try {
            if (m_nsesCurrent != null)
                m_nsesCurrent.recycle();
        }
catch (Exception e) {
            e.printStackTrace();
            throw new JspTagException("Error in redbook:viewNav tag: " +
                "NOTES ERROR #" + ne.id + " " + ne.text);
        }

    }
catch (Exception e) {
        e.printStackTrace();
        try {
            if (m_nsesCurrent != null)
                m_nsesCurrent.recycle();
        }
catch (Exception e2) {
            e2.printStackTrace();
            throw new JspTagException("Error in redbook:viewNav tag: " + e);
        }
    }
}
public int doAfterBody() throws JspTagException {
    try {
        if (m_vwnav != null && m_vwentCurrent != null) {
            // output body contents
            BodyContent body = getBodyContent();
            JspWriter out = body.getEnclosingWriter();
            out.print(body.getString());
            body.clearBody(); // clear for next evaluation

            // go to next view entry
            m_vwentCurrent = m_vwnav.getNext();
            return BodyTag.EVAL_BODY_TAG;
        }
        // finished processing view entries
        return Tag.SKIP_BODY;
    }
    catch (NotesException ne) {
        ne.printStackTrace();
        try {
            if (m_nsesCurrent != null)
                m_nsesCurrent.recycle();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        throw new JspTagException(
            "Error in redbook:viewNav tag: " +
            "NOTES ERROR #" + ne.id + " " + ne.text);
    }
    catch (Exception e) {
        e.printStackTrace();
        try {
            if (m_nsesCurrent != null)
                m_nsesCurrent.recycle();
        }
        catch (Exception e2) {
            e2.printStackTrace();
        }
        throw new JspTagException("Error in redbook:viewNav tag: " + e);
    }
}

/**************************** Notes object getters ****************************/
public Session getNotesSession() {
    return m_nsesCurrent;
}
public ViewEntry getVwentCurrent() {


return m_vwentCurrent;
}

/******************* tag attribute setters ******************/
public void setDatabase(String database) {
    m_sDatabase = database;
}
public void setView(String view) {
    m_sView = view;
}
public void setSessionKey(String sessionKey) {
    m_sSessionKey = sessionKey;
}
public void setCategory(String category) {
    m_sCategory = category;
}
public void setMaxlevel(String maxlevel) {
    m_sMaxLevel = maxlevel;
}
}

8.5.1.2 doStartTag() method
First, we validate the category and maxlevel attributes if they were provided. Note that both cannot be present. Next, we obtain a reference to the current Domino session, which we assume has already been stored as an attribute of the HttpServletRequest. If the Domino session is not found, an exception is thrown. We will discuss exception handling in more detail later.

From the Domino session we obtain references to the database and view specified in the attributes. Once we have a reference to the Domino view, we can create the ViewNavigator object which the tag represents. Note that we call different create methods based on the attributes provided.

Finally, we store the ViewNavigator and first ViewEntry in member fields and return BodyTag.EVAL_BODY_TAG to indicate that the body of the tag should be evaluated.

We store ViewNavigator and the current ViewEntry in member fields because the doAfterBody() method will access them to traverse the view entries. Also, the nested tags will require access to the current ViewEntry for their processing. This is why we have provided a public getter method for the current ViewEntry.
Similarly, we have stored the Domino session in a member field and have provided it a public getter method. If all goes well, we assume that the JSP author will recycle the Domino session. But, if an exception occurs in this class or one of the nested tag handler classes, it is the tag handler’s responsibility to clean up the Domino session. Therefore, the nested tag handler classes will need access to the Domino session so they can execute the recycle() method.

In order to support the JSP error handling page directive 

```html
<%@ page errorPage="error.jsp" %>
```

we must throw a JspTagException from our tag handler classes for exceptions we do not want to handle locally. Notice that we declare the tag handler methods as throwing JspTagException and that when we catch a NotesException, we convert it into a new JspTagException and throw it.

When a fatal exception occurs and we want to stop processing the current tag, we must call recycle() on the current Domino session to free up its memory. That is why we have a global catch block for JspTagException where we just throw it again. Otherwise we would have to repeat the code to recycle the Domino session at every point we throw the JspTagException.

### 8.5.1.3 doAfterBody() method

In the doAfterBody() method, we first check to see that there is a current ViewEntry. If so, we then output the contents of the tag’s body and advance to the next ViewEntry.

If we have finished processing all the entries in the ViewNavigator, then the current ViewEntry is null and we return Tag.SKIP_BODY.

## 8.5.2 `<redbook:viewEntry>` tag

The `<redbook:viewEntry>` tag represents the current ViewEntry being processed. Based on the `type` parameter, it will control whether the body content is output or not. This depends on whether the current ViewEntry is a document, category, or total.

### 8.5.2.1 ViewEntryTag tag handler class

The following is the source code for the ViewEntryTag class:

```java
/**
 * File: ViewEntryTag.java
 * Defines the behavior of the `<redbook:viewEntry>`
 * custom tag
 */
package com.lotus.redbook.banking.taglib;
```
import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

import lotus.domino.*;

public class ViewEntryTag extends BodyTagSupport {

private Session m_nsesCurrent;
private ViewEntry m_vwentCurrent;

private String m_sEntryType;

/******* tag handler methods ******* /
public int doStartTag() throws JspTagException {
    try {
        // validate type attribute
        if (m_sEntryType != null
             && !m_sEntryType.equalsIgnoreCase("document")
             && !m_sEntryType.equalsIgnoreCase("category")
             && !m_sEntryType.equalsIgnoreCase("total")
        ) {
            throw new JspTagException(
                "Error in redbook:viewEntry tag: Illegal type attribute");
        }

        // get reference to enclosing <redbook:viewNav> tag
        ViewNavTag viewnavtag = (ViewNavTag) findAncestorWithClass(this, ViewNavTag.class);
        if (viewnavtag == null) {
            throw new JspTagException(
                "Error in redbook:viewEntry tag: Nesting Error");
        }

        // get current ViewEntry being processed
        ViewEntry vwent = viewnavtag.getVwentCurrent();
    }
}

// Notes objects
// Servlet classes
// JSP classes
// Domino classes

Chapter 8. JavaServer Pages  327
// store current ViewEntry for use by nested tags
m_vwentCurrent = vwent;

// store current Domino session
m_nsesCurrent = viewnavtag.getNotesSession();

// based on type attribute, decide whether to output body or not
if (vwent != null) {
    if (m_sEntryType == null ||
        (m_sEntryType.equalsIgnoreCase("document") &&
         vwent.isDocument()) ||
        (m_sEntryType.equalsIgnoreCase("category") &&
         vwent.isCategory()) ||
        (m_sEntryType.equalsIgnoreCase("total") &&
         vwent.isTotal())
    ) {
        return BodyTag.EVAL_BODY_TAG;
    }
}
return Tag.SKIP_BODY;
}

try {
    if (m_nsesCurrent != null)
        m_nsesCurrent.recycle();
}
catch (Exception e) {
    e.printStackTrace();
}
throw new JspTagException(
    "Error in redbook:viewEntry tag: " +
    "NOTES ERROR #" + ne.id + ": " + ne.text);
}
e.printStackTrace();
try {
   if (m_nsesCurrent != null)
      m_nsesCurrent.recycle();
}
catch (Exception e2) {
   e2.printStackTrace();
}
throw new JspTagException("Error in redbook:viewNav tag: " + e);
}
}

public int doAfterBody() throws JspTagException {
try {
   // output body contents
   BodyContent body = getBodyContent();
   JspWriter out = body.getEnclosingWriter();
   out.print(body.getString());
   body.clearBody(); // clear for next evaluation
   return Tag.SKIP_BODY;
}
catch (Exception e) {
   e.printStackTrace();
   try {
      if (m_nsesCurrent != null) m_nsesCurrent.recycle();
   }
catch (Exception e2) {
      e2.printStackTrace();
   }
   throw new JspTagException("Error in redbook:viewNav tag: " + e);
}
}

/****************** Notes object getters ******************/
public Session getNotesSession() {
   return m_nsesCurrent;
}
public ViewEntry getVwentCurrent() {
   return m_vwentCurrent;
}

/****************** tag attribute setter ******************/
public void setType(String type) {
   m_sEntryType = type;
}
8.5.2.2 doStartTag() method

After we validate the type parameter, we obtain a reference to the enclosing ViewNavTag tag handler object. Since we do not know if the page author nested the tags properly in the JSP source, the findAncestorWithClass() method may return null. In this case we throw a JspTagException.

We then get the current ViewEntry from the enclosing tag and store it for use by the contained <redbook:viewEntryColumn> and <redbook:viewEntryIndent> tags. Similarly, we store the current Domino session in case we have to execute recycle() in a catch block.

Finally, we decide whether to output the tag’s body contents or not. This decision is made by comparing the type tag attribute to the type of the ViewEntry, obtained using the isDocument, isCategory() and isTotal() methods.

8.5.3 <redbook:viewEntryColumn> tag

To display an actual column value from the view, the <redbook:viewEntryColumn> tag is used. It has a required column attribute which specifies which column in the current ViewEntry to display.

8.5.3.1 ViewEntryColumnTag tag handler class

In this case the tag does not have a body, so the tag handler class extends TagSupport. The following is the source code for the ViewEntryColumnTag class:

```java
package com.lotus.redbook.banking.taglib;

// Java classes
import java.io.*;
import java.util.*;

// Servlet classes
import javax.servlet.*;

// JSP classes
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

// Domino classes
```
import lotus.domino.*;

public class ViewEntryColumnTag extends TagSupport {

    // tag attribute
    private String m_sColumnNum;

    public int doStartTag() throws JspTagException {
        Session nses = null;
        try {
            // validate column parameter
            int iColumn = 0;
            try {
                iColumn = Integer.parseInt(m_sColumnNum);
            } catch (NumberFormatException nfe) {
                throw new JspTagException(
                    "Error in redbook:viewEntryColumn tag: Invalid Column Number");
            }
            if (iColumn < 1) {
                throw new JspTagException(
                    "Error in redbook:viewEntryColumn tag: Invalid Column Number");
            }

            ViewEntryTag viewEntryTag = (ViewEntryTag) findAncestorWithClass(this, ViewEntryTag.class);
            if (viewEntryTag == null) {
                throw new JspTagException(
                    "Error in redbook:viewEntryColumn tag: Nesting Error");
            }

            nses = viewEntryTag.getNotesSession();

            ViewEntry vwent = viewEntryTag.getVwentCurrent();
            if (vwent != null) {
                Vector vColumns = vwent.getColumnValues();
                Object objColval = null;
                try {
                    objColval = vColumns.elementAt(iColumn - 1);
                } catch (ArrayIndexOutOfBoundsException e) {
                    objColval = "ERROR: no such column";
                }
                if (objColval != null) {
                    JspWriter out = pageContext.getOut();
                    out.println("Column ");
                    out.println(iColumn);
                    out.println(" value: ");
                    out.println(objColval);
                    out.println("\n");
                }
            } else {
                throw new JspTagException("viewEntryTag is null");
            }
        } finally {
            // release resources
        }
    }

    // other methods...
}

Chapter 8. JavaServer Pages  331
out.print(objColval);
}
}

return Tag.SKIP_BODY;
}
catch (JspTagException jte) {
ejte.printStackTrace();
try {
    if (nses != null) nses.recycle();
} catch (Exception e) {
    e.printStackTrace();
    throw jte;
}
catch (NotesException ne) {
    ne.printStackTrace();
    try {
        if (nses != null) nses.recycle();
    } catch (Exception e) {
        e.printStackTrace();
        throw new JspTagException("Error in redbook:viewEntryColumn tag: " + "NOTES ERROR #" + ne.id + " " + ne.text);
    }
catch (Exception e) {
        e.printStackTrace();
        try {
            if (nses != null) nses.recycle();
        } catch (Exception e2) {
            e2.printStackTrace();
            throw new JspTagException("Error in redbook:viewNav tag: " + e);
        }
    }
}

*************** tag attribute setter ***************
public void setColumn(String column) {
    m_sColumnNum = column;
}
}
8.5.3.2 doStartTag() method
First, we must test the validity of the column attribute. In this implementation we decided that a value of "1" will indicate the first column. When we call the elementAt() method on the vector of column values, we catch the possible ArrayIndexOutOfBoundsException and display an error message. In this case we decided to allow the JSP page to continue processing.

8.5.4 <redbook:viewEntryIndent> tag
To assist in laying out the view data, we provide the <redbook:viewEntryIndent> tag, which outputs the indent level of the current ViewEntry.

8.5.4.1 ViewEntryIndentTag tag handler class
The following is the source code for the ViewEntryIndentTag class:

```java
/**
 * File: ViewEntryIndentTag.java
 * Defines the behavior of the <redbook:viewEntryIndent>
 * custom tag
 */

package com.lotus.redbook.banking.taglib;

// Java classes
import java.io.*;
import java.util.*;

// Servlet classes
import javax.servlet.*;

// JSP classes
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

// Domino classes
import lotus.domino.*;

public class ViewEntryIndentTag extends TagSupport {

*************** tag handler methods ***************
public int doStartTag() throws JspTagException {
    Session nses = null;
    try {
        ViewEntryTag viewEntryTag =
            (ViewEntryTag) findAncestorWithClass(this, ViewEntryTag.class);
```

Chapter 8. JavaServer Pages  333
if (viewEntryTag == null) {
    throw new JspTagException(
            "Error in redbook:viewEntryIndent tag: Nesting Error";
}

nses = viewEntryTag.getNotesSession();

ViewEntry vwent = viewEntryTag.getView WentCurrent();
if (vwent != null) {
    JspWriter out = pageContext.getOut();
    out.print(vwent.getIndentLevel());
}

return Tag.SKIP_BODY;
}
catch (JspTagException jte) {
    jte.printStackTrace();
    try {
        if (nses != null) nses.recycle();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    throw jte;
}
catch (NotesException ne) {
    ne.printStackTrace();
    try {
        if (nses != null) nses.recycle();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    throw new JspTagException(
            "Error in redbook:viewEntryIndent tag: " +
            "NOTES ERROR #" + ne.id + " " + ne.text);
}
catch (Exception e) {
    e.printStackTrace();
    try {
        if (nses != null) nses.recycle();
    }
    catch (Exception e2) {
        e2.printStackTrace();
    }
    throw new JspTagException("Error in redbook:viewNav tag: " + e);
}
8.5.5 TLD file

To describe the custom tags to the JSP 1.1 Processor, we created the following TLD file:

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tablib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tag library descriptor -->
<taglib>
    <!-- after this the default space is "http://java.sun.com/j2ee/dt/dts/web-jsptaglibrary_1_2.dtd" -->
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>redbook</shortname>
    <urn></urn>
    <info>Test for Redbook</info>
</taglib>

<tag>
    <name>viewNav</name>
    <tagclass>com.lotus.redbook.banking.taglib.ViewNavTag</tagclass>
    <info>Outer tag, iterates over view entries in the specified view</info>
    <bodycontent>JSP</bodycontent>

    <attribute>
        <name>database</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

    <attribute>
        <name>view</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

    <attribute>
        <name>sessionKey</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
```
<attribute>
  <name>category</name>
  <required>false</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
  <name>maxlevel</name>
  <required>false</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
<tag>
  <name>viewEntry</name>
  <tagclass>com.lotus.redbook.banking.taglib.ViewEntryTag</tagclass>
  <info>Second level tag, represents a single view entry in a view entry navigator</info>
  <bodycontent>JSP</bodycontent>
  <attribute>
    <name>type</name>
    <required>false</required>
  </attribute>
</tag>
<tag>
  <name>viewEntryColumn</name>
  <tagclass>com.lotus.redbook.banking.taglib.ViewEntryColumnTag</tagclass>
  <info>displays the current column value</info>
  <bodycontent>EMPTY</bodycontent>
  <attribute>
    <name>column</name>
    <required>true</required>
  </attribute>
</tag>
<tag>
  <name>viewEntryIndent</name>
  <tagclass>com.lotus.redbook.banking.taglib.ViewEntryIndentTag</tagclass>
  <info>displays the indent level of the current view entry</info>
  <bodycontent>EMPTY</bodycontent>
</tag>
</taglib>

336  Domino and WebSphere Together
Refer to 8.4.2.2, “Tag library descriptors” on page 308 for details on the structure of TLD files.

8.5.6 Where to create the Domino session

We designed the ViewNavigator custom tags assuming that a controller servlet had already created the Domino session and stored it as a request attribute. Another option available to us is creating the Domino session in the doStartTag() method of ViewNavTag. There are advantages and disadvantages to both methods.

One key advantage of creating the Domino session in doStartTag() is that we do not have to create and maintain a controller servlet. Another advantage is that we can then call its recycle() method in the doEndTag() method. In our example, the JSP author must remember to include scriptlet code after the tags to recycle the Domino session. If this step is omitted, the sessions will remain in memory in the Domino server, causing decreased performance. By creating in doStartTag() and recycling in doEndTag(), our custom tag is a self-contained unit and the JSP author does not have to deal with the Domino session.

The foremost disadvantage is with the handling of runtime exceptions. In JSP, you surrender the ability to have fine-grained runtime exception handling, as you have with a servlet. Instead, runtime exceptions are handled for you automatically by the PageContext object. For error handling, your choices are to forward to an error page using the <%@ page errorPage="error.jsp" %>
page directive or let the JSP processor handle the error. Also, recall that when implementing custom tag handlers, you can only throw exceptions of type JspTagException from the tag handler methods.

There is great potential for errors to occur when creating a Domino session. This is especially true when working in an SSO environment where the WebSphere administrator can set a short life span for the LPTA token. In our example application, when the user’s token has expired, we want to forward to a dynamic URL where the user can log in again. This behavior is more complicated because the JSP page is shown in a frame and we wanted the forward to occur at the top frameset level in the Web browser.

In JSP, implementing an error handling behavior other than forwarding to a static URL is complex. It would require the inclusion of custom exception handling code in scriptlets. But this strategy defeats the goal of minimizing Java code in the JSP page.
For this reason, we decided to create the Domino session in a controller servlet where it can handle most runtime exceptions before the JSP page is invoked. This follows the Model/View/Controller methodology and provides a cleaner separation between the data access and presentation layers. However, if the default JSP error handling is sufficient for another application, then creating the Domino session in doStartTag() may make more sense.

Another advantage to creating the Domino session in a controller servlet is that the session can be shared by many tags on a single page. One JSP page in our banking example uses the $<redbook:viewNav>$ tag twice. By creating the Domino session once in the servlet both tags can use it. Otherwise, each tag would create their own Domino session, which is inefficient.

8.5.6.1 ReadNames example revisited
To demonstrate a tag that creates a Domino session itself, we created another version of the ViewNavTag class. In doStartTag() of this new version, the Domino session is created from the LTPA rather than “acquired” from the HttpServletRequest. The other tag handler class files remained the same.

Using the new custom tag, we created a new version of the ReadNames “application” from Chapter 4, “WebSphere - Domino security and single sign-on” on page 109 without any servlet involved.

The example is comprised of the following files:
- SessionViewNav.jar
- SessionViewNavSource.jar
- redbook-taglib2.tld
- readnames.jsp

These files are available as part of the additional Web material in the file 5955jsp-sessiontaglib.zip. For details on how to get these files see Appendix K, “Using the additional Web material” on page 521.

You can deploy the files in the same way as we did in 8.4.3, “Deploying custom tag libraries” on page 312. Remember to stop and start the application server before testing.

Note that, just like in the original ReadNames example, you have to access a secured resource first to be prompted to authenticate and get the LTPA token cookie passed back. You can, for example, do this by accessing Domino Directory like this:

http://odin.lotus.com/names.nsf
Once you have a valid LTPA token, you can try the JSP-only version of ReadNames with a URL similar to this:

http://odin.lotus.com/webapp/Redex/readnames.nsf

The response should look like Figure 216.

---

8.5.7 Using JSP custom tags to create scripting variables

In all of the custom JSP tags we have discussed in this chapter, the final result of the tags is to insert values into the HTML. Another useful feature of custom tags is that they can generate objects that are accessible from server-side script. We do not use such tags in our banking application, but we have made a small sample to illustrate the use of custom tags for scripting variables. We show how to do this in Appendix H, “JSP custom tags that create scripting variables” on page 511.

8.5.8 Other Domino JSP examples

To illustrate how our JSP ViewNavigator tag library allows total separation between storage in a Domino database and how to present that data in a browser, we created three JSP page variations of the People view in Domino Directory. Our JSP samples show content from the People view formatted as:

- An HTML table
- An Excel spreadsheet
• A tabbed table created using JavaScript

In Appendix I, “Variations of the People view using JSP custom tags” on page 515 you can see more about these small samples.

This concludes our discussion of the Java tag library code. We next discuss the JSP files and the servlet we created for our example application.

8.6 Navigation tree JSP in our sample

Refer to Chapter 5, “Introduction to the example application” on page 183 for a description of the banking example application. In this section we discuss the content navigation tree that sits in the left frame.

8.6.1 Overview of the navigation tree

Figure 217 shows the navigation tree in the left frame of the main frameset.

Figure 217. Banking example: navigation tree

Recall that all content documents belong to one, and only one, information area. These areas are in turn classified using one or more levels of categories. In our application, the top-level category for the area is called the Division. In our sample we work with divisions like Sales, Development, Communication, and so on. Figure 217 lists all the areas in the Communication division.
The top `<select>` drop-down list holds a list of all the divisions (that is, top level categories). Changing the division will refresh the frame to display the areas in the new division under the drop-down list. Also, each division can have its own home page, which will be displayed in the middle document frame when the division initially is selected.

The folders in the navigation tree are subcategories used to group similar areas together. Under the folders are the names of actual areas. Clicking on an area name like Human Resources will display a list of the area’s contents in the middle document frame. An example of this content list is shown in Figure 218.

<table>
<thead>
<tr>
<th>Content Name</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your Career</td>
<td>05/16/2001 12:00:32 PM 05/16/2001 12:04:44</td>
</tr>
<tr>
<td>Your Life</td>
<td>05/16/2001 12:05:07 PM 05/16/2001 12:09:5</td>
</tr>
<tr>
<td>Your Health</td>
<td>05/16/2001 12:06:37 PM 05/16/2001 12:09:5</td>
</tr>
<tr>
<td>Your Money</td>
<td>05/16/2001 12:07:52 PM 05/16/2001 12:09:5</td>
</tr>
</tbody>
</table>

*Figure 218. List of content documents in the Human Resources area*

Then, in the document frame, clicking on a document title will bring up the document in the same frame.

We discuss the actual JSP code used to build the navigator page a little later, but first we briefly describe the Domino views and forms used in the content application. We also discuss the servlet we created to manage setup and forwarding to our JSP pages.

### 8.6.2 Domino elements

The content creator assigns an information area to their document when they are working with it in the authoring database. In Figure 219 on page 342 you can see the content author’s view of what will be shown in the navigation tree JSP.
The documents that are approved are transferred to the published documents database by a Domino agent. While testing, we invoked it from the Actions menu, but it might as well run as scheduled.

In the published documents database (intra_01.nsf), the view shown in Figure 220 is used as a single category view to display documents belonging to a certain area.

Figure 219. Published documents view in content author database by content area

Figure 220. Domino view used to show single area category
The form used to display the single category view with the area content list is shown in Figure 221.

![Figure 221. Area content list form](image)

Clicking an area name in the left navigation menu JSP page will call the pictured form via a URL such as 

```
/intra_01.nsf/fo.AreaContentList?OpenForm&CSTE-4TGLBW
```

When calling the form via a URL, Domino will automatically populate the hidden `Query_String` field with the query string of the HTTP request. For the previous URL the value for the `Query_String` field would be “OpenForm&CSTE-4TGLBW”. The value “CSTE-4TGLBW” is a unique identifier of an area the user has clicked on.

The other element in the form, the embedded view, is a list of content documents categorized by the unique area identifier. For this application we only display the content for a single area specified in the query string. As
shown in Figure 221, we accomplish this by entering the formula @Right(Query_String; "&") as the Show single category value.

This was a very basic overview of the Domino design elements used. The Domino databases in our sample application are available for download, so you can continue the exploration of the Domino elements on your own. See Appendix K, “Using the additional Web material” on page 521 for instructions on how to get the databases.

We now look at the the servlet we created to control our JSP pages.

8.6.3 The CallDominoJsp controller servlet

In keeping with the Model/View/Controller programming model, we created a servlet which performs the required checking and setup before invoking the JSP page. The goal is to have the JSP page deal solely with presentation issues. Complex Java code should occur only in servlets, JavaBeans, EJBs and custom tags.

Recall that the ViewNavigator custom tags developed in 8.5, “Custom tag example: ViewNavigator tags” on page 319 expected that a Domino session was already created and stored as an attribute of the request object. The controller servlet basically creates the Domino session object, stores it in the request, and forwards the request to a JSP page.

Refer to Chapter 7, “Servlets” on page 233 for information on servlets and details on the other servlets used in the banking example. Specifically, you should review 7.3.1, “The BankingServletTemplate class” on page 242 and 7.3.2, “ApplicationInfo class” on page 246, which discuss the BankingServletTemplate and ApplicationInfo classes. Like the other servlets in the example, the controller servlet extends BankingServletTemplate and uses the ApplicationInfo object.

The CallDominoJsp servlet is meant to be a generic servlet that can be used to forward to any JSP that requires a Domino session. In the banking example, we use the same servlet for both the top frame JSP and the navigation tree. By passing parameters in the HTTP query string, we can invoke specialized behavior as necessary. For example, the viewnav.jsp file that creates the navigation tree requires the current division, which is stored in a cookie. By including division=true in the query string, we indicate we want to pass the current division to the JSP. For the navigation tree, we invoke the servlet as follows:

.../CallDominoJsp?forward=/viewnav.jsp&division=true
Following is the source code for the CallDominoJsp servlet. (It is also available in BankingSource.jar in 5955java.zip in the Additional Web material.)

```java
package com.lotus.redbook.banking;

// Java classes
import java.util.*;
import java.io.*;

// servlet classes
import javax.servlet.*;
import javax.servlet.http.*;

// Domino classes
import lotus.domino.*;

// EJB classes
import com.lotus.redbook.banking.ejb.*;
import com.ibm.ejs.doc.account.*;
import java.rmi.RemoteException;
import javax.ejb.*;

public class CallDominoJsp extends BankingServletTemplate {

public void doGet(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {

  // check that the forward query string parameter was received
  String sForwardParam = (String) req.getParameter("forward");
  if (sForwardParam==null) {
    printHTML(getPrintWriter(res),
      "Forward parameter is missing", true, true);
    return;
  }

  HttpSession httpSes = req.getSession(true);
```

Chapter 8. JavaServer Pages  345
ApplicationInfo objApp = null;
if (httpSes != null)
    objApp = (ApplicationInfo) httpSes.getAttribute("AppInfo");

if (objApp != null) {
    Hashtable hParameters = getAllServletParameters(req);
    String sLtpaToken = getCookie(req, "LtpaToken");
    getDbPath(req);
    if (sLtpaToken == null) {
        generateError(req, getPrintWriter(res),
                        "Search:doGet", 1213, "User not logged in");
    } else {
        // create Domino session and store as request attribute
        Session nses = null;
        try {
            nses = NotesFactory.createSession(
                    req.getServerName(), sLtpaToken);
        } catch (NotesException ne) {
            generateError(req, getPrintWriter(res),
                          "LeftMenu:doGet", ne.id, ne.text);
            return;
        }
        req.setAttribute("notesSession", nses);

        // if division parameter set, get current division from cookie
        // and store as request attribute
        String sDivisionParam =
                getServletParameter(hParameters, "division");
        if (sDivisionParam != null &&
            sDivisionParam.equalsIgnoreCase("true")
        ) {
            // get current division from cookie
            // and store as request attribute
            String sDivision = getCookie(req, "IntranetDivision");
            if (sDivision == null) sDivision = ";
            req.setAttribute("divisionName", sDivision);
        }

        // if banking parameter set, get user's bank accounts info
        // and store as request attribute as JavaBean
        String sBankingParam =
                getServletParameter(hParameters, "banking");
        if (sBankingParam != null &&
            sBankingParam.equalsIgnoreCase("true")
        ) {
// get user's bank account data
BankAccountsInfo beanAccounts =
    getBankAccountsInfo(req, res, objApp);
if (beanAccounts == null) // error occurred
    return;

    req.setAttribute("accountsInfo", beanAccounts);
}

// prevent caching of the response
res.setHeader("Pragma", "no-cache");
res.setHeader("Cache-Control", "no-cache");
res.setDateHeader("Expires", 0);

// forward response to JSP page specified
// by forward parameter
ServletContext sc = getServletContext();
RequestDispatcher rd =
    sc.getRequestDispatcher(sForwardParam);
rd.forward(req, res);
}
}

else if (getDbPath(req).compareTo("") != 0) {
    generateError(req, getPrintWriter(res),
        "Search:doGet", 1213, "User not logged in");
}
else {
    printHTML(getPrintWriter(res),
        "No application info available!", true, true);
}

/**
 * Return the value of a specified cookie from
 * the current request
 */
private String getCookie(HttpServletRequest req, String sCookieName) {
    try {
        Cookie[] cookies = req.getCookies();
        if (cookies != null) {
            for (int i = 0; i < cookies.length; i++) {
                if (cookies[i].getName().equalsIgnoreCase(sCookieName)) {
                    return cookies[i].getValue();
                }
            }
        }
    }
}
To forward to the JSP, we create a RequestDispatcher using the JSP page’s URL. The RequestDispatcher looks in the document root of the servlet’s Web application for the Web resource.

For the navigation tree we did not include a banking parameter in the query string. We use the banking parameter for the top frame where we display the user’s bank account balances. The getBankAccountsInfo() method is not included in the preceding source code listing since we do not call it here. It is discussed in 8.7, “Banking example: Top frame JSP” on page 358.

Note that this class has its own getCookie() method, which does not pass a PrintWriter as the one in BankingServletTemplate does. The getPrintWriter() method defined in BankingServletTemplate actually outputs some HTML before returning the PrintWriter to the calling method. However, when we are using the RequestDispatcher to forward the request, we are not allowed to output HTML first. So in the doGet() method we only call getPrintWriter() when an error has occurred and we know that we will not be forwarding to the JSP page.

### 8.6.4 JSP source for viewnav.jsp

To create the navigation tree, we used a JSP page that employs the custom ViewNavigator tags that we developed in 8.5, “Custom tag example: ViewNavigator tags” on page 319. Here is the source code for the JSP page (it is also available as viewnav.jsp in 5955deploy.zip, which is part of the additional Web material):

```html
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page import="lotus.domino.Session" errorPage="/error.jsp" %>
<%@ taglib uri="redbook-taglib.tld" prefix="redbook" %>

<jsp:useBean id="AppInfo" scope="session" type="com.lotus.redbook.banking.ApplicationInfo" />
<jsp:useBean id="divisionName" scope="request" type="String" />

String sServer = request.getServerName();
String sDb = AppInfo.getBeanPath("01");
```
<%-- link to CSS document stored in Notes database --%>
<link rel="stylesheet" type="text/css"
href="http://<%= sServer %>/<%= AppInfo.getBrowser(true) %>">

<script language="JavaScript1.3">
var NUM_DAYS = 10;

function on_load() {
    // select current division in <select>
    var division = "<%= divisionName %>";
    var divisions = document.forms["frm"].divisionList.options;
    var i;

    for (i = 0; i < divisions.length; i++) {
        if (divisions[i].value == division) {
            divisions[i].selected = true;
        }
    }
}

function setCookie(name, value, expire) {
    document.cookie = name + "=" + value
    + ((expire == null) ? "" : ("; expires=" + expire.toGMTString());
}

function newDivision( slct ) {
    // determine cookie expiration date
    var date = new Date();
    date.setTime(date.getTime() + (86400 * 1000 * NUM_DAYS));

    // get new division
    var division = slct.options[slct.selectedIndex].value;

    // set the cookie
    setCookie("IntranetDivision", division, date);

    // refresh menu
    window.location.reload();

    // call the new division's home page in the middle frame
    parent.frames["Middle"].location=
        "http://<%= sServer %>/<%= sDb %>/" +
In the following sections we break the JSP source code into sections and discuss each one separately.

### 8.6.4.1 Initial setup

First we have a page declaration which imports the Domino `Session` class and declares `error.jsp` as the resource to forward to when a runtime exception occurs.
Next we obtain references to the two JavaBeans to be used in the page. The AppInfo bean is an object of type ApplicationInfo, discussed in 7.3.2, “ApplicationInfo class” on page 246. It contains application-specific settings. The second bean is a String that represents the current division selected. The controller servlet gets this value from a cookie and passes it to the JSP as a request attribute. We set the cookie using client-side script, as discussed in 8.6.4.3, “Client-side script” on page 352.

8.6.4.2 CSS properties link
<html>
<head>

<!-- link to CSS document stored in Notes database --%>
<link rel="stylesheet" type="text/css"
href="http://<%= sServer %>/<%= AppInfo.getBrowser(true) %>">

The ApplicationInfo class defines the getBrowser() method, which returns a URL to a browser-specific CSS file. This file is stored as an attachment to a Domino document. In our case, getBrowser(true) returns intranet/intra_01.nsf/viref_foFiles$ByName/NetscapeNavigator6/$File/ns6.css for a Netscape Navigator Web browser.

By storing layout and formatting properties in a different file, we separate the content of the page from the page’s design, which makes for easier maintenance. The CSS properties used to lay out the navigation tree are discussed in 8.6.4.5, “Menu tree” on page 355.

8.6.4.3 Client-side script
<script language="JavaScript1.3">
var NUM_DAYS = 10;

function on_load() {
    // select current division in <select>
    var division = "<%= divisionName %>";
    var divisions = document.forms["frm"].divisionList.options;
    var i;

    for (i = 0; i < divisions.length; i++) {
        if (divisions[ i ].value == division) {
            divisions[ i ].selected = true;
        }
    }
}

function setCookie(name, value, expire) {

document.cookie = name + "=" + value
+ ((expire == null) ? "" : ("; expires=" + expire.toGMTString()));

function newDivision( slct ) {
// determine cookie expiration date
var date = new Date();
date.setTime(date.getTime() + (86400 * 1000 * NUM_DAYS));

// get new division
var division = slct.options[ slct.selectedIndex ].value;

// set the cookie
setCookie("IntranetDivision", division, date);

// refresh menu
window.location.reload();

// call the new division's home page in the middle frame
parent.frames["Middle"].location=
  "http://<%= sServer %>/<%= sDb %>/" +
  "viref_foFrameset$ByName/ContentFrameset?OpenDocument";
}
</script>
</head>
<body onload="on_load()">

We define three JavaScript functions in the <head> section of the HTML document:

- **on_load**
  The on_load() function is called from the HTML body’s onLoad event, which occurs after the document is loaded. Here we are selecting the current division’s option in the top <select>. The CallDominoJsp servlet reads the current division from a cookie and stores it as a request attribute. Note the use of a JSP expression to pass the division name from server-side script to client-side script.

- **setCookie**
  The setCookie function simply creates a cookie with the name, value, and expiry date that is passed to it.

- **newDivision**
  The newDivision() function is called when the division is changed using the top <select>. This function does the following:
a. Refreshes the middle document frame to the new division’s home page
b. Sets a cookie containing the new division name by calling `setCookie()`
c. Updates the navigation tree by refreshing the left frame, which invokes the `CallDominoJSP` servlet again

Even though we have forwarded to `viewnav.jsp`, the browser stores the `CallDominoJSP` servlet path as its current location. So when the frame is refreshed the controller servlet is called again.

### 8.6.4.4 Division list `<select>`

```html
generated code
```

Here is our first use of the custom `ViewNavigator JSP tags`. For the `<select>` we want to present a list of divisions. We are using the view with the alias `viref_foContent$ByAllCat`. In Figure 222 on page 355 you can see how this view is shown in a Notes client.
Chapter 8. JavaServer Pages

8.6.4.5 Menu tree

```jsp
<%@ -- ************ MENU TREE ************ --%>

<%@ -- only show menu tree if a current division exists --%>
<%@ if (!divisionName.equals("")) { %>
<redbook:viewNav
  database="<%= sDb %>"
  view="viref_foContent$ByAllCat"
  sessionKey="notesSession"
  category=""""%>
<%@ } %>
```
The menu tree displays all the area documents for a single division. So in this case we must specify category="divisionName" for the <redbook:viewNav> tag since the division is the top-level category. Like a categorized view, we want to display the area’s subcategories as well in a hierarchical format.

To accomplish the formatting of the view entries, we used different CSS classes for categories and documents and for each different indent level. For example, if the entry is a document at the indent level 2, we generate the following line in the HTML:

```html
<div class="doc2">area name</div>
```

For a category, we create the class "headx" where “head” stands for heading and x is a number representing the indent level.

As an example, Figure 223 is a fragment from the navigation tree shown in Figure 217 on page 340.
Figure 223. Fragment of a navigation tree

The dynamically generated HTML code for the three entries is roughly:

```html
<div class="head1">Intranet</div>
<div class="doc2"><a href="...">Media</a></div>
<div class="doc2"><a href="...">Human Resources</a></div>
```

To distinguish between document and category view entries, we use both `<redbook:viewEntry type="document">` and `<redbook:viewEntry type="category">` tags. This allows us to output different HTML depending on the type of entry.

By using CSS classes, we do not need to concern ourselves with formatting issues in the JSP. For example, if we wanted to change the width of the indent, we would update the separate CSS properties file, not the JSP.

The CSS properties file is stored as an attachment to a Notes document. The following is the part of the file we created that deals with the navigation tree:

```html
<!-- Styles for dynamic JSP menu -->
div.head1 {
    text-indent: 0;
    font-size: 8pt;
}
div.head2 {
    text-indent: 15px;
    font-size: 8pt;
}
div.head3 {
    text-indent: 30px;
    font-size: 8pt;
}
div.head4 {
    text-indent: 45px;
    font-size: 8pt;
}
div.doc2 {
    text-indent: 25px;
    font-size: 8pt;
}
div.doc3 {
    text-indent: 40px;
}
```
In this very simple example we only set the text-indent property to a different number of pixels. We used the same properties for all Web browsers.

8.6.4.6 Recycle Domino session

```jsp
<%-- must recycle Notes session --%>
<% Session nses = (Session) request.getAttribute("notesSession"); if (nses != null) nses.recycle(); %>

Finally, it is up to the JSP to call the recycle() method on the Domino session. This tells the Domino server that we are finished with the session (and all Domino objects created from the session) and that its memory can be reclaimed. Domino must be told explicitly to recycle the session. It does not happen automatically when the Java reference to the Domino session is garbage collected.

In Appendix G, “HTML output from viewnav.jsp” on page 507 we have included an example of the HTML produced by the viewnav.jsp. We now look at the topframe.jsp file.

8.7 Banking example: Top frame JSP

The top frame of the banking example is noteworthy because it combines data from a Domino database and from a relational database. Like the navigator tree, the top frame uses the custom ViewNavigator JSP tags to build a list of links. Also, the JSP displays information from two EJBs whose data is stored in a relational database.
8.7.1 Elements of the top frame JSP

The top frame navigator is available as topframe.jsp in 5955deploy.zip that is part of the additional Web material. The JSP is made up of the following components:

- Domino form buttons
- Quick search field
- Links combobox
- Display of bank account information.

8.7.1.1 Domino form buttons

The two buttons open different Domino forms in the middle content frame.

8.7.1.2 Quick search field

Clicking the Go button beside the search field will call the Search servlet, adding the contents of the search field as a parameter in the query string. Refer to 7.4.3, "Search servlet" on page 261 for details on the Search servlet.
Figure 227. The search field in the top frame navigator

8.7.1.3 Links combobox

Similar to the `<select>` in the left navigation tree, the links `<select>` is also built using the custom JSP tags we developed in 8.5, “Custom tag example: ViewNavigator tags” on page 319.

Figure 228. Links combobox in the top frame navigator

The following JSP source code is used to create the `<select>`:

```html
<select name="gateway" onChange="goUrl(this)">
  <option selected>More Sites</option>

  <redbook:viewNav
    database="<%= sDb %>">
    view="viref_foContent$MainNavCombobox"
    sessionKey="notesSession"
  >
    <redbook:viewEntry>
      <option value='<redbook:viewEntryColumn column="3" />'>
        <redbook:viewEntryColumn column="2" />
      </option>
    </redbook:viewEntry>
  </redbook:viewNav>
</select>
```

The value of each `<option>` is either a full URL for an external site or a Notes document unique identifier if the link refers to an internal document. The following `goUrl()` JavaScript function is called in the `onChange()` event of the `<select>`. It inspects the value and opens a new browser if it is a URL, or opens the document in the main content frame if the value refers to a Notes content document.

```javascript
function goUrl(s) {
  var d = s.options[s.selectedIndex].value;
  iResult = d.search(/\./);
  if (iResult == -1)
    /* open content document in "middle" frame */
```
The last element is the display of the user’s bank account information. We discuss this in detail in the next section.

### 8.7.2 Accessing the Account EJB

The user’s checking and savings accounts are each represented as an EJB in the banking example. Following the Model/View/Controller programming model, we use a servlet to obtain the bank account balances from the EJBs and then create a JavaBean for use by the JSP to display the information.

#### 8.7.2.1 CallDominoJsp servlet

The same servlet we used to invoke `viewnav.jsp` for the navigation tree is also used to invoke the top frame navigator. However, this time we set the `banking` parameter to `true` in the query string as follows:

```
.../CallDominoJsp?forward=/topframe.jsp&banking=true
```

Refer to 8.6.3, “The CallDominoJsp controller servlet” on page 344 for details of the `doGet()` method of `CallDominoJsp`. If the banking parameter is set to true, then the `BankAccountInfo` method shown in the following is called to access the Account EJBs.

```java
/**
 * Get user's bank accounts from EJB and store info in JavaBean
 * Returns the bean if successful, null otherwise
 */
private BankAccountsInfo getBankAccountsInfo(
    HttpServletRequest req,
    HttpServletResponse res,
    ApplicationInfo objApp ) {

    Hashtable hPersonalization = null;
    try {
        PersonalizationHome ejbhPersonalization = (PersonalizationHome)
            getHomeObject (objApp, "Personalization");
        if (ejbhPersonalization == null) {
            generateError(req, getPrintWriter(res), "TopFrame:doGet", 0,
```
"No home interface for Personalization EJB";
return null;
}

Personalization ejbPersonalization = ejbhPersonalization.create();

hPersonalization = ejbPersonalization.getPersonalization(
   req.getServerName(), objApp.getDbPath("02"), false);
if (hPersonalization == null) {
   // try again, because null is also returned
   // if the session was dropped by the Domino server
   hPersonalization = ejbPersonalization.getPersonalization(
      req.getServerName(), objApp.getDbPath("02"), false);
}
if (hPersonalization == null) {
   // an error occurred again ... display this error
   generateError(req, getPrintWriter(res), "TopFrame:doGet", 0,
      "Cannot create personalization EJB");
   return null;
}
}
catch (CreateException ce) {
   generateError(req, getPrintWriter(res), "CallDominoJsp:doGet", 0,
      "Cannot create Personalization EJB");
   return null;
}
catch (RemoteException re) {
   generateError(req, getPrintWriter(res), "CallDominoJsp:doGet", 0,
      "Error accessing Personalization EJB");
   return null;
}

/**************** get banking data from EJB ************/

// get account numbers
int iSavingsNum =
   ((Integer)hPersonalization.get("savings")).intValue();
int iCheckingNum =
   ((Integer)hPersonalization.get("checking")).intValue();
float fSavingsBal = 0;
float fCheckingBal = 0;

// the first time a new user logs on the account information
// may not exist only check for balances if there is a
// valid account number
if (iCheckingNum > 0) {
   try {

AccountHome ejbhAccount = (AccountHome) 
getHomeObject (objApp, "Account"); 
if (ejbhAccount == null) { 
    generateError(req, getPrintWriter(res), "CallDominoJsp:doGet", 
0, "No home interface for Account EJB"); 
    return null; 
} 

// savings balance 
AccountKey objKey = new AccountKey(iSavingsNum); 
Account ejbAccount = ejbhAccount.findByPrimaryKey(objKey); 
fSavingsBal = ejbAccount.getBalance();

// checking balance 
objKey = new AccountKey(iCheckingNum); 
ejbAccount = ejbhAccount.findByPrimaryKey(objKey); 
fCheckingBal = ejbAccount.getBalance(); 
}
catch (FinderException fe) { 
    generateError(req, getPrintWriter(res), "CallDominoJsp:doGet", 0, 
"Cannot find Account EJB"); 
    return null; 
} 
catch (RemoteException re) { 
    generateError(req, getPrintWriter(res), "CallDominoJsp:doGet", 0, 
"Error accessing Account EJB"); 
    return null; 
}
} 

// construct bean and store in request 
return (new BankAccountsInfo(
iCheckingNum, iSavingsNum, fCheckingBal, fSavingsBal));

In the first section of the method we access the Personalization EJB to obtain 
the user's bank account numbers. Then, in the next section we use the 
account numbers to find the Account EJBs and get their balances. We test the 
account numbers before getting the balances because the first time a user 
logs in the accounts will not be created yet. Finally, we create a JavaBean of 
type BankAccountsInfo and store it as an attribute of the request. The JSP will 
then use the bean to output the information.

The Account EJB is a sample that ships with WebSphere. Refer to the product 
documentation for information on this EJB. For information on the 
Personalization EJB, read 9.3, “The Personalization EJB” on page 373. To
learn more about using servlets as EJB clients, refer to 9.4, “Calling EJBs that are managed by WebSphere” on page 382.

8.7.2.2 BankAccountsInfo class

The `BankAccountsInfo` JavaBean class has four read-only properties: the account numbers and balances for both checking and savings accounts.

The following is the Java source code for the `BankAccountsInfo` class:

```java
/**
 * File: BankAccountsInfo.java
 * JavaBean to store checking and savings
 * account information
 */

package com.lotus.redbook.banking;

// Java classes
import java.io.Serializable;

public class BankAccountsInfo implements Serializable {

    private int m_iSavingsAcctNum;
    private int m_iCheckingAcctNum;
    private float m_fCheckingAcctBal;
    private float m_fSavingsAcctBal;

    public BankAccountsInfo(int iCheckingAcctNum, int iSavingsAcctNum,
                            float fCheckingAcctBal, float fSavingsAcctBal) {
        m_iCheckingAcctNum = iCheckingAcctNum;
        m_iSavingsAcctNum = iSavingsAcctNum;
        m_fCheckingAcctBal = fCheckingAcctBal;
        m_fSavingsAcctBal = fSavingsAcctBal;
    }

    public float getCheckingAcctBal() {
        return m_fCheckingAcctBal;
    }

    public int getCheckingAcctNum() {
        return m_iCheckingAcctNum;
    }

    public float getSavingsAcctBal() {
        return m_fSavingsAcctBal;
    }

    public int getSavingsAcctNum() {
        return m_iSavingsAcctNum;
    }
```
8.7.2.3 JSP Source

We do not list the entire source code here since most of it is similar to viewnav.jsp. We discuss only the fragments that deal with displaying the bank account information.

```jsp
<jsp:useBean id="accountsInfo" scope="request"
    type="com.lotus.redbook.banking.BankAccountsInfo" />
```

The servlet stored the BankAccountsInfo bean as a request attribute using the key "accountsInfo." The `<jsp:useBean>` tag obtains a reference to the bean so we can display its properties.

```jsp
<% if (accountsInfo.getCheckingAcctNum() > 0) { %>
YOUR bank account information:
<table>
<tr>
    <th>Type</th>
    <th>Number</th>
    <th>Balance</th>
</tr>
<tr>
    <td>Savings</td>
    <td align="center">
        <jsp:getProperty name="accountsInfo"
            property="savingsAcctNum" />
    </td>
    <td align="right">
        $<jsp:getProperty name="accountsInfo"
            property="savingsAcctBal" />
    </td>
</tr>
<tr>
    <td>Checking</td>
    <td align="center">
        <jsp:getProperty name="accountsInfo"
            property="checkingAcctNum" />
    </td>
    <td align="right">
        $<jsp:getProperty name="accountsInfo"
            property="checkingAcctBal" />
    </td>
</tr>
</table>
<% } else { %>
Checking and savings accounts have been created for you.
```
Notice that we first check for a valid account number before displaying the account information. In our application, the first time a user logs in their accounts are created, but they are not accessible by CallDominoJsp. This is because the accounts are created immediately before CallDominoJsp is invoked and the Domino view that lists the personalization documents does not refresh in time for the account numbers to be available.

### 8.8 Using WebSphere Studio for JSP development

The JSP pages we have discussed in this chapter can be developed using a simple text editor; but in the long run, creating complex JSP pages which include scriptlets, JavaBeans, and custom tags can be cumbersome with a text editor.

You will be much more productive using a tool such as WebSphere Studio that has dedicated JSP development functionality and can assist page designers in creating, publishing, and maintaining the pages in a Web application.

Among its many features, WebSphere Studio allows you to import JavaBeans and custom tags into its visual development environment. You can then include these beans and tags in pages via drag and drop. WebSphere Studio also integrates with VisualAge for Java for increased efficiency.

For more information on WebSphere Studio, visit http://www-4.ibm.com/software/webservers/studio/

### 8.9 Summary

In this chapter we have introduced JavaServer Pages (JSP) in general, and looked at how you can use JSPs together with Domino either by passing a JavaBean with Domino data to a JSP or by using custom JSP tags. We explained how we programmed a custom tag library that enables JSP developers to access Domino views without any Domino programming knowledge.
Chapter 9. Enterprise JavaBeans

Our example application contains an EJB called Personalization to access the personalization documents that are stored in Domino. It also uses the Account and Transfer EJBs that belong to the banking example application of WebSphere 3.5.

Using these examples, we show how you can enhance your Domino / WebSphere application with EJBs that access relational databases and enterprise systems as well as Domino data.

We start by discussing when it makes sense to use EJBs to access Domino from WebSphere and then show how to use VisualAge for Java to develop and deploy EJBs.

We look at the code in the Personalization EJB and describe the different ways you can call EJBs from your Java client programs. While servlets running in WebSphere already have a lot of their environment set up, there are additional considerations if you are accessing EJBs from a standalone Java program or a Domino Java agent. We show how we access the personalization EJB from a servlet and the Transfer EJB from a Domino agent in our sample application.

9.1 When to access Domino data from an EJB

Accessing Domino data from an EJB is appropriate if:

- The resources are accessed in the same way across different servlets.
- The access is part of a larger transaction.
- The access involves capturing persistent data apart from Domino.
- The users should access the Domino data with the rights of someone else, but in a controlled way.

Note: The examples we show in this chapter are not to be considered as general design recommendations, but instead as illustrations of programming techniques to use when accessing Domino from EJBs. For example, in some situations the functionality we cover in this chapter could just as well be handled by servlets.

It is very important that you spend enough time on your application design before you start implementing EJBs. You should not use EJBs to access Domino just because you have the capability, and it may not even make
sense to use EJBs at all in your application. EJBs represent a very strong technology, but they also have an overhead, which means that you must carefully evaluate when they make sense for your application.

9.2 Working with Enterprise JavaBeans in VisualAge for Java

In 2.2.3, “Enterprise JavaBeans” on page 22 we explained what Enterprise JavaBeans are and how they are used. This chapter describes in detail how you can write and deploy an EJB using the features of VisualAge for Java.

9.2.1 Creating an Enterprise Bean in VisualAge for Java

IBM VisualAge for Java Enterprise Edition 3.5 provides an integrated environment in which to write EJBs. This feature is not available in the Standard and Professional editions of VisualAge for Java. In addition, it must be activated after the installation as described in 6.2, “Configuring VisualAge for Java V3.5 to support our examples” on page 204.

As an example, we describe how we created the Personalization EJB. To create an EJB, start VisualAge for Java and select the EJB tab. Select EJB -> Add -> EJB Group. Select or enter the name of your Redbook examples project and the group name RedbookBanking, as shown in Figure 229. Click Finish to create the group.

![Figure 229. Creating an EJB group in VisualAge for Java](image-url)
After you have created the group, right-click its name and select **Add -> EnterpriseBean**. Keep the default selection **Create a new enterprise bean** and enter **Personalization** as the bean name. The Personalization bean is a session bean (it does not have persistent data), so keep the default selection for the bean type. In the project field, select or enter the name of your redbook example project and as the package, enter `com.lotus.redbook.banking.ejb` as shown in Figure 230.

![Create Enterprise Bean dialog box](image)

**Figure 230. Create EJB dialog box in VisualAge for Java**

As soon as you click **Finish**, the EJB is created with all the methods a session bean is required to provide. Your VisualAge for Java screen now looks like Figure 231 on page 370; you have created a valid EJB and can start adding business logic.

The next section explains how VisualAge for Java can help you do this. The methods we added to the Personalization EJB are discussed in 9.3, “The Personalization EJB” on page 373.
9.2.2 Modifying and deploying EJBs in VisualAge for Java

This section shows some techniques that make EJB development in VisualAge for Java easier.

9.2.2.1 Adding new methods and fields to an EJB

To add a method to an existing EJB in VisualAge for Java, right-click the class name of the bean and select **Add -> Method**. Then enter the name and the parameters of the new method exactly as explained earlier in 6.3.2, “Creating the getAllServletParameters method” on page 224 for servlets. The dialog box for creating new EJB methods is displayed in Figure 232 on page 371.

You can also add fields by right-clicking an EJB name and selecting **Add -> Field**.
9.2.2.2 Adding methods to interfaces

All methods that will be called from outside the EJB must be added to the remote interface of it. You can do this in VisualAge for Java by right-clicking the method name and selecting Add To -> EJB Remote Interface. The method is created in the remote interface that belongs to the EJB and all changes are reflected automatically.

In the Personalization EJB the methods writePersonalization and getPersonalization must be called from outside the EJB and, thus, be added to the Personalization interface.

9.2.2.3 Adding and modifying the method control descriptor

For EJB methods the developer can specify some attributes that determine its behavior. In VisualAge for Java you can specify these attributes by right-clicking the method name and selecting EJB Method Attributes -> Add Control Descriptor. If the method already has a control descriptor, select EJB Method Attributes -> Edit Control Descriptor instead. In both cases the dialog box shown in Figure 233 on page 372 appears and you can create or modify the method control descriptor.
For the `getPersonalization` and the `writePersonalization` methods we describe in 9.3.2, “Methods of the PersonalizationBean class” on page 374 you select CLIENT_IDENTITY in the “Run-As Mode” field and leave the other fields on their default selection.

### 9.2.2.4 Deploying EJBs

The first time you deploy a new EJB, you should use the WebSphere Administrative Console to create the EJB in the WebSphere Administration database and set its security. WebSphere expects a deployable EJB for this.

To create a .jar file that contains a deployable EJB, highlight the EJB class and select **EJB -> Export -> EJB JAR**. Note that you must use the Export EJB JAR function so that the jar file includes the necessary descriptor file which specifies how the EJBs are deployed. The default directory for deployable EJBs is:

C:\WebSphere\AppServer\deployableEJBs

Once you have deployed the EJB, you do not have to repeat this after every code change. You can create the deployed code that is accepted by WebSphere in VisualAge for Java instead of this.

To create the deployed code, highlight the EJB class you want to export and select **EJB -> Generate Deployed Code**. After the deployed code is generated, select **EJB -> Export -> Deployed JAR**. The EJB container this EJB belongs to must be stopped before the code is replaced. The default directory for deployed EJBs is:

C:\WebSphere\AppServer\deployedEJBs
9.3 The Personalization EJB

The Personalization EJB retrieves and writes personalization documents to the Domino personalization database. It encapsulates the personalization data for a specific user. For an overview of the Personalization EJB functionality, see 5.4.2, “The Personalization EJB” on page 188.

The full source code for the Personalization EJB is in PersonalizationSource.jar that is part of 5955ejb.zip. See Appendix K, “Using the additional Web material” on page 521 for instructions on how to get the source code.

9.3.1 Handling the Domino session in EJBs

You can either create a new Domino session every time an EJB method is called or keep the Domino session open as long as the EJB exists. The latter is not possible if you need Domino security because the Domino session always will be associated with the user you specified when creating the session initially.

In our Personalization EJB we decided to keep the Domino session open to increase the performance of the application, but Domino does not keep inactive sessions infinitely. By default it drops IIOP sessions after one hour of inactivity. This time-out value can be changed in the Domino server document.

If Domino has dropped the session, the methods of the Personalization EJB remove their internal Domino session object and return an error. The next time they are called a new Domino IIOP session is opened. So we always try to call these methods for a second time if the first time in one servlet was not successful.

In our example, all users access the data using an administrative ID like this:

```java
m_sessCurrent = new NotesFactory().createSession(sHostName, "WASAdmin", "password");
```

The EJB code ensures that each user can only access their own personalization document.

9.3.1.1 EJB-specific createSession method

However, the first time a user connects to the system and the personalization document is created in the Domino database, we create a session specifically for this user (in order to get the hierarchical name of the user). We do that using an EJB-specific version of the createSession method introduced in
Domino R5.0.5, where access automatically is granted based on the current credentials object in the WebSphere environment:

```
Session sesUser = new NotesFactory().createSession (sDominoServer, null);
```

Using this method the developer does not need to care about passing the LTPA token in every user-specific call to an EJB that accesses Domino.

### 9.3.1.2 Considerations for keeping IIOP session open

When using our permanent IIOP session we retry method calls if they return an error. This is to recreate the IIOP session in case it has been dropped by the Domino server. This approach does not seem optimal, but currently there is no way to test whether an IIOP session is valid before using it. This ability may be added to a future version of the Domino Java API. In the meantime, you may want to run tests with a prototype of your application and see how big the overhead is if you create a new session for each user (and perhaps do it from a servlet instead of an EJB). If you choose to go with one always-open IIOP session, you can set the timeout higher than the default one hour specified in the Domino Directory. You can also read the specified timeout value from your program and do your own housekeeping with a timer so you renew the session before Domino drops it. The best approach will vary from application to application.

### 9.3.2 Methods of the PersonalizationBean class

This class has six methods:

1. `getEJBUserName()`

   This returns the user of the EJB derived from the EJB session context object.

   ```java
   private String getEJBUserName() {
     if (mySessionCtx == null) {
       return null;
     } else {
       String sEJBUser = mySessionCtx.getCallerIdentity().getName();
       int iPosSlash = sEJBUser.indexOf ('/');
       if (iPosSlash < sEJBUser.length()) {
         return sEJBUser.substring (iPosSlash + 1);
       } else {
         return null;
       }
     }
   }
   ```
2. getNextAccountNo(ApplicationInfo, Database ndbPersonal)

This method searches the Domino personalization database to find the current highest numbered account in use and returns the next account number. It also updates the database with the new highest numbered account number. Account numbers are allocated in pairs (checking and savings account numbers), separated by 10 to allow for issuing up to eight more accounts for the user. If the document with the highest account number does not exist, it is created (with account number 11). Following is the code for this method:

```java
private int getNextAccountNo(String sDominoServer, String sPersDbPath) {
    try {
        Database ndbPers = getPersDatabase(sDominoServer, sPersDbPath);
        if (ndbPers == null) {
            m_sesCurrent = null;
            return 0;
        } else {
            View vwAccountNo = ndbPers.getView("viref_foAccountID");
            Document docAccountNo = vwAccountNo.getFirstDocument();
            if (docAccountNo == null) {
                // If there isn't an account document, create it with Account #1
                docAccountNo = ndbPers.createDocument();
                docAccountNo.replaceItemValue("Form", "fo_AccountID");
                docAccountNo.replaceItemValue("Account", new Integer(11));
                docAccountNo.save();
                return 10;
            } else {
                int iCheckingAccount =
                    docAccountNo.getItemValueInteger("Account") + 9;
                docAccountNo.replaceItemValue("Account", new Integer(iCheckingAccount + 1));
                docAccountNo.save();
                return iCheckingAccount;
            }
        }
    } catch (NotesException e) {
        System.out.println("Personalization EJB: Notes Error ", e.id + ": " + e.text);
        m_sesCurrent = null;
        return 0;
    } catch (Exception e) {
        System.out.println("Personalization EJB: Error: " + e.getMessage());
        m_sesCurrent = null;
        return 0;
    }
}
```
3. getPersDatabase (String sHostName, String sDbPath)

This method opens the Domino personalization database. If there is no IIOP session established, it will create (and leave open) an IIOP session to the Domino server. The method returns a database object set to the (open) personalization database. An extract of the method is shown below:

```java
private Database getPersDatabase(String sHostName, String sDbPath) {
    Database ndbPersonalization = null;
    try {
        if (m_sesCurrent == null) {
            // create new notes session and access db
            m_sesCurrent = new NotesFactory().createSession(sHostName, "WASAdmin", "password");
        }
        if (m_sesCurrent != null) {
            // get personalization db
            ndbPersonalization = m_sesCurrent.getDatabase(m_sesCurrent.getServerName(), sDbPath, false);
            if (ndbPersonalization != null) {
                // open personalization db
                if (!ndbPersonalization.isOpen())
                    ndbPersonalization.open();
            }
        }
    }
    return ndbPersonalization;
}
```

4. getPersonalization(ApplicationInfo, String, boolean)

This method searches for a personalization document for the user; if one is not found and the bCreateDocuments variable is set to true, it is created with default values; the method then returns a hashtable of values from the personalization document just created. The hashtable contains a flag to indicate whether banking accounts should be created by the invoking method.
public Hashtable getPersonalization(String sDominoServer, 
String sPersDbPath, boolean bCreateDocuments)
throws java.rmi.RemoteException {
    Hashtable hPersonalization = new Hashtable();
    try {
        Database ndbPers = getPersDatabase (sDominoServer, sPersDbPath);
        String sShortName = getEJBUserName();
        if (ndbPers != null && sShortName != null) {
            View vwPeople = ndbPers.getView("viref_foPers$ByShortName");
            // get personalization document
            Document docPerson = vwPeople.getDocumentByKey(sShortName);
            if (docPerson == null) {
                // no document found -> create it
                String sDefaultQuery = "FIELD Form CONTAINS fo_Content";
                if (bCreateDocuments) {
                    int iChecking = writePersonalization(sDominoServer, 
                        sPersDbPath, sLtpaToken,
                        sDefaultQuery, 100, "RELEVANCE", "1");
                    if (iChecking == 0) return null;
                }
                // return the default values
                hPersonalization.put("checking",new Integer(iChecking));
                hPersonalization.put
                    ("savings", new Integer(iChecking + 1));
                hPersonalization.put("sstring", sDefaultQuery);
                hPersonalization.put("maxresults", new Integer(100));
                hPersonalization.put("sortoptions", "RELEVANCE");
                hPersonalization.put("displayoptions", "1");
                hPersonalization.put
                    ("docreateaccounts", new Boolean (true));
                return hPersonalization;
            }
        }
    }

Figure 234. The getPersonalization method: Retrieve personalization and write a new document

If the personalization document is found, it is read using the
getItemValueString and getItemValueInteger methods of the Domino
document class, as shown in Figure 235 on page 378.
5. **getSessionContext()**

   This method simply returns the EJB session context in a session context object.

6. **writePersonalization(ApplicationInfo, String sDominoServer, String sQuery, int iMaxResults, String sSortOptions, String sDisplayOptions)**

   This method updates personalization documents with the input parameters; if the personalization document does not exist, it is created with the input values. The method opens an IIOP session to the Domino server; if it has been invoked by the getPersonalization method, this will create a second session, but this will only happen the first time a user accesses their personalization document.
Figure 236. The writePersonalization method: Retrieving personalization document

If there is no document to retrieve, one is created, as shown in Figure 237 on page 380. Note that only the fields not input (such as the checking and savings account numbers) are created; the remaining fields are filled in when we drop out of the test for a null document into the logic to update an existing document, as shown in Figure 238 on page 381. In this case the second session is created using the rights of the current EJB user.

We create a new session for first-time users because the getEJBUserName method only returns the short name for the requesting user and we want to use the hierarchical name in the profile document. When we have a user-specific session, we can get the hierarchical name using the getUserName method on the Domino session. We could instead do a lookup in the directory to get the hierarchical name, but this would only work when using Domino Directory. If using, for example, IBM Secureway directory, we would need further manipulation in our code to produce a Domino hierarchical name, so we stick with creating an extra session this first time.
if (docPerson == null) {
    // The person does not yet have a Personalization Document in the database
    // We will create a default Personalization Document
    // (using the input parameters supplied)
    iChecking = getNextAccountNo (sDominoServer, sPersDbPath);
    if (iChecking == 0) return 0;
    docPerson = ndbPers.createDocument();
    Session sesUser = new NotesFactory().createSession
        (sDominoServer, null);
    docPerson.replaceItemValue ("tUserName",
        sesUser.getUserName());
    docPerson.replaceItemValue ("tShortName", sShortName);
    docPerson.replaceItemValue ("Form", "fo_Personalisation");
    docPerson.replaceItemValue ("CheckingAccountNo",
        new Integer(iChecking));
    docPerson.replaceItemValue ("SavingsAccountNo",
        new Integer(iChecking + 1));
}

Figure 237. The writePersonalization method: Creating a new document

Finally, we update the document with the input values as shown in Figure 238.
} else {
    iChecking = docPerson.getItemValueInteger
        ("CheckingAccountNo");
}

// update personalization document
    docPerson.replaceItemValue ("SearchString", sQuery);
    docPerson.replaceItemValue ("MaxResults",
        new Integer(iMaxResults));
    docPerson.replaceItemValue ("SortOptions", sSortOptions);
    docPerson.replaceItemValue ("DisplayOptions",
        sDisplayOptions);
    docPerson.save();
    vwPeople.refresh();
    return iChecking;
} else {
    System.out.println ("Personalization EJB: Could not open database " + sDominoServer + " ! " + sPersDbPath);
    if (sShortName != null) System.out.println
        ("Personalization EJB: EJB user is " + sShortName);
    if (m_sesCurrent != null) System.out.println ("Personalization EJB: Notes Session is open for " + m_sesCurrent.getUserName());
    return 0;
}
}
} catch (NotesException e) {
    System.out.println ("Personalization EJB: Notes Error #" + e.id + ": " + e.text);
    m_sesCurrent = null;
    return 0;
} catch (Exception e) {
    System.out.println ("Personalization EJB: Error: " + e.getMessage());
    m_sesCurrent = null;
    return 0;
}
9.4 Calling EJBs that are managed by WebSphere

This section explains how you can call EJBs that are distributed in WebSphere containers from any Java program or agent.

9.4.1 Prerequisites for calling WebSphere EJBs

In every Java program that works as a client for WebSphere EJBs, you use the Java RMI classes and some additional JDK and WebSphere classes to call the EJBs. The steps we describe in this section ensure that you can access them.

9.4.1.1 Include required Java archives into your classpath

An EJB client must have access to certain Java archives which are provided by WebSphere. Servlets that run in the WebSphere environment automatically have access to these archives via the WebSphere classpath.

If your Java client is not a servlet running in WebSphere, then make sure that these two files can be accessed in your development environment and in your runtime environment:

- ujc.jar
- ejs.jar

You find them in the AppServer\lib subdirectory of your WebSphere directory. They contain the Websphere EJB client environment. In VisualAge for Java you have access to these files when you add the IBM WebSphere Test Environment.

If your EJB client is not a WebSphere servlet, you need one additional archive per EJB you are calling. The WebSphere application server creates it when you deploy the EJB and places it in the subdirectory deployedEJBs of your WebSphere directory. WebSphere adds the prefix “Deployed” to the name of the original archive. That means after you deploy an EJB that is packaged in an archive named Account.jar, you will find a file named DeployedAccount.jar in the deployedEJBs directory.

In addition to all EJB classes you need at execution time, this file contains the class for the client stub. Therefore, you will not be able to look up the client stub as described in 9.4.2, “Getting the client stub via the naming service” on page 383 unless your EJB client has access to this file in the runtime environment. In the development environment you should use the original EJB classes instead of the classes in this file.
9.4.1.2 Importing required Java packages

Each WebSphere EJB client must import the following packages:

- **java.util** - This package contains some utility classes the WebSphere EJB client uses.
- **java.rmi** - This package contains classes for remote method invocation (RMI). You have to include this package because, typically, Java EJB clients use the RMI interface of EJBs.
- **javax.rmi** - This package contains the PortableRemoteObject class required to get a reference to an EJB object.
- **javax.ejb** - This package contains the classes and interfaces defined in the EJB specification.
- **javax.naming** - This package is used by the naming service to get a reference to the EJB objects.

In addition, you need to import the packages that contain the EJB classes your client is interacting with. If you are calling the account example that comes with WebSphere, the import section of your client class looks like:

```java
// Classes necessary for EJB access
import java.util.*;
import java.rmi.*;
import javax.rmi.*;
import javax.ejb.*;
import javax.naming.*;
// EJB classes that are used in this program
import com.ibm.ejs.doc.account.*;
```

9.4.2 Getting the client stub via the naming service

Each EJB has a home interface that works as a stub of the EJB in the client. WebSphere provides a name service that finds the bean and returns its home object. This service uses IIOP for the client-server communication.

This example code creates the initial context necessary to access the name service:

```java
String sProviderURL = "iiop://YOUR_HOST_NAME/:900";
String sNameService = "com.ibm.ejs.ns.jndi.CNInitialContextFactory";
Hashtable htEnv = new Hashtable();
htEnv.put(javax.naming.Context.PROVIDER_URL, sProviderURL);
htEnv.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY, sNameService);
InitialContext ctx = new InitialContext(htEnv);
```
The steps covered in the sample code are:

1. Create a new Java Hashtable.

2. Set the property `javax.naming.Context.PROVIDER_URL` in the hashtable. It contains the URL of the name service. The URL consists of the string "iiop://" to indicate an IIOP connection, the host name of your WebSphere server, and the port. The IIOP port the server is listening to usually is 900.

3. Set the property `javax.naming.Context.INITIAL_CONTEXT_FACTORY` in the hashtable to the name of the class that implements the naming service. This class depends on the WebSphere server you are using. For the advanced edition you use `com.ibm.ejs.ns.jndi.CNInitialContextFactory`. For the enterprise edition you need `com.ibm.ejb.cb.runtime.CBCtxFactory`.

4. Create a new object of the class `javax.naming.InitialContext` and pass the hashtable as a parameter to the constructor of this class.

After you have created the initial context, use it to look up the EJB. You use the name that was specified as the JNDI Home name of the EJB. You can find it in the WebSphere administrative console, as shown in Figure 239.

![WebSphere Administrative Console](image-url)

**Figure 239. EJB name in the WebSphere administrative console**

384  Domino and WebSphere Together
To change the JNDI Home Name, click **Edit** beside the Deployment Descriptor field. You can edit the name here, as shown in Figure 240 on page 385. You must make sure that this name is unique for one WebSphere server.

![Deployment Properties](image)

**Figure 240. Changing the JNDI home name of an EJB**

You should check the class of the object the name service returns using the `narrow` method of the `javax.rmi.PortableRemoteObject` class. This class throws a `ClassCastException` if the object that is returned by the name lookup is not an object of the intended class. A possible reason for this exception can be that the bean-specific archive is not accessible in the runtime environment. If so, you have to modify the setup as described in 9.4.1.1, “Include required Java archives into your classpath” on page 382.

The code for the EJB lookup and the class check looks like this:

```java
Object objHome = ctx.lookup("Account");
ejbAccount = (AccountHome)PortableRemoteObject.narrow
    (objHome, AccountHome.class);
```

### 9.4.3 Creating an EJB and calling the methods it provides

After you have located the home object of an EJB, you can use it to create a new EJB or find an existing EJB. A create method is invoked to create an EJB object or a finder method is invoked to find an existing EJB object. The finder method is only valid for entity beans because only one instance of an entity bean exists for any given primary key in the system.
The code to create a bean looks like:

```java
// Create the EJB.
Account ejbAccount = ejbhAccount.create(objKey, iTypeAcct, fBalance);
```

After the bean is created or found, you can use all methods its remote interface contains. For session beans you should handle the `java.rmi.NoSuchObjectException`. This exception is thrown when the session bean does not exist any longer. This can happen when the WebSphere server is stopped and restarted.

### 9.5 Calling EJBs from Domino agents

If you want to call a WebSphere EJB from a Domino agent, you have to do a bit more work than you would if calling it from a WebSphere servlet. We discuss three different ways to call an EJB from a Domino agent:

- Calling an EJB directly from an agent
- Calling an EJB from an agent via an RMI server
- Calling an EJB from an agent via a servlet

#### 9.5.0.1 Calling an EJB from a Domino R5 agent directly

Theoretically, you can call an EJB from any Java program. Since Domino applications can contain Java agents, one could assume that it is possible to include calls to an EJB into a Domino R5 agent.

In fact, in the last edition of this book, in which we used WebSphere 3.0.2 and Domino 5.0.4, we showed an example of an agent that calls an EJB directly. But the Java Virtual Machine of WebSphere 3.5.3 is on 1.2.2 level, whereas Domino only supports version 1.1.8.

This means that the WebSphere name service classes cannot be loaded by the Domino servlet engine and it is not possible to access a WebSphere 3.5 EJB directly from Domino R5.

In addition, this technique has other serious drawbacks, including:

- The EJB classes expect to be able to access the class code on the local machine, so you must copy them to the hard disk of the Domino server instead of deploying them inside the Domino agent, which is the normal way of deploying the classes of a Domino R5 Java agent.
- Domino agents try to stop all threads they created when they stop. Since Domino cannot stop a thread that initialized an EJB access previously, it displays an error message. You can prevent this error message by starting
a thread in the system thread group, but this possibility may be disabled in future versions of Domino because of security concerns.

- For a Domino R5 agent there is no possibility to create user credentials that are accepted by an EJB. This means that you can only call an EJB directly if the WebSphere global security is switched off. This is not acceptable for most production environments because it enables all Java clients to use all EJBs in the WebSphere Administrative Domain.

These problems imply that you should not call an EJB from an agent directly. Even if future versions of Domino support JDK 1.2.x, we do not recommend calling EJBs directly until Domino agents are redesigned to support EJB calls.

9.5.0.2 Calling an EJB from a Domino agent via an RMI server

EJBs can be called by a Java application. If this application also is an RMI server, it can work as a gateway that passes requests from a Domino R5 Java agent to an EJB that is managed by WebSphere. The architecture of this solution is displayed in Figure 241.

This architecture separates the EJB access from Domino. In Domino, only the standard RMI communication is used. Thus, the Domino agent manager has no problem stopping all threads after the agent execution. It also supports WebSphere security. The RMI server can connect to WebSphere as a specified user.

The disadvantage of this solution is the need for running the RMI server as a separate application. Our RMI server is written in a way that it expects to be executed on the WebSphere server, but there is no special need to design an
application in this way. If properly written, a RMI server can run anywhere on
the network.

9.5.0.3 Calling an EJB from a Domino agent via a servlet
Using the Java URL classes, you can use a servlet from a Domino R5 agent
by calling its URL, as shown in Figure 242. This architecture has the
advantage that you only use the HTTP protocol. This avoids possible
problems with blocked port numbers, proxy servers, or fire walls. In addition,
you need not provide access for the WebSphere archives since EJB access
is performed inside of WebSphere.

If you want to pass data to the servlet, you must use the parameters of the
servlet URL. To pass data from the servlet or from the EJB via the servlet to
the Domino R5 agent, you can parse the page the servlet returns. But these
techniques are more complicated than the mechanism of using an RMI
server.

We did not include this architecture in our sample application.

We now look at how to access the EJBs we use in our sample application
from servlets and from a Domino agent.
9.6 Using Enterprise Beans in the sample application

In this section we discuss how to access the EJBs in our sample application, including:

- Accessing the Account EJB and the Personalization EJB from servlets

  The IBM banking example Account bean and our Personalization bean are used in the application from servlets that run in WebSphere, so we access the EJBs in the way that was intended for WebSphere applications.

- Accessing the Transfer EJB from a Domino agent

  We then call the Transfer EJB from a Domino Java agent. This is possible because EJBs also can be called by Java applications and the WebSphere EJB container treats a Domino agent like an independent Java application.

9.6.1 Using the Account and Personalization EJBs from servlets

We are accessing the Account and Personalization EJBs from the PanelBuilder, SaveQuery and CallDominoJsp servlets. Descriptions of these servlets are in 7.4.2, “PanelBuilder servlet” on page 257, 7.4.4, “SaveQuery servlet” on page 263 and 8.6.3, “The CallDominoJsp controller servlet” on page 344.

When using EJBs from servlets that run in WebSphere all necessary libraries can be accessed automatically, so you do not have to include anything into the class path as described in Chapter 9.4.1.1, “Include required Java archives into your classpath” on page 382. However, there must be import statements for the packages that are used as we noted in Chapter 9.4.1.2, “Importing required Java packages” on page 383.

Since the task of assigning the home object is common for all servlets that use EJBs, we created the method `getHomeObject` in the class BankingServletTemplate, which is the superclass of all our servlets. The EJB packages are only necessary for this lookup, so only this class must contain the common EJB import statements.

A description of the `getHomeObject` method in 7.3.1.2, “Methods for the BankingServletTemplate class” on page 242. It always returns the home object of the EJB we want to access.

We use this home object to create the EJB and then call other methods of the EJB if this is necessary. This is done in the `doGet` and `doPost` methods of the servlet classes.
9.6.1.1 Security considerations
If global security is switched on in IBM WebSphere 3.5, no unprotected EJB can be called anymore.

In addition, most Web applications are configured to pass the current servlet user to all EJBs they call. Servlets only can pass this user if they are protected themselves.

It is also not possible to use a servlet session object like the ApplicationInfo object we describe in 7.3.2, “ApplicationInfo class” on page 246 in an unprotected servlet if it was created by a protected servlet.

So as soon as you have switched on global WebSphere security, you will not be able to test the example application unless you create the security information for it. We explain in 10.5.1, “Installing the RMI server” on page 446 how to do this.

9.6.2 Calling the Transfer EJB from a Domino agent
In most scenarios that contain user interactions and EJB access, servlets are preferable over Domino agents. Not only is it much easier to initiate the EJB access from a servlet, also the performance will be much better for servlets, because they remain in memory and can keep data between their calls.

Even if you are posting data from a Domino form, you can use a servlet instead of a QuerySave agent. We show this in the SaveQuery servlet that we discuss in 7.4.4, “SaveQuery servlet” on page 263.

However, calling an EJB from a Domino agent can be the best solution if the agent is a scheduled background agent. In this case the slower performance when the agent connects to the EJB is acceptable and you can take advantage of the scheduling capabilities the Domino R5 agent manager provides.

Our example agent loops through all personalization documents. It transfers a fixed amount from the checking account to the savings account of a user.

Note: We did not include a Web user interface to edit the amount that will be saved, so you will have to use the Notes client to enter amounts for some users.

9.6.2.1 The AgentTransfer class
The AgentTransfer class performs the operation of the agent. It must be defined as the base class of the agent, as we describe in 9.6.2.5, “Creating the Transfer Funds Domino agent” on page 398. Agent base classes extend
lotus.domino.AgentBase. When the agent is started, the agent manager calls NotesMain().

AgentTransfer only contains this method. It connects to the RMI server and then loop trough all personalization documents. For each document is moves the specified amount from the checking to savings account and stores the messages that the EJB returns in the documents.
public void NotesMain() {
    try {
        // Get the Domino context
        Session sesCurrent = getSession();
        Database ndbCurrent =
            sesCurrent.getAgentContext().getCurrentDatabase();
        View vwPers = ndbCurrent.getView("viref_foPers$ByName");
        String sUser = "YOUR ADMIN USER";
        String sPassword = "YOUR ADMIN PASSWORD";

        // Get link to RMI server
        AgentTransferInterface rmiTransfer = (AgentTransferInterface)
            java.rmi.Naming.lookup("//YOUR_HOST_NAME/TransferServer");

        // process the documents
        Document docPers = vwPers.getFirstDocument();
        Hashtable hTransfer = null;
        Double dAmount = null;
        while (docPers != null) {
            dAmount = new Double (docPers.getItemValueDouble("nMonthlySavings"));
            if (dAmount != null) {
                hTransfer = rmiTransfer.transferFunds(sUser, sPassword,
                    docPers.getItemValueString("CheckingAccountNo"),
                    docPers.getItemValueString("SavingsAccountNo"),
                    dAmount.floatValue());
                docPers.replaceItemValue("tEjbMessage", hTransfer.get("Message"));
                docPers.save(true, true, true);
            }
            docPers = vwPers.getNextDocument (docPers);
        }
        sesCurrent.recycle();
    } catch (NotesException e) {
        System.out.println ("Notes Error #" + e.id + " " + e.text);
    } catch (Exception e) {
        System.out.println ("Error: " + e.getMessage());
    }
}

Figure 243. The NotesMain method of the AgentTransfer class

To keep the code simple, we hard-coded username and password of the person we used to connect to Domino. In a real-world application you would
keep these names in a setup document. However, it is important that they
have to be passed with every method call to prevent unauthorized users from
using the RMI server to connecting to Domino.

9.6.2.2 The AgentTransferServer class
The AgentTransferServer class is a Java application that transfers the
requests from the Domino agent to the Transfer EJB. It is designed to run on
the WebSphere server.

It contains the variables on class level shown in Table 16.

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_ejbTransfer</td>
<td>Home object of the transfer EJB (is reused for performance reasons)</td>
</tr>
<tr>
<td>m_sServerName</td>
<td>WebSphere server name</td>
</tr>
<tr>
<td>m_sUserName</td>
<td>User name to compare if the current request comes with the same credentials as the last one</td>
</tr>
<tr>
<td>m_sPassword</td>
<td>Password to compare if the current request comes with the same credentials as the last one</td>
</tr>
</tbody>
</table>

The main method creates an object of the own class and then publishes the
availability of the RMI server to the RMI registry.

```java
public static void main(String args[]) {
    try {
        // create and bind RMI-EJB proxy object
        AgentTransferServer objServer = new AgentTransferServer(args[0]);
        Naming.rebind("rmi://" + args[0] + "/TransferServer", objServer);
        System.out.println("Transfer agent RMI server bound");
    } catch (Exception e) {
        System.out.println("bind RMI server failed!");
        e.printStackTrace();
    } finally {
        Runtime.getRuntime().gc();
    }
}
```

Figure 244. The main method of the AgentTransferServer class

The constructor saves the host name to use for the connection to WebSphere
later. This usage of the host name for the connection to WebSphere is the
only reason the RMI server and the WebSphere server must run on the same host.

```java
public AgentTransferServer(String sServer) throws RemoteException {
    m_sServerName = sServer;
}
```

*Figure 245. The constructor of the AgentTransferServer class*

The `transferFunds` method is called for every document. It first calls the private method `getTransferHomeObject` to connect to the Transfer EJB. It then calls the EJB method `transferFunds` and returns a hashtable containing the new balances and the message that describes errors or successful transfer.
public Hashtable transferFunds(String sUser, String sPassword, String sFromAccount, String sToAccount, float fAmount) throws RemoteException
{
    Hashtable hReturn = new Hashtable();
    TransferHome ejbhTransfer = null;
    ejbhTransfer = getTransferHomeObject (sUser, sPassword);
    if (ejbhTransfer == null) {
        // could not create home object
        hReturn.put ("Success", new Boolean (false));
        hReturn.put ("Message", "Could not create home object");
        return hReturn;
    } else {
        try {
            long lFromKey = Long.parseLong(sFromAccount);
            long lToKey = Long.parseLong(sToAccount);
            // create session bean and invoke transfer method
            ejbTransfer = Transfer ejbTransfer = ejbhTransfer.create();
            ejbTransfer.transferFunds(lFromKey,lToKey,fAmount);
            hReturn.put ("BalanceFrom", new Float (ejbTransfer.getBalance(lFromKey)));
            hReturn.put ("BalanceTo", new Float (ejbTransfer.getBalance(lToKey)));
            hReturn.put ("Success", new Boolean (true));
            hReturn.put ("Message", "Transfer successful");
            return hReturn;
        } catch (Exception e) {
        ...
    }
}

Figure 246. The transferFunds method of the AgentTransferServer class

The getTransferHomeObject method first tests if the home object already exists and if the user that initialized the request is the same user as during the last request. If yes, it simply returns the home object that already exists.

Otherwise, it creates the home objects in three steps. These steps must be performed in this order:

1. Create the initial context that contains the provider URL and the context factory class.

2. Call a helper method to perform a login. This is only necessary if you connect from a client outside of WebSphere to a protected EJB.
3. Get the home object using the `lookup` and `narrow` methods as described in 9.4.2, “Getting the client stub via the naming service” on page 383.

The home object then is stored in a class level variable and returned.

```java
private TransferHome getTransferHomeObject(String sUser, String sPassword) {
    try {
        if (m_ejbhTransfer == null || sUser.compareTo(m_sUserName) != 0 || sPassword.compareTo(m_sPassword) != 0) {
            Hashtable hEnv = new Hashtable();
            hEnv.put(javax.naming.Context.PROVIDER_URL, "iiop://" + m_sServerName + ":900");
            hEnv.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                     "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
            // Create the initial context. and get the home object
            Context ctx = new InitialContext(hEnv);
            Object objHome = null;
            if (ctx != null) {
                if (performLogin (sUser, sPassword)) objHome =
                    ctx.lookup("Transfer");
            }
            if (objHome != null) m_ejbhTransfer = (TransferHome) narrow(objHome, TransferHome.class);
        }
        return m_ejbhTransfer;
    } catch (Exception e) {
        return null;
    }
}
```

*Figure 247. The `getTransferHomeObject` method of the `AgentTransferServer` class*

The `performLogin` method connects to the WebSphere server using the `LoginHelper` class that comes with WebSphere. You find this class on your WebSphere server in the directory:

C:`\WebSphere\AppServer\hosts\default_host\examples\security`

The easiest way to use it is to import it into the package you are using for your project.
private boolean performLogin(String sUserId, String sPassword) {
    if (sUserId.compareTo(m_sUserName) == 0
        && sPassword.compareTo(m_sPassword) == 0) {
        return true;
    } else {
        try {
            m_sUserName = sUserId;
            m_sPassword = sPassword;
            LoginHelper objLogin = new LoginHelper();
            org.omg.SecurityLevel2.Credentials objCredentials =
                objLogin.login(sUserId, sPassword);
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}

public interface AgentTransferInterface extends Remote {
    Hashtable transferFunds(String sUser, String sPassword,
        String sFromAccount, String sToAccount, float fAmount)
    throws RemoteException;
}

Figure 248. The performLogin method of the AgentTransferServer class

9.6.2.3 The AgentTransferInterface remote interface
Each RMI server needs a remote interface that specifies all methods that can
be called remotely. In our AgentTransferServer this is only the transferFunds
method. So our AgentTransferInterface interface looks like Figure 249.

Figure 249. The AgentTransferInterface

The AgentTransferServer class must implement this interface and extend the
RMI UnicastRemoteObject. You modify the definition of the
AgentTransferServer class as displayed in Figure 250 on page 398.
9.6.2.4 The RMI stub and skeleton classes

Each RMI server also needs RMI stubs and skeletons. Since Domino only supports JDK 1.1.8, we created these versions of the stubs and skeletons.

These classes can be generated by the Development Environment or by the Sun JDK utility rmic. To create the stubs and skeleton in VisualAge for Java, highlight the AgentTransferServer class and select Selected -> Tools -> Generate RMI -> JDK 1.1 stubs/skeletons. VisualAge for Java generates the classes.

9.6.2.5 Creating the Transfer Funds Domino agent

You create the Transfer Funds agent in the Personalization Domino database because it processes all the database documents.

Before you can create the agent, export the Java code from you IDE into a directory. In VisualAge for Java, highlight the com.lotus.redbook.banking package and select File -> Export. Then select Directory and click Next. Here you enter a directory on your hard disk and click Finish.

Open the Personalization database in the Domino designer and create a new agent. Name it Transfer Funds, and for test reasons set the schedule to every hour. The field Which document(s) should it act on? can remain at its default value.

In the code frame, select Imported Java in the Run field and click Import class files. Specify your Java Home directory as the base directory for this agent. You have done this correctly if you can expand com\lotus\redbook\banking directories in the listbox on the left side, as displayed in Figure 251 on page 399.

```java
public class AgentTransferServer
    extends java.rmi.server.UnicastRemoteObject
    implements AgentTransferInterface
{
...
}
```

Figure 250. Definition of the AgentTransferServer class
Select the three classes AgentTransfer, AgentTransferInterface and AgentTransferInterface_Stub and click **Add/Replace File(s)**. Select **AgentTransfer** as the base class of the agent and click **OK**. Your agent now should look like Figure 252 on page 400.
Figure 252. The Transfer Funds agent

Save the agent and deactivate it until you have distributed the application because it cannot work if the RMI server is not running.

In 10.5.1, “Installing the RMI server” on page 446 we discuss how to set up the RMI server.

9.7 Summary

In this chapter we have briefly discussed when it makes sense to access Domino from an EJB. We have shown how to use VisualAge for Java for development and deployment of EJBs and we have looked at the code in the Personalization EJB. We have also explained how to access EJBs from servlets or Domino agents.
Chapter 10. Installation of the banking application

This chapter describes the actual installation of our sample application.

We first take you through placement of the actual physical files used, and then describe the necessary Domino configuration followed by the setup of the WebSphere part of the application.

In our environment, we created the applications on workstations with VisualAge for Java installed. We chose to install the application on two computers, one running Domino (odin.lotus.com) and the other (Thor-2000.lotus.com) running WebSphere. The Domino server was running the HTTP task and connected to WebSphere using OSE remote.

We used the configuration described in Chapter 3, “Installation and setup” on page 39; SSO should be configured and tested as described in 4.4, “Setup of the Domino-WebSphere single sign-on environment” on page 122. If you have problems with the setup, you should resolve these before proceeding (4.7, “Troubleshooting single sign-on” on page 175 may help you identify common problems).

We exported the application servlets and helper classes to a jar file, the EJBs to three jar files, and finally, imported the JSPs as source text files (since they are compiled to servlets as needed).

All the sample code we install in this chapter is available as additional Web material from the IBM Redbooks Web site, or as part of the examples that are installed together with WebSphere. See Appendix K, “Using the additional Web material” on page 521 for instructions on how to get the sample files we developed.

The components you will need to install this example are in a file named 5955deploy.zip. This file contains four other zipped files. Their content is described in Table 17.

### Table 17. Zip files with install code for sample application

<table>
<thead>
<tr>
<th>Zip file</th>
<th>Content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5955run-domino.zip</td>
<td>intra_01.nsf</td>
<td>The Domino databases that makes up the content management and publishing part of the application. Unpack the zip file in the Domino data directory and keep the path stored in it.</td>
</tr>
</tbody>
</table>
<pre><code>                                             |
</code></pre>
<p>|                      | intra_02.nsf     |                                                                             |
|                      | intra_03.nsf     |                                                                             |</p>
We installed WebSphere with the default application environment; if your environment is different you will need to modify these instructions appropriately. The portions of the default environment we assume are present are:

- The virtual host **default_host**
- The **Default Server** application server together with the **Default Servlet Engin**.

The following is a high level overview of the installation steps.

1. Place the application files in their execution environment.
   a. Place the Domino databases in an application directory added to the Domino server’s data directory.
   b. Place the application jar files, Java Server Pages files, and EJBs in the WebSphere file system.

2. Configure the WebSphere environment.
   a. Create an enterprise application containing:
      - A Web application containing the four servlets in the application.
      - An EJB Container (containing a data source for the Bank accounts, and the three EJBS used by the application).
   b. Configure resource security for the EJB’s and Web resources.
   c. Configure permissions for the enterprise application.

3. Configure the Domino environment:

<table>
<thead>
<tr>
<th>Zip file</th>
<th>Content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5955run-ejb.zip</td>
<td>Personalization.jar</td>
<td>The deployable Personalization EJB. Unpack this file to \WebSphere\AppServer\deployableEJBs on the WebSphere server.</td>
</tr>
</tbody>
</table>
| 5955run-webapp.zip| Banking.jar redbook-taglib.tld
error.jsp
topframe.jsp
viewnav.jsp      | Files for the Web application part of the WebSphere application. Banking.jar contains all servlets. Unpack the zip to \WebSphere\AppServer\hosts\default_host and keep the path stored in it. |
| 5955run-rmi.zip   | sas.client.props transfer.bat | Files used to have Domino agent call EJB via RMI server. Are used in conjunction with Domino agent in intra_02.nsf. Unpack in same directory as Banking.jar. |
a. Place the three Domino databases in a directory under the Domino data directory
b. Make sure the user referenced in the Personalization EJB code exists.
c. Add group for editors to Domino Directory
d. Populate the databases with sample content documents and links
e. Configure the application setup information so that it properly points to the WebSphere environment.

4. Test the installed environment.

10.1 Installation of the application components

Before starting installation, we need to decide on a naming structure so that we know which physical locations (that is, directories) need to be created in the file systems and what they should be called. In addition, we need to decide on the Web path that will be used to locate the components, although the design of our application is such that users will not have to type in the URLs to the servlets and JSPs. We chose the term Redbook Banking as a common name to be used where possible for most components:

1. On the Domino computer: BankingExample (no spaces) is the application directory to hold our application databases.
2. On the WebSphere computer: we created a new directory tree to hold the WebSphere components (other than the EJBs).
3. We created a Web application whose Web application Web path is RedbookBanking (no space). This will be part of the URL to access the application.

Note that the directory on the Domino server (BankingExample) and the Web application Web path (RedbookBanking) can not be the same, since the Domino Web server plugin for WebSphere will direct all requests with a path starting with /RedbookBanking to the WebSphere server.

Do not change the file names of the Domino databases, since these are hard coded in parts of our sample application.

10.1.1 Placement of the Domino application components

On the Domino server we put the Domino part of our application in a directory under the Data directory. We called the directory BankingExample (without a space, so that servlets can create URLs to point to the databases).
• If you unpack the 5955run-domino.zip file (that is part of 5955deploy.zip) in the Domino Data directory and keep the path information for the zipped files, this directory will be created automatically.

• If you already have unpacked the databases (intra_01.nsf, intra_02.nsf and intra_03.nsf) in another location, create the BankingExample directory under the Domino Data directory and move the databases to this directory.

In an environment where Domino is installed using default options, the full path to our Domino databases should be:
C:\Lotus\Domino\Data\BankingExample

Again, do not change the file names of the Domino databases.

This completes the placement of code on the Domino server.

10.1.2 Placement of the WebSphere application components
We placed code for our Web application (servlets and JSPs) and our EJB on the WebSphere server.

10.1.2.1 Placing the servlets and JSP files
For our Web application files we chose to name the base directory Redbook Banking. This directory should be created in the \WebSphere\AppServer\hosts\default_host directory since we will use the virtual host default_host created by WebSphere when it was installed.

• If you unpack the 5955run-webapp.zip file (that is part of 5955deploy.zip) in the \WebSphere\AppServer\hosts\default_host directory and keep the path information for the zipped files, the Redbook Banking directory and those below will be created automatically.

Following are the steps for manually placing the Web application files. If you have unpacked the 5955run-webapp.zip file as described above, you can skip the rest of this section and continue with 10.1.2.2, “Placement of the Enterprise JavaBeans” on page 406.

To place the Web application files manually, do the following:
1. Using the Windows explorer program (or using a command prompt, if you prefer), create the directory Redbook Banking under the \WebSphere\AppServer\hosts\default_host directory.

2. Create two directories under the Redbook Banking directory:
- **servlets** - this will be the path to locate class files for servlets and associated classes
- **web** - this will be the path to locate text files, such as HTML (there will be none in our example) and JSPs.

The completed directory structure will be similar to Figure 253.

![Figure 253. Directory structure for Redbook Banking files](image)

3. Place the Banking.jar file in the Redbook Banking\servlets directory.
4. Place the JSPs and the taglib file in the Redbook Banking\web directory:
   - error.jsp
   - redbook-taglib.tld
   - topframe.jsp
   - viewnav.jsp

   Again, if you use our copy from 5955run-webapp.zip, the files should already be in their right place.

10.1.2.2 Placement of the Enterprise JavaBeans
Now navigate to the deployableEJBs directory under \WebSphere\Appserver and place the EJBs for this application:
   • Account.jar - This should already be present, provided you installed the Samples when you installed WebSphere.
   • Transfer.jar - This should already be present as part of the WebSphere Samples.
   • Personalization.jar - This is the personalization EJB used to access Domino. You can get this from the 5955run-ejb.zip file.

   This completes the placement of code on the WebSphere server.

10.2 Configuration of the Domino application components
All of the application code is now in place, but we need to configure it appropriately for our application to work. We will configure Domino first, and then WebSphere.

The Domino part of the application configuration includes:
   • Making sure the user specified in the EJB code exists
   • Adding a user group to Domino Directory or modifying the ACLs of the application databases
10.2.1 Make sure the EJB Domino user exists

In our Personalization EJB code, we hard coded the name and password to use when creating the Domino session.

- If you use our sample EJB code, make sure that a user exists in the Domino Directory with the following information:
  - Short name: WASAdmin
  - Password: password

This user also needs editor access to the Domino Personalization database, as we discuss in the next section.

10.2.2 Verify the database access control lists

Our sample application is dependent on correct settings in the access control lists (ACLs) of the Domino databases.

The publishing database should not allow anonymous user access because we need to authenticate the user so we can get their personalization information. At the same time, those users should have read access to the database. You should be able to use the existing ACL settings for intra_01.nsf.

The personalization database must give editor access to the user ID we use for the persistent EJB connection. We also need to give all users of the application at least depositor access because they connect to the database when their initial personalization document is created. You will have to update the ACL of intra_02.nsf under all circumstances.

The authoring database must provide editor access to the content providers and approvers. Furthermore, the content manager roles must be associated on division level. If you only want to test the existing application and not add any new content, you do not need to change the ACL for intra_03.nsf.

Table 18 on page 408 contains an overview of the ACL settings we used in our sample application.
Table 18. ACL setting for the Domino databases

<table>
<thead>
<tr>
<th>Database</th>
<th>User/Group</th>
<th>Access</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>intra_01.nsf</td>
<td>-Default- Reader</td>
<td>Reader</td>
<td>All authenticated users are allowed to read.</td>
</tr>
<tr>
<td></td>
<td>Anonymous No Access</td>
<td>No Access</td>
<td>Users must not access the application without authenticating.</td>
</tr>
<tr>
<td></td>
<td>RedContentManagers Editor</td>
<td>Editor</td>
<td>Group for users/servers that copies content from authoring database to publishing.</td>
</tr>
<tr>
<td>intro_02.nsf</td>
<td>-Default- Depositor</td>
<td>Depositor</td>
<td>Users need access for initial creation of personalization document</td>
</tr>
<tr>
<td></td>
<td>Anonymous No Access</td>
<td>No Access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WebSphere Administrator/DOMWAS Editor</td>
<td>Editor</td>
<td>This is the user with short name WASAdmin that we use for the persistent EJB connection</td>
</tr>
<tr>
<td>intra_03.nsf</td>
<td>-Default- No Access</td>
<td>No Access</td>
<td>This database is only for content creators and approvers</td>
</tr>
<tr>
<td></td>
<td>RedContentManagers Editor</td>
<td>Editor</td>
<td>We use the same groups for all content creation and approval. Thus all roles in the database are associated with this group</td>
</tr>
<tr>
<td>All database</td>
<td>Administrators Manager</td>
<td>Manager</td>
<td>The Administrators group has manager access to all databases</td>
</tr>
</tbody>
</table>

These settings give adequate security for the Domino part of our application. To learn more about general security considerations for Domino Web
application see Lotus technote 176360, *A Guide to Secure Domino Applications*. You can see it by searching for 176360 on this Web site:

http://support.lotus.com

10.2.3 Updating the setup document

The setup document in the Domino database (intra_01.nsf, administration and content) must be set with parameters to match your WebSphere environment. To set this:

1. Open the intra_01.nsf database from a Notes client with access level of at least editor in the database.

2. Open to the `viref910Setup` view, as shown in Figure 254, and open the only document in the view.

![Figure 254. Selecting the Banking Example setup document in the intra_01.nsf database](image)

3. Open the document and place it in edit mode. Change the fields to match your environment:

   - **Servlet Path** - The Web application path that you will specify in URLs to locate the servlets. You can keep the default value of RedbookBanking.

   - **EJB Server** - The DNS host name of your WebSphere server. If you extend our configuration to run EJBs on another WebSphere server, specify that host name. For our example, this was thor-2000.lotus.com. This is required so that servlets can locate the name service to find the home method of the EJBs.
- **Login Servlet** - For our example, leave (or set) this to the Login servlet.

- **Debug Flag** - For installation verification, leave this set to Off. The other two settings enable debug information to be output from the Login servlet or from all servlets, respectively.

If you turn debug on, the output will go to:

```
WebSphere\AppServer\logs\<Virtual Host>_stdout.log
```

(in our case C:\WebSphere\AppServer\logs\default_server_stdout.log)

When all input is correct, save the document or press F9 to refresh it. The HTML head field is built from your input. The document, when filled in properly for our example, should be similar to the one shown in Figure 255.

![Image](image.png)

**Figure 255. The setup document for the Banking Application**

Save and close the document.

4. Select the document from the view (it will be the only one so it should already be selected) and copy a doclink to it to the clipboard (right-click the document and select **Copy as Link**) as shown in Figure 256 on page 411.
Figure 256. Creating a doclink to the setup document

Paste this doclink into the About document, so users will be pointed to the setup document when opening the database (and thus get the HTML head returned).

5. Right-click the database tab and select **Open in designer**. Alternatively, open the Domino Designer client and then open intra_01.nsf. In either case, locate the intra_01.nsf database in the navigator.

   - Select **Resources** and expand it.
   - Select **Other**.
   - Finally, open the **About Database** document.
   - If there already is a doclink in this document, delete it.
   - Paste the doclink you created earlier to the setup document and save it.

This procedure is necessary so that when the database is opened, it will open the Setup document first. This behavior is set by the database property: **When opened in a browser, launch first doclink in About document**.

If you want to use a WebSphere Web application that is different from /RedbookBanking, which is the one we refer to in this chapter, you need to go through one more Domino setup step that we describe in the next section.
Otherwise, you are done with Domino for now and can go to 10.3, “WebSphere configuration of the sample application” on page 413.

10.2.4 Setting up Web application path for main frame set

If your Web application path is different from /RedbookBanking, you need to change definitions in the main frameset document in intra_01.nsf to match your path. This is because the frameset refers to WebSphere servlets for frame content and the Web application path is part of the servlet references.

To change the Web application path on the Domino side, do the following in addition to the setup we already have covered:

1. Open the intra_01.nsf database and access the Administration—All documents view.

2. Select the document categorized under fo_FramesetMain, as shown in Figure 257.

![Figure 257. Locating the main frameset document](image)

3. Open the main frameset document and edit the paths for the frames, as shown in Figure 258 on page 413.
Chapter 10. Installation of the banking application

10.3 WebSphere configuration of the sample application

This section presents a high-level overview of the steps necessary to configure the sample application and then describes each step on detail.

The steps we need to perform are:

1. Create a data source for the Account EJB.
2. Create an EJB container in WebSphere.
3. Deploy the Account, Transfer, and Personalization EJBs to the container.
4. Create a new Web application for our servlets and JSPs.
5. Define an enterprise application for our EJBs and Web application.
6. Configure resource security and security permissions for the enterprise application.

The frameset definition at the bottom of the document is computed from the fields you enter. Ensure that you do not alter the remainders of the URLs for the framesets (unless these will match changed servlet names).

This completes the setup of Domino for the application.
10.3.1 Installing the EJBs

We start by installing the EJBs. In the following sections we take you through creating a database and defining a data source for our entity EJB, creating an EJB container for the EJBs, and deploying the EJBs to that container.

10.3.2 Create a database for the Account EJB

The Account EJB is an entity Bean with persistent data. It stores the banking account data in a relational database. First we must create the database on the database server. In our case DB2 was on the same machine as WebSphere. If the database server and the WebSphere server are not the same machine, you need to establish a connection from the WebSphere server to the database server, after creating the database for persistent data. This is described in Appendix E.1.2, “Connect to ACCOUNTS database from WebSphere server” on page 480.

10.3.2.1 Creating the ACCOUNTS database on the database server

Use the following steps to create a DB2 database for our Account EJB:

1. Start the IBM DB/2 Control Center on your database server.
2. Expand the name of your database host and the Instances and DB2.
3. Right-click Databases and select Create -> Database Using Wizard.
4. Fill out the fields on the first page as shown in Figure 259. We entered the name ACCOUNTS for the database and we will use this name later in this chapter.

![Create Database Wizard](image)

*Figure 259. Create ACCOUNTS database in the IBM DB/2 Control Center*

5. Click Finish when you have filled out the first page

The database will be created. You can close the DB/2 control center and continue with creating a data source in WebSphere that uses this database.
10.3.3 Creating a data source for the Account EJB

To establish a connection from the Account EJB to the database you just created, create a WebSphere data source using the following steps:

1. Start the IBM WebSphere administrative console.

2. Select Console -> Tasks -> Create Data source

   In most cases you do not need a new JDBC driver because WebSphere already installed one for the WebSphere administration database. We will also use this driver.

3. Select Use an already installed JDBC driver.

4. Click Next and enter the WebSphere data source name and the real name of the database.

   We used ACCOUNTS for both, as you can see in Figure 260.

5. Click Finish to have the ACCOUNTS data source created.

We now create a container in WebSphere for our EJBs.

10.3.4 Create an EJB container

WebSphere stores all EJBs in containers. We decided to create a separate container for the EJBs that belong to our sample application. Strictly speaking, this step is not absolutely necessary since we could use the
existing Default Container for the EJBs. However, it is convenient to create a new container since this will group our EJBs together in the WebSphere administrative console. In a production environment you may also want to use EJB containers with different settings for different EJBs.

Use the following steps to create an EJB container:

1. Start or give focus to the IBM WebSphere administrative console.
2. Expand the host you want to install the EJBs on. In our case it was Thor-2000.
3. Right-click **Default Server**.
4. Select **Create -> EJBContainer**.
5. Enter the name of the new container.
   We used *Redbook Banking* as our container name, as shown in Figure 261.

6. Click **OK** to create the container.

*Figure 261. Creating a new EJB Container in WebSphere*
You could also define a database source for the EJB container. This data source will be valid for all EJBs in this container, but you cannot specify a create table flag that tells WebSphere to create a new table if necessary, on this level. We need this functionality, which is why we decided to leave the data source tab in the EJB container empty.

We are now ready to deploy EJBs to our container.

10.3.5 Deploy the EJBs

Our sample application contains three EJBs: Account, Transfer, and Personalization.

Account is an entity bean that represents the data of one banking account. We have already created that DB/2 database and a data source in WebSphere for this EJB.

Transfer is a session bean that represents the transaction of moving a certain amount from one account to another. The Transfer EJB uses the Account EJB to read and modify the account data.

Personalization is also a session bean, used to read personalization information from a Domino database and create accounts using the Account EJB.

10.3.5.1 The Account EJB

To deploy the Account EJB do the following:

1. Start or give focus to the IBM WebSphere administrative console.
2. Right-click an EJB container we just created (Redbook Banking) and select Create -> EnterpriseBean.
3. Click Browse and find the deployable .jar file, which contains the EJB. The files for all WebSphere examples including the Account and Transfer EJBs are, by default, located in:

   C:\WebSphere\AppServer\deployableEJBs

   **Note:** There is a deployedEJBs and a deployableEJBs directory. Once an EJB is deployed, WebSphere copies it to the deployedEJBs directory and modifies the jar file with deployment information.

4. Select the jar file that contains the EJB to deploy. The Account EJB is in account.jar.
A jar file can contain several EJBs. If you double-click on the .jar file in the File Open dialog box, all EJBs it contains are listed. However, the account.jar only contains the Account EJB.

5. Double-click **account.jar** to see whether it contains more than one EJB.

If you click the **Select** button instead of double-clicking, all EJBs in the jar file will be selected for deployment. For the account.jar it is OK to do this as well.

6. Select the EJB that is shown and click **Select**.

7. You are asked if you want to deploy the EJB and if you want to activate the WebSphere Workload Management (WLM) for the bean, as shown in Figure 262. We did not use WLM for our sample application.

![Deploy EnterpriseBean dialog box](image)

Click **Deploy only**.

All fields of the General Tab in the Create EnterpriseBean dialog box will be filled out, as you can see in Figure 263.
Do not click OK yet. Now you must specify the data source the account bean will use.

8. Click the **DataSource** tab and then click **Change**.

9. Select the **ACCOUNTS** data source you created before and click **OK**.

10. Enter the name and password of the database user you want to use for this EJB.

   The security for EJBs is handled by WebSphere and not by the database system. All users access the database using the user name and password you specify here.

11. To ensure that WebSphere creates a table for the EJB in case it doesn’t already exist, you must check the Create table box.

   The Data Source Tab of the Create EnterpriseBean dialog box should now look like Figure 264.
12. Click **OK** and the account EJB is created.

### 10.3.5.2 The Transfer EJB

You deploy the Transfer EJB in the same way as we just described for the Account EJB. By default you find it in:

C:\WebSphere\AppServer\deployableEJBs\Transfer.jar

Since the Transfer EJB is a session EJB, you should not specify a data source and can leave all fields on the data source tab empty.

### 10.3.5.3 The Personalization EJB

The Personalization EJB is also deployed using the same procedure as the Account and Transfer EJBs, with the exception that you have to modify the JNDI Home Name. This is because in our servlets we refer to a JNDI home name of *Personalization* instead of the instead the longer default home name that is created by the VisualAge deployment descriptor for the EJB.
Make sure that you set the JNDI Home Name to Personalization, as explained in 9.4.2, “Getting the client stub via the naming service” on page 383. If the EJB is not named properly, it will not be found by the servlets.

Do the following:

- Once the EJB has been deployed, click the deployment descriptor **Edit** button.
- A new dialog opens where the JNDI home name is set to:
  
  com/lotus/banking.ejb/Personalization
  
  Change this to:
  
  Personalization

  The dialog should now look similar to Figure 265 on page 421.

![Deployment Properties dialog](image)

**Figure 265. Setting the JNDI home name**

- Click the **Set** button and then click **OK** to close the dialog.
- Click **OK** to close the deployment dialog as well.

The completed EJB container with the EJBs added is shown in the console view in Figure 266 on page 422.
This completes the deployment of our EJBs. You do not need to start the EJBs as they will be included in our enterprise application, and thus will be started when it starts.

10.3.6 Create a Web application and add servlets

We need to create a new Web application, which we will call **Redbook Banking Web Application**.

1. From the WebSphere console select **Tasks -> Create a Web Application**.
   - Use the Web application name **Redbook Banking Web Application**.
   - Select to enable JSP 1.1 and click **Next**.

---

You must specify JSP 1.1!

For all Web applications in our examples, we specified JSP 1.1 so that JSP tag libraries will be supported.

The default is JSP 1.0, which does not support tag libraries.

You **must** specify JSP 1.1 to enable our sample application.
2. Specify a servlet engine in which to run the Web application.
   - Expand the tree under the server (Thor-2000 in our example)
   - Pick an application server (Default Server in this case).
   - Select a servlet engine within the selected application server (Default Servlet Engine).
   - Click Next.

The next panel allows you to specify the Web path within a virtual host to access the contents of the Web application.

The suggested default Web application will not work because it contains spaces.

3. Change the Web application path to /RedbookBanking as shown in Figure 267.

![Create Web Application](image)

**Figure 267. Setting the /RedbookBanking Web application path**

Note that this path must match the one specified in the setup document of the intra_01.nsf database (except it uses no starting slash) as described in 10.2, “Configuration of the Domino application components” on page 406 and shown in Figure 255 on page 410.

If you want to use another Web application path, you must update the setup document in intra_01.nsf as well as the frameset definition, as described in 10.2.4, “Setting up Web application path for main frame set” on page 412.
Click **Next**.

Our Web application name does not match the directory name we used when we placed our files on the WebSphere server in:

```
\WebSphere\AppServer\hosts\default_host\Redbook Banking
```

Therefore, we cannot use the default values for document root and class path in our Web application.

4. Change the document root, as shown in Figure 268.

- The last part of the document root should be:
  
  \default_host\Redbook Banking\web

- The last part of the class path should be:
  
  \default_host\Redbook Banking\servlets

![Figure 268. Web application advanced settings](image)

5. Click **Finish**.

The Web application is complete. Now we need to add the servlet definitions.

**10.3.6.1 Add servlets**

There are several ways to create servlets. One way is described in 4.5.0.2, “Creating the ReadNames servlet” on page 152. Another easy way is to right-click on the Web application in the IBM WebSphere administrative console and select **Create -> Servlet** from the context menu.
Within the Web application you must now create the servlet definitions shown in Table 19.

### Table 19. Redbook Banking application servlets

<table>
<thead>
<tr>
<th>Servlet Name</th>
<th>Servlet Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login</td>
<td>com.lotus.redbook.banking.DominoLogin</td>
</tr>
<tr>
<td>Search</td>
<td>com.lotus.redbook.banking.Search</td>
</tr>
<tr>
<td>PanelBuilder</td>
<td>com.lotus.redbook.banking.PanelBuilder</td>
</tr>
<tr>
<td>SaveQuery</td>
<td>com.lotus.redbook.banking.SaveQuery</td>
</tr>
<tr>
<td>CallDominoJsp</td>
<td>com.lotus.redbook.banking.CallDominoJsp</td>
</tr>
</tbody>
</table>

Use the servlet name for the Web application path as well.

For example, the Web application path for the Login servlet must be /RedbookBanking/Login, as shown in Figure 269.

![Figure 269. Defining the Login servlet](image)
The completed Web application with all the servlets defined is shown in the console view in Figure 270.

![Redbook Banking Web Application](image)

Figure 270. WebSphere console view of Redbook Banking Web application

We are now ready to define an enterprise application for our EJBs and Web application.

10.3.7 Define an enterprise application

In IBM WebSphere 3.5 the only way to apply security to any resources is to create an enterprise application.

In the WebSphere administrative console you create an enterprise application using the following steps.

1. Select **Console -> Tasks -> Create Enterprise Application**.
2. Enter a name for the application. In our case we used the name Redbook Banking as shown in Figure 271 on page 427. Click **Next**.
3. Select the resources that will be part of the enterprise application.
   a. First, expand **EnterpriseBeans** and select the **AccountHome** bean, as shown in Figure 272.
b. Click **Add** to add the EJB to the enterprise application.

   **Note:** If you do not click the Add button, the EJB will not be added to the enterprise application.

c. Select the **TransferHome EJB** and click **Add** again.

d. Finally, select the **PersonalizationHome EJB** and click **Add** again.

You are now ready to add your Web application to the enterprise application.

4. Expand **Web Applications** and select the name of your Web application, in our case **Redbook Banking Web Application**. Click **Add**.

   Adding the Web application will implicitly add the Web resource (the URI /RedbookBanking) as well.

5. Click **Next** and the dialog box should look like Figure 273 once you expand **EnterpriseBeans** and **Web Applications**.

![Figure 273. Elements of the sample enterprise application](image)

6. Click **Finish** and the enterprise application for our sample application is created.

We are now ready to define security for our application.

### 10.3.8 WebSphere security

There are three aspects involved in configuring WebSphere security:

---

428 Domino and WebSphere Together
• Configure application security
  Specify how to authenticate the users of our application.
• Configure resource security
  Specify which groups (or roles) should be allowed to executed the different methods in our application.
• Configure security permissions
  Specify which users belong to which groups (or roles).

We will describe these three steps in the following sections.

10.3.9 Configure application security

It is necessary to enable application security for an enterprise application to make sure that user name and password are requested when a resource is called by an unauthenticated user. If no application security was applied, the application will raise the error “Authentication failed” instead of displaying the dialog box for user name and password.

To enable application security, do the following:

1. Select **Console -> Tasks -> Configure Application Security** in the WebSphere administrative console.
2. Expand **Enterprise Applications** and select the application you want to switch on security for, in our case Redbook Banking.
3. Click **Next** and the dialog box for the application security settings is displayed, as shown in Figure 274 on page 430.
Figure 274. Enterprise application security settings

These settings are copied from the global WebSphere server security settings and normally you do not have to change them.

4. Click Finish to save the application security settings.

10.3.10 Configure resource security

Now we need to configure resource security. For our purposes, we need to secure:

- The three EJBs: Account, Transfer and Personalization
- The Web resources for accessing our servlets and JSPs

10.3.10.1 Configuring resource security for the Account EJB

To configure resource security for the Account EJB, do the following:

1. In the WebSphere administrative console select Console -> Tasks -> Configure Resource Security.

2. Expand EnterpriseBeans and select the AccountHome bean as shown in Figure 275 on page 431.
3. Press **Next**. As soon as you do this you are asked if you want WebSphere to apply default groups to the EJB’s method, as displayed in Figure 276.

4. Click **Yes**. Every method of the bean is assigned a method group, as you can see in Figure 277 on page 432 where we have expanded two of the methods to see which groups are assigned to them.
5. Click **Finish** to save the assignment of method groups.

### 10.3.10.2 Configuring resource security for the Transfer EJB
Configure resource security for the Transfer EJB by doing the following:

1. Select **Console -> Tasks -> Configure Resource security**.
2. Select **TransferHome** below EnterpriseBeans.
3. Repeat the same process as you did for the Account EJB.

### 10.3.10.3 Configuring resource security for the Personalization EJB
Configure resource security for the Personalization EJB by doing the following:

1. Select **Console -> Tasks -> Configure Resource security**.
2. Select **PersonalizationHome** below EnterpriseBeans.
3. Continue with the same step as you did for the Account EJB but **do not** click **Finish** when you have assigned method groups. Instead, click **Next** to specify a Run-as mode that is different from the default SYSTEM.
4. Set the Run-as Mode to **CLIENT**, as shown in Figure 278 on page 433.
Figure 278. Setting Run-as Mode for Personalization EJB

We need to specify a CLIENT Run-as mode because when we create personalization documents for users we create a Domino session where the user information is picked up from the EJBs security context.

5. Click Finish.

We are now done configuring resource security for our EJBs.

10.3.10.4 Configuring resource security for servlets
In this section we go through one example of configuring resource security for servlets and then we give you a list of all the Web resources you need to repeat this process for. We will configure resource security for the Login servlet using the following steps:

1. Select Console -> Tasks -> Configure Resource security.

2. Expand Virtual Hosts and the name of the virtual host you are using for this example. By default this is default_host.

3. Select the entry for the Login servlet that you created in 10.3.6, “Create a Web application and add servlets” on page 422 as shown in Figure 279 on page 434.
4. Click **Next**.

5. Answer the question if the default method groups should be used with **Yes**.

   For servlets and JSPs the default method groups that WebSphere assigns are shown in Table 20.

   **Table 20. Default method groups for HTTP methods**

<table>
<thead>
<tr>
<th>HTTP request</th>
<th>Default method group</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_DELETE</td>
<td>Remove Methods</td>
</tr>
<tr>
<td>HTTP_GET</td>
<td>Read Methods</td>
</tr>
<tr>
<td>HTTP_POST</td>
<td>Read Methods</td>
</tr>
<tr>
<td>HTTP_PUT</td>
<td>Write Methods</td>
</tr>
</tbody>
</table>

6. Click **Finish**.

You have now configured resource security for the Login servlet by having WebSphere assign the default method groups (or roles) to its methods.

Repeat the above steps to secure the following Web resources in our application:

- /RedbookBanking
Chapter 10. Installation of the banking application

10.3.11 Configure security permissions

We now need to specify permissions to the protected resources. This is where we can assign different users or groups of users to the method group we had WebSphere assign to our application method. We will not setup any advanced permission matrix, but simply add all authenticated user to all of our method groups.

To configure security permissions, do the following:

1. Select Tasks -> Configure Security Permissions.
2. Expand Enterprise Applications; select the Redbook Banking enterprise application and click Next.
3. Select all method groups (Ctrl+a) and click Next.
4. Specify All Authenticated Users for all the method group we selected.
   You can, of course, experiment with more restrictive settings if you wish, but these settings will be adequate to test the installation integrity of the Redbook Banking application.
   Click Next.
5. On the last page you can remove user assignments from method groups.
   We did not remove anything. Click Finish to complete the security permission configuration.

It is not strictly necessary to protect /RedbookBanking itself, but the others should be protected for the application to work properly.

Note: You have to click Finish for each resource. There is no fast way to secure several resources at once when using the administrative console.
This completes the configuration of the WebSphere components of the Redbook Banking application.

10.3.12 Make sure the HTTP plug-in properties files are up to date

If you have Domino and WebSphere on different machines and you haven’t set up automatic update of the Domino HTTP plug-in configuration as described in 3.7.8.3, “Automatic generation of plug-in properties files” on page 97, you must copy the three property files in the WebSphere\AppServer\temp directory on the WebSphere computer to the same directory on the Domino machine. The files are:

1. vhosts.properties
2. rules.properties
3. queues.properties

On the Domino machine add a line like the following to the queues.properties file:

ose.srvgrp.ibmoselink.clone1.host=thor-2000

This is described in more detail in 3.7.8.2, “Manual update of properties files for the HTTP plug-in” on page 95.

You are now done with the setup.

Start the Redbook Banking enterprise application from the WebSphere administrative console so you can test the sample application.

10.4 Installation confirmation testing

To confirm that the application has been installed and configured correctly, simply access it with a Web browser. To open the application, simply use a URL that points to the intra_01.nsf database on your Domino server; it will redirect your browser to the main frameset and initiate the servlets necessary to populate the browser frames. The URL we used in our configuration was:

http://odin.lotus.com/RedbookBanking/intra_01.nsf

You should get a Domino session-based login prompt like the one shown in Figure 280 on page 437.
Chapter 10. Installation of the banking application

Figure 280. Domino SSO login prompt for the Redbook Banking Application

Once you have successfully logged into Domino, the frameset will be displayed as shown in Figure 281.

Figure 281. The Redbook Banking main panel
This completes the installation confirmation for the Redbook Banking example. You can access the authoring database (intra_03.nsf) to add more content if you wish.

**Note:** As you experiment with our sample application, you may experience various peculiarities. For example, if your LTPA cookie has expired when you click on a document link, the login panel will only be shown in the middle frame. It is important to remember that it is not a production or reference application we have deployed, but rather an application where we illustrate the use of many different Domino and WebSphere integration techniques. It is up to you to take what makes sense for your application and then add what is required to make it a full production application.

### 10.4.1 Creating content

In the following section we briefly go through the steps to add a new information area and a content document to our application using the authoring database (intra_03.nsf).

There is much functionality in the Domino part of the application that you can explore by looking at the code. Here we cover a bit of the user functionality to help you identify, as a developer, which parts of the application to explore.

#### 10.4.1.1 Creating a new content area

In our example we call the content areas for divisions. Here we walk through the steps to add an area for the Human Resources (HR) division.

The approval functionality in the authoring database is based on roles, so the first thing we must do is to add a content manager role for our new division to the ACL of intra_03.nsf, using the following steps:

- Open the authoring database with a user ID that has manager access to the database.
- Select File -> Database -> Access control.
- Click Roles and then add the role [CM HR] to the ACL.

The ACL dialog should look similar to Figure 282 on page 439 when the role has been added.
Now we must associate one or more users with the content manager role for the HR division.

- Click **Basics** and associate the [CM HR] role with one or more users.

In a production environment you would associate the content manager role with a group, and then update the group in the LDAP directory (in our case Domino Directory) as people move in and out of the content manager role. If you use a group, there is no need to update the database ACL to handle changes in who is assigned to which role.

We are now ready to define our new content area in the application.

- Open the authoring database with a user ID that has editor access to the database.

- Click the **Intranet Areas** button.

  You must now select from the pop-up list where in the process (which view) the area documents you want to work with are.

- Select to work with **Draft** documents.

  If any other area documents had been under preparation you would have seen them in the view that opens. In our case, no other areas are under preparation, so the view is empty.

- Click the **New Intranet Area** action button.
Fill out the area definition document. We filled out the Division (HR), Group (Policies) and Category 1 (Travel) as shown in Figure 283.

![Figure 283. New content area document](image)

The division name *must* match the part of the role we created coming after “CM”.

Also, your area *must* have one or more category levels in order for associated documents to be published correctly.

- Click the **Ready for Approval** button. Save and close the document.

  The content area is defined. It now must be approved before documents can be published using the area.

- Open the authoring database with a user ID that has the [CM HR] role.

- Click the **Intranet areas** button.

  You must now select what kind of area documents (which view) you want to work with. Draft is no longer a choice.

- Select to work with documents **By Division and Status** and expand the **HR** category in the view.

- Open the newly created area document (Policies / Travel)

- Click the **Approve** action button. Save and close the document.
The area is now published and we can create content for that area.

### 10.4.1.2 Creating a content document

We will now create a travel policy document under our HR division. Following that, we will approve and publish it.

- Open the authoring database with any user ID that has editor access to the database.
- Click the Examples button.

  This opens a view that in our case only has one example, but in production would have ‘templates’ for the different kinds of documents that we will publish.

- Select our sole example document that uses subform su\_ContBlank and you will see a preview of a document in the right frame.

  The preview document has a banner that explains which Domino subform is used and what division this document type is for.

  In our case the subform is a very simple version where just the body field is published. We will now create a content document based on this example.

- In the right preview frame click the Create Content button, as shown in Figure 284.

![Figure 284. Example view of content document](image)
Fill out the new content form. You should enter something for at least the following fields:

- Content name
- Abstract
- Link text (for Navigator)
- Area. Click the **Add Areas** button to select HR / Policies / Travel.
- Body

You can leave the default values for the other fields on the form.

Now ask to have this policy document approved by clicking the **Ready for Approval** button, as shown in Figure 285.

![New content document ready for approval](image)

It is now up to the content manager for HR to approve that our document is published.

- Open the authoring database with a user ID that has the [CM HR] role.
- Click the **Content** button.
  You must now select what kind of area documents (which view) you want to work with.
- Select to work with documents **By Division and Status**.
• Expand the **HR** category in the view until you see the document we just created, as shown in Figure 286.

![Figure 286. Content document ready for approval](image)

• Open the newly created area document (Policies/Travel).
  You will only be able to see the Approve button if your user ID has the [CM HR] role.

• Click the **Approve** action button. Save and close the document.
  The document now is ready to be transferred to the published documents database. In a production environment this could be handled by a scheduled agent, but we have not set any scheduled agent up, so we will invoke the agent to publish our document from the Action menu.

• Open the authoring database with a user ID that has editor access both to this database and to the **published documents** database (intra_01.nsf).

• Select **Action -> Publish Approved Documents**.
  The document will disappear from the view because it changes status to **Published** as it is being copied to the published documents database. Published documents are shown in another view in the authoring database. If you want to update a published document, you can select it from the published documents view and create a new version of it.

**10.4.1.3 The new published content**

You can now see the published content as a user.

• Log into the Redbook Banking application and select **HR** in the division drop-down list on the navigation pane.
Click **Travel** and you will see a list off all documents belonging to the Policy/Travel category in the middle frame. In our case there is only a single document so far. If you click the document link, it will open in the same frame and your application will look similar to Figure 287.

![Figure 287. The new travel policy document in our application](image)

You do not have to go through the selection of division, group and category to find a document. You can have it appear in your right frame as part of your personalization preferences, or you can simply search for it as shown in Figure 288 on page 445. If you click the search result link the document will open in the same way as we saw in the previous figure.
Chapter 10. Installation of the banking application

Figure 288. Search for documents relating to Vegas

Note that the abstract we entered for the document is shown in the search result.

The search result was generated by the user typing in a search argument in a JSP that invoked the Search servlet when the user hit Enter or clicked the GO button. The Search servlet invoked a method in the Personalization EJB which in turn executed a full text search on the Domino database for published documents. The EJB returned the result to the Search servlet, which built the HTML around the result and populated the results frame.

Even though our application just is a sample to illustrate techniques and concepts, there is much more functionality in it than what we briefly have described here. Go ahead and explore on your own!

10.5 Installation and activation of the TransferFunds agent

In 9.6.2, “Calling the Transfer EJB from a Domino agent” on page 390 we showed how you can access an EJB from a Domino agent. To achieve this we used an RMI server. Here we explain what actions are necessary to get our example up and running.
10.5.1 Installing the RMI server

The RMI server passes requests from the agent to the EJB. You have to install the RMI server on your WebSphere server. The classes of the RMI server are part of the Banking.jar file you already installed (see 10.1.2, “Placement of the WebSphere application components” on page 404).

Because the RMI server is a stand-alone Java application, it must authenticate with the WebSphere server. The parameters for this authentication are in the file sas.client.props. We also created a batch file, transfer.bat that starts the RMI registry and the RMI server. Put these two files into the same directory as the Banking.jar. We copied it to:

C:\WebSphere\AppServer\hosts\default_host\Redbook_Banking\servlets

Then you edit transfer.bat and change the following three variables if necessary:

- **WAS_HOME** must contain the WebSphere Home directory. By default this is C:\WebSphere\AppServer
- **BANKING_PATH** must contain the path to the directory that contains the files Banking.jar and sas.client.props.
- **DEF_HOST** contains the host name of your WebSphere server.

The rest of the file must not be changed. Our Transfer.bat file looked like Figure 289.

```batch
@ECHO OFF
ECHO Starting RMI Server...
set WAS_HOME=C:\WebSphere\AppServer
set BANKING_PATH=%WAS_HOME%\hosts\default_host\Redbook_Banking\servlets
set DEF_HOST=thor-2000.lotus.com

set CLASS_PATH="%BANKING_PATH%\Banking.jar;"%WAS_HOME%\lib\ejb.jar
set CLASS_PATH=%CLASS_PATH%;%WAS_HOME%\lib\uc.jar
set CLASS_PATH=%CLASS_PATH%;%WAS_HOME%\deployedEJBs\DeployedAccount.jar
set CLASS_PATH=%CLASS_PATH%;%WAS_HOME%\deployedEJBs\DeployedTransfer.jar
set CLASS_PATH=%CLASS_PATH%;%WAS_HOME%\jdk\jre\lib\ext\rmiorb.jar
set CLASS_PATH=%CLASS_PATH%;%WAS_HOME%\jdk\jre\lib\ext\iioprt.jar
set CLASS_PATH=%CLASS_PATH%;%WAS_HOME%\lib\sslight.jar
set CLASS_NAME=com.lotus.redbook.banking.AgentTransferServer
set PROP_FILE=-Dcom.ibm.CORBA.ConfigURL=file:"%BANKING_PATH%\sas.client.props"
start %WAS_HOME%\jdk\jre\bin\rmiregistry

Figure 289. The Transfer.bat file
```
After saving and closing the Transfer.bat file, you call it. The RMI server is started and displays the lines you see in Figure 290.

![Figure 290. The RMI server after it was started](image)

### 10.5.2 Activating the Transfer Funds agent

You find the Transfer Funds agent in the Personalization Domino database (intra_02.nsf).

The classes of this agent are simple examples. This is why they contain hard coded user names, passwords and WebSphere host names. You cannot run the agent without modifying these three values and reimporting the class files.

The source code of the classes we are using is in BankingSource.jar. You can import this file into your VisualAge for Java installation.

The agent code is discussed in detail in 9.6.2.1, “The AgentTransfer class” on page 390. You must change the values for the variables sUser and sPassword and you also have to change the server you are connecting to using the `java.rmi.Naming.lookup` method. After these changes, create new class files for the classes you need to import into the agent.

See 9.6.2.5, “Creating the Transfer Funds Domino agent” on page 398 for an explanation of how you can import the new class files into the agent.

Then you can schedule the agent for hourly execution on your Domino server.

We did not create a Web interface that allows the users to enter amounts they want to transfer regularly. To enter values for these amounts you open the personalization database using a Notes client and modify the personalization documents you find in the view viref100PersByName. Enter the values in the field Monthly savings, as you can see in Figure 291 on page 448. Do not modify the account numbers because they match with account numbers in DB/2.
This completes our description of the installation and activation of the TransferFunds agent.

10.6 Summary

In this chapter we have taken the different elements of our application, described in the other chapters in part 2 of this book, and deployed them as one secure application.

We have shown how to set up the Domino database and how to link it to the WebSphere elements of the application. We have set up a secure WebSphere enterprise application so you can make a real-life test of the combined Domino and WebSphere application.

Finally, we described how to deploy the example, and how to access a WebSphere EJB from a Domino agent.
Appendix A. Configuration of thin servlet redirector for Domino

In this appendix we cover the steps to set up a connection to a WebSphere server from a Domino machine using the thin servlet redirector that is described in Chapter 3.7.2, “Servlet redirectors” on page 78.

Ensure that you have performed the steps in 3.7.6, “Installation of WebSphere components on the Domino server” on page 89 and 3.7.7, “Configuring Domino to use the WebSphere plug-in” on page 91 before beginning.

There are three steps to perform:

1. Define the ports that the administration server and the application server will use for IIOP. This step is necessary if you are using a firewall between Domino and WebSphere, but may be omitted in a test environment. If omitted, WebSphere will use a random port number. The port numbers are specified as command line arguments to the JVMs when they start.

2. Configure the thin servlet environment.
   
   a. Edit the script (batch) files used in the next steps to reflect the environment. There are two script files, one to create the properties files and one to start the servlet redirector.

   b. Generate the properties files (queues, rules and vhosts).

   c. If you have enabled WebSphere (Global) security and do not wish to be prompted for authentication, edit the sas.client.props file to change the prompt type and add the user ID and password to log into WebSphere.

3. Start the thin servlet redirector.

A.1 Defining IIOP port numbers for WebSphere

Again, this step is only necessary if you need to know the port numbers that will be used by the servlet redirector. This would be necessary in order to configure a firewall to allow traffic, but would be unnecessary in a test environment. There are two ports to be configured:


   Briefly, edit the admin.config file in the WebSphere\AppServer\bin directory and add the argument -Dcom.ibm.CORBA.ListenerPort=33000 (the actual port number is arbitrary, but must not already be used) to the end of
the line beginning with:

Remember that you must stop and restart the WebSphere administrative server for this change to become effective. This step is necessary to allow automatic configuration of the thin redirector.

2. Now the application server must be similarly modified to accept a port number. From the WebSphere Advanced administrative console, locate your server under the host name you installed WebSphere on. In our case this was Default Server under Thor-2000. Highlight the application server entry and stop it. Once it has stopped, select the General tab on the right-hand pane and enter this string in the field Command LineArguments as shown in Figure 292:

-Dcom.ibm.CORBA.ListenerPort=36000

Again, the actual port number is arbitrary, but must be unique on the computer host running the WebSphere administrative server.

This port is used by the application server to listen for IIOP requests. In our case, the requests will be from the thin servlet redirector calling the RemoteSRP bean.

You can restart the application server; its JVM will now use the port number entered on the command line for incoming IIOP connections. The rest of the discussion assumes that you have already generated all the host aliases (as

Figure 292. Specifying IIOP port for application server for thin servlet redirector use

6 Domino and WebSphere Together
described in 3.5.1.1, “Adding aliases to the WebSphere default_host” on page 62) and any application definitions you wish to test. In our case, testing with the default applications installed with WebSphere is all that is necessary. Any later changes will require rerunning the configuration script on the Domino HTTP server.

A.2 Configuring the thin servlet redirector environment

Using a text editor, open the iiopredirector.xml file in the \WebSphere\AppServer\properties directory. It is not necessary to alter the transport section(s). Instead, edit the lines beginning with <admin-node-name> and <name-service-node-name>. Change the entry for localhost to the hostname of your WebSphere server. In our case, this was Thor-2000. Thus, for our example, we changed the lines:

```
<admin-node-name>localhost</admin.node.name>
<name-service-node-name>localhost</name-service-node-name>
```

to:

```
<admin-node-name>Thor-2000</admin.node.name>
<name-service-node-name>Thor-2000</name-service-node-name>
```

This is shown in Figure 293.

![Figure 293. Editing the iiopredirector.xml file](image)

This file will not need to be changed again unless you want to add more target WebSphere hosts to your configuration.
Now, edit the script (batch) file thinRedirectorConfig.bat. This file is located in the \WebSphere\Appserver\bin directory on your Domino server.

Edit the Java command at the end of the file and add the arguments:

- **serverRoot**
  Set to the WebSphere directory on the target WebSphere server. By default this is set to the environment variable %WAS_Home%, which in turn is set to C:\WebSphere\AppServer. If this is correct for your WebSphere server (not the WebSphere code installation on the Domino server), you do not need to change the default entry.

- **adminNodeName**
  Set to the host name of your WebSphere Server (we assume that you have not changed the default installation). In our case, we specified Thor-2000.

- **queueprops**
  This is the local path to the iiopredirector.xml file we just edited. By default this is C:\WebSphere\AppServer\properties\iiopredirector.xml; you should ensure that it matches your environment.

Note that all of these arguments must be on the same line as the Java command; do not insert line breaks into this line. The results of our configuration are shown in Figure 294 on page 453.
In the figure, the lines we changed are shown in bold type; we inserted line breaks for figure clarity, but you should not do this.

Ensure that the WebSphere application server is running on your WebSphere server and run the thinRedirectorConfig.bat file; this will generate the properties files needed by the DSAPI plug-in. You may inspect these if you wish, but they should be ready for use without further alteration. Note that these files will not be the same as the ones on your WebSphere server; instead they refer to the local OSE queue that is used by the DSAPI plug-in to communicate with the thin servlet redirector.

You can also run the batch command and specify more arguments:

- `-queueName` *(queue1)*
- `-queuePort` *(8110)*
- `-queueType` *(local)*

in place of the `-queueProps` argument; if you do this, you do not need the iiopredirect.xml file.
There is also a file for starting the thin servlet redirector in the same directory called thinRedirectorStart.bat. This file does not need to be changed unless you wish to divert output to a log file as we did. The copy we used is shown in Figure 295.

Again, we inserted line breaks in the Java command line for clarity; you must not do this or the program will not execute. Note that we redirected output to a log file. We also show logging with tracing all events. You may wish to change this to disable tracing in a production environment; otherwise the log file will grow rapidly.

This completes the configuration of the thin servlet redirector environment.

Remember that, if you have enabled WebSphere Global security (described in 4.4.3, “Configuring WebSphere V3.5 global security for single sign-on” on page 127), you will get a password prompt to log into the WebSphere administrative server when you run these batch files. If you wish to automate the authentication process, you can edit the file sas.client.props in the \WebSphere\AppServer\properties directory, as we describe in 3.7.8.3, “Automatic generation of plug-in properties files” on page 97.
A.3 Testing the thin servlet redirector with Domino

Start the WebSphere server and WebSphere console; ensure that your application server is running.

Start the thin servlet redirector by running the thinRedirectorStart.bat file in the \WebSphere\AppServer\bin directory. It is convenient to open a command prompt window to view the command, but this is not absolutely necessary. When the servlet redirector is ready for service you will see the line:

| IIOPredirecto A SERVE0034I: Servlet Redirector Running... |

This line will only be shown if you do not enable logging; if you do, it will be written to the log file. To stop it, you must enter Ctrl-C in the command window and enter y or n (it does not matter which) to the prompt “Terminate batch job?”. For now, leave it running.

Start your Domino server and ensure that the HTTP task is loaded and has loaded the WebSphere DSAPI plug-in. There will not be any communications messages on the console.

Open a Web browser and point it at your Domino server (in our case, odin.lotus.com) and request a WebSphere resource. We chose /webapp/examples/showCfg for our test.

You should see a panel resembling the one in Figure 296 on page 456.
Domino and WebSphere Together

Figure 296. Confirmation of thin servlet redirector for Domino DSAPI plug-in

Note that the hostname (circled in the figure) is the host name of the WebSphere server, not the Domino server running the HTTP task (this is shown in the URL line circled in the browser). Also note that the transport shown on the left of the panel is “Local pipes.” This is the transport used for OSE communications on the WebSphere server and would be used by an HTTP server (for example, the IBM HTTP server) if it were installed on the server.

Note that the properties files installed in this step will need to be recreated if there are any changes to the WebSphere server, such as:

- Adding or removing a URL (Web resource)
- Securing or unsecuring a URI
- Adding or removing a host alias
- Changes to the queue properties of a servlet engine (name and/or port)
- Adding or removing a servlet engine
- Adding or removing a server clone

This completes the configuration and testing of the thin servlet redirector.
It is possible to switch between using the thin servlet redirector and OSE remote by simply running the configuration batch files (and starting the thin servlet redirector batch file). It is not absolutely necessary to stop the HTTP task when you do this, although it would be prudent to do so. Of course, in a production environment such a switch “on the fly” would likely never be necessary or desirable.
Appendix B. The ReadNamesRemote Servlet

The RemoteNames servlet was designed to illustrate single sign-on (SSO). It accesses a Domino Directory using authentication credentials extracted from the LTPA (SSO) token created during session-based sign-on. The servlet sets up a session to the the Domino server from WebSphere using IIOP; therefore the Domino server need not be (but can be) on the same computer as WebSphere. The source code is shown in the following figures.

The code shown in Figure 297 is the initialization section for the servlet.

```java
// JDK classes
import java.io.*;
import java.util.*;

// Servlet classes
import javax.servlet.*;
import javax.servlet.http.*;

// Domino classes
import lotus.domino.*;

public class ReadNamesRemote extends javax.servlet.http.HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws javax.servlet.ServletException, java.io.IOException {
        // Start the HTML code
        res.setContentType("text/html");
        res.setHeader("Pragma", "no-cache");
        res.setHeader("Cache-control", "no-cache");
        PrintWriter out = res.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>People</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
```

Figure 297. ReadName servlet Initialization code
The highlighted line:

```java
// Domino classes
import lotus.domino.*;
```

makes the Domino class library accessible to the servlet. Once we instantiate a Printwriter, we output the HTML header lines for the returned output.

First, we get the LTPA token using a helper method (shown in Figure 301 on page 462) to retrieve the user’s authentication credentials.

```java
try {
    // get the Ltpa Cookie (i.e. the user credentials)
    String sToken = getCookie(req, "LtpaToken");

    if (sToken == null) {
        out.println("<BR>ERROR: User not logged in!");
    } else {
        // start the IIOP session
        String sDominoHost = req.getParameter("Host");
        Session sesCurrent = NotesFactory.createSession(sDominoHost,
            sToken);
        String sCurrentServer = sesCurrent.getServerName();
        // open Domino directory
        Database ndbNames = sesCurrent.getDatabase(sCurrentServer,
            "names.nsf", false);
        if (!ndbNames.isOpen()) {
            ndbNames.open();
        }
        // print heading
        out.println("<H1 ALIGN="CENTER">Persons in Domino directory</H1>");
        out.println("<H2 ALIGN="CENTER">User: " +
            sesCurrent.getUserName() + "</H2>");
        out.println("<TABLE>");
    }
}
```

Figure 298. Initializing Domino Access

If the LTPA token is not found, the servlet exits with an error message; otherwise, an IIOP session is started using the NotesFactory.getSession method using the hostname supplied as a parameter to the servlet (from the URL) and the LTPA token. Thus the IIOP session connection to the specified Domino server is opened with the user’s authentication credentials. This ensures that Domino objects’ access control will be enforced on the remote Domino server.
We then open the Domino Directory (*names.nsf*) and write the heading lines for the Directory entries.

Now we retrieve the entries in the Domino Directory by opening the People view and retrieving all entries as shown in Figure 299. Each entry is displayed in a table row. Each cell of the row contains the column entries for the retrieved person.

```java
// loop through all person entries and print them
View vwPeople = ndbNames.getView("People");
ViewEntryCollection vwcolPeople = vwPeople.getAllEntries();
ViewEntry vwentPerson = vwcolPeople.getFirstEntry();
Object objCell = null;

while (vwentPerson != null) {
    out.println("<TR>");
    Enumeration esAllColumns =
        vwentPerson.getColumnValues().elements();
    while (esAllColumns.hasMoreElements()) {
        objCell = esAllColumns.nextElement();
        if (objCell == null) {
            out.println("<TD> </TD>");
        } else {
            out.println("<TD>" + objCell.toString() +
                       "</TD>");
        }
    }
    out.println("</TR>");
    vwentPerson = vwcolPeople.getNextEntry();
}
```

*Figure 299. Retrieving People from the Domino Directory*

Once we exit the loop through the people entries, we clean up and exit as shown in Figure 300 on page 462. Note that the recycling of the session object also recycles (that is, releases) storage for other Domino objects (the database and view objects).
The `getCookie` method simply cycles through all the cookies (there could be more than one), searching for the LTPA cookie using the supplied string (“LtpaToken”). If found, it is returned. If not found a `null` string is returned.

```java
private String getCookie(HttpServletRequest req, String sCookieName) {
    Cookie [] cookies = null;
    try {
        cookies = req.getCookies();
        if (cookies != null) {
            for (int iCookieCounter = 0; iCookieCounter < cookies.length; iCookieCounter++) {
                if (cookies[iCookieCounter].getName().toLowerCase().equals(sCookieName.toLowerCase())) {
                    return cookies[iCookieCounter].getValue();
                }
            }
        }
        return null;
    } catch (Exception e) {
        return null;
    }
}
```
Appendix C. Using WebSphere advanced LDAP properties

We referenced the fact that WebSphere Global security allows configuration of advanced LDAP properties in 4.4.3, “Configuring WebSphere V3.5 global security for single sign-on” on page 127. In most cases the default settings shown in Figure 302 will be adequate, but you may wish to change them in certain circumstances.

![LDAP Advanced Properties](image)

Figure 302. LDAP Advanced Properties: WebSphere Global Security

The field “Initial JNDI context Factory” should be left at its default value of com.ibm.jndi.LDAPCtxFactory unless you have a replacement jar file known to provide equal or better functionality. We do not recommend changing this field unless you are told to do so by IBM support in tracing a specific problem.

Note that the Directory Type will be shown as *Domino 5.0* when you enter the panel; the entries following will reflect this choice. If you change the Directory Type field value to one of the other supported directories, your selection will also be reflected on the User Registry panel (shown in Figure 78 on page 130) when you click OK. If you do change this entry to another directory, you will also have to change Domino to use Directory Assistance (as shown in the Domino Administrator’s help database) to use the chosen directory for Web authentication. We did all of our testing using the Domino R5.0.6a LDAP task with satisfactory results.
Note that, if you change any of the other fields, the directory type is changed to Custom to reflect the fact that your entries are no longer the default for the Domino 5.0 directory.

The next entry, User Filter, will by default be set to:

\[(\&(uid=%v)(objectclass=dominoPerson))\]

where \%v is substituted with the string a user will enter at an authentication challenge. This filter is used to create the ID WASAdmin at the authentication prompt would be searched for using the string:

\[(\&(uid=WASAdmin)(objectclass=dominoPerson))\]

### LDAP anonymous access

This discussion assumes that you have your LDAP directory configured to accept anonymous queries.

If you do not allow anonymous access to your LDAP directory, you must use **fully distinguished name** to authenticate. The contents of the user file are not used to parse the input string.

The steps necessary to prevent anonymous access are described in Appendix D, “Securing the LDAP server from anonymous access” on page 471.

The meaning of the filter is that the search is looking for an entry with the attribute `uid` set to `WASAdmin` and belonging to the objectclass `dominoPerson`. The objectclass is part of the Domino LDAP schema and cannot be changed; however, it is possible to change the field to base the search on if you wish. We tested:

- \[(\&(dn=%v)(objectclass=dominoPerson))\]
- \[(\&(cn=%v)(objectclass=dominoPerson))\]

for distinguished name and common name, respectively. Both of these allowed us to authenticate. However, if you do make these format changes, you must also change the format of the field Security Server ID on the User Registry panel to match your chosen format. Figure 303 on page 465 shows the entry for common name (`cn`) authentication.
Appendix C. Using WebSphere advanced LDAP properties

Figure 303. Specifying Common Name Authentication for Administrator

It is also possible to allow multiple entry formats; you can also change the filter entry to designate this. For example, if you wished to allow people to authenticate with any of:

- common name (cn)
- distinguished name (dn)
- or user id (sn)

you would change the “User Filter” field to:

\((\&(|(cn=%v)(sn=%v)(dn=%v))(objectclass=dominoPerson))\)

This simply tells WebSphere to create an LDAP query to search substituting the entry in the authentication challenge for %v in each of the filter entries. Even though in most cases only one of these will match, the or specification (I) allows locating a person even if the other two cannot be matched. Note that the Domino LDAP task will still retrieve all attributes for the person; the filter is only used to search for the person. Note, as well, that this implies that all three types of entries (common name, shortname and distinguished name) must be unique since authentication requires a unique match to a single entry in the directory. Although Domino will force shortname (or uid) and distinguished name (or FullName) to be unique, there is clearly no such control on common name since this is a person’s name; in practice, duplicates are fairly frequent in large groups.
Care should be taken when changing the authentication string as we describe if you have automated client logins; the com.ibm.login.CORBA.loginUserid field in the properties file sas.client.props must be formatted consistently with the User Filter field. The configuration of sas.client.props is described in 3.7.8.3, “Automatic generation of plug-in properties files” on page 97.

You must have a distinguished name in the Person document!

Both Domino and WebSphere can be configured to accept any or all of:

- shortname or user ID (WASAdmin in our example)
- distinguished name (cn=WebSphere Administration/o=DOMWAS for Domino, cn=WebSphere Administration,o=DOMWAS for WebSphere [more precisely, for an LDAP query])
- common name (WebSphere Administration)

We tested and found that we could authenticate with either Domino or WebSphere using any of these strings (by ensuring that the User Filter in WebSphere’s LDAP Advanced Properties was configured appropriately).

However, in all cases, we found that, for the LTPA Token to be acceptable to the other server, the user’s Person document had to have their distinguished name in the FullName field and it had to be first in this field if there was more than one entry.

The FullName field has a label of “User name:” in the Person document form.

The entry for (objectclass=dominoPerson) could be changed to any of

- objectclass=ePerson
- objectclass=organizationalPerson
- objectclass=Person
- objectclass=inetOrgPerson

since these all map to the Domino Person document, but there is no advantage to do so.

The “Group Filter” field works similarly for group names; we did not test this entry since in most cases the default mapping to group name is all that would be valid.
The remaining entries “User ID map” and “Group ID map” are used to map returned LDAP entries against permissions. In other words, once a user is authenticated, this field describes what string will be used to test access control or permissions entries. Figure 304 on page 468 shows the relationship between the input specification User Filter and the returned User ID Map entry. In our example, we have a (default) User Filter of:

\[(\&(uid=%v)(objectclass=dominoPerson))\],

which, when the specific ID of WASAdmin is specified, tells LDAP to search for an object whose uid field contains WASAdmin and whose objectclass is dominoPerson. WebSphere attempts to bind to the (unique) person using the input field (WASAdmin) and the supplied password. If the bind is successful, LDAP then returns the field specified by the User ID Map. In our case, we specified the default of dominoPerson:shortname. The LDAP directory returns the contents of the shortname field for the retrieved (the one we were able to bind to) as a string that Websphere then uses to compare to permission lists. Although in this case uid and shortname are identical (since they map to the same field in the directory), this is not be required. For example, it would be possible to change the User ID Map field to return Fullname to retrieve the user's fully distinguished name. We tested a configuration with this setting and also one with User ID Map set to dominoPerson:cn (that is, common name or “flat” name) successfully.
We tested User ID map changes. For most of our testing against Enterprise application permissions we used the default entry of `dominoPerson:shortname`. (Note that this is the same as `dominoPerson:uid` since Domino maps both of these to the person’s Shortname/UserID field in their Person document in the Domino directory).

You can most easily see the format of the mapped entries by opening the permissions dialog for one of your Enterprise Applications. The default mapping (`dominoPerson:shortname`) was shown in our description in Figure 123, “Reviewing permissions for the enterprise application” on page 168. If we change the User ID map to `dominoPerson:fullname` (in other words, distinguished user names), the permissions panel retrieves names and displays them using this mapping as shown in Figure 305 on page 469.
We further found that, even though we had defined the permissions for our application using the default mapping of "dominoPerson:shortname", the permissions for the Enterprise application were dynamically changed to distinguished names format as shown above. Our testing showed that the mapping appeared to leave application security unchanged with the sole exception that retrieving the user identification from the context of an EJB returned a null unless we used the default (dominoPerson:shortname) mapping.

We did not test using client certificates so we cannot comment on the usage of the last two entries on the LDAP Advanced Properties panel.

Finally, when we used the default entry for User Filter of (uid=%v) we found that WebSphere accepted a user entry of shortname (uid) as expected, but also accepted a name in distinguished format. Thus both the entries WASAdmin and cn=WebSphere Administration,o=DOMWAS were accepted for authentication and successfully authenticated; apparently WebSphere checks for the token cn= and adjusts its LDAP query accordingly.
Appendix D. Securing the LDAP server from anonymous access

In our discussion in 4.4.2, “Configuring the Domino server to support LDAP and IIOP” on page 123 we allowed anonymous read access to the Domino LDAP directory. In this appendix we describe how to limit access to the LDAP directory.

D.1 Limiting fields an anonymous reader can query

We mentioned how to check what fields an anonymous reader can retrieve from any individual entry in 4.7, “Troubleshooting single sign-on” on page 175. Often the default settings are acceptable, but companies may very well wish to restrict anonymous access to the LDAP directory by:

- Limiting the fields an anonymous reader can retrieve from an individual entry in the directory
- Limiting the number of entries an anonymous reader can retrieve in a single query

Both of these can be modified from their default settings by opening (or creating) the server’s LDAP Configuration document. Using the Domino administrator client, select the Configuration tab, expand the Directory entry in the navigator on the left and select Directory Settings. The resulting configuration document is shown in Figure 306 on page 472.
Figure 306. Domino LDAP directory settings

The list of fields that can be queried by an anonymous user is shown in the upper part of the panel. If you click the **Edit Directory Settings** action button, a new action button with the label `<<>>` will be displayed above the list of fields. If you click this button, you will be presented with a dialog box (shown in Figure 307) that will allow you to remove (and add) entries to the list.

Figure 307. Adding or deleting fields an anonymous LDAP client can query

Using this panel you can restrict the ability of anonymous users to retrieve information that could be sensitive (such as mail addresses). Naturally, limiting access in this way must be balanced against legitimate uses of the directory to retrieve other's telephone numbers, mail addresses, public keys, and other
information necessary to locate and communicate with the person. You will, in any case, see the results of your selection in the right pane of Figure 307; as well it will be displayed when you click **OK** and return to the LDAP Settings document as shown in Figure 306. On the LDAP Settings document, you can further limit the number of entries returned in the field “Maximum number of entries returned” to discourage scanning the LDAP directory; again you will want to balance restricting this value against the need to allow users to select from many similar entries to locate the one they want to read.

You will not likely want to enable anonymous users the ability to write the directory, but this is also possible from this panel using the field “Allow LDAP users write access.”

Finally, you can increase the number of characters an LDAP client must enter to perform an LDAP query by changing the default of 1 to a larger value in the field “Minimum characters for a wildcard search.” Again, you will want to balance the greater restriction on acceptable search strings against the ability of users to locate entries with minimal information. However, the default setting allows searching with “*”; this allows retrieving all entries. Forcing a user to enter at least two characters, including the first letter, would not likely pose a significant hardship to locate entries.

### D.2 Requiring LDAP clients to authenticate to retrieve directory entries

In many cases, limiting anonymous clients’ access in the ways we describe will be adequate. However, there are situations where you will wish to disallow any anonymous access because of your company’s policies, the size of the directory, or the sensitivity of the directory information. A common example would be an LDAP directory used to authenticate customers over the Internet. If the directory contains many entries (hundreds of thousands or millions), even the existence of the entries should be concealed from casual view.

In our scenario we authenticated with the Domino LDAP directory on the same physical server that ran the HTTP server; in practice, even if we used Domino LDAP, we would likely place this directory behind a firewall separating it from the HTTP server(s) in the DMZ. In this case, clients would never access the LDAP directory directly; Domino or WebSphere would gather user credentials and attempt to authenticate across the firewall. However, even in a secure environment like this, the potential exists for servers in the DMZ to be compromised. If this were to occur, a malicious user could usurp the client privileges of the Domino or WebSphere server and access the LDAP
Forcing any access to the directory to be by authenticated users only will reduce this exposure.

D.2.1 Securing the Domino LDAP server from anonymous access

The actual setting to secure the Domino LDAP directory from anonymous access is very simple: open the server document for the server that runs the LDAP directory, select **Ports, Internet Ports** and **Directory** and change the entry **Authentication Options - Allow Anonymous access** from its default setting of No to Yes, as shown in Figure 308.

![Figure 308. Preventing anonymous access to the LDAP server in Domino](image)

Once you save the server document, stop and restart the LDAP task by typing:

```
tell ldap quit
```

followed by:

```
load ldap
```

At this point any user who wishes to query the LDAP directory will have to supply a distinguished name and password to authenticate. The console entries will be similar to those shown in Figure 309 on page 475.
You can further confirm that anonymous access is not allowed by using the ldapsearch command line utility as we described in 4.4, “Setup of the Domino-WebSphere single sign-on environment” on page 122. If you attempt the search as shown in Figure 71, you will receive an error message: “Bind failed: Anonymous bind not allowed” as shown in Figure 310.

You should confirm that you can access the Domino LDAP search using ldapsearch before proceeding. To do this you need to add the parameters -D and -w with the fully distinguished name of an entry and the password, respectively. LDAP will bind using the distinguished name-password pair before executing the search as shown in Figure 311 on page 476.
Domino and WebSphere Together

Note that all fields in the Person record are returned now that access has been accomplished through authenticated entity.

This completes the Domino configuration of LDAP to prevent anonymous access to the LDAP service. Note that, unless you use directory access to authenticate Web users, these changes will not affect the acceptable formats users can use to authenticate with the Domino HTTP server.

However, all access to the LDAP directory must now specify a distinguished name and the corresponding password; user ID (shortname) or common name will not be accepted.

**D.2.2 Configuring WebSphere to provide a distinguished name to bind**

Open the Global Security settings dialog by selecting **Tasks->Configure Global Security Settings** and selecting the **User Registry** tab. On this panel you need to:

- Change the format of the “Security Server ID” field to its distinguished format. In our case, for the existing uid of WASAdmin, the corresponding distinguished name was **CN=WebSphere Administration,O=DOMWAS**. Note that you must specify the ID in this format when you receive a security prompt on starting the WebSphere Console.
The password entered in “Security Server Password” should not need to be changed unless you changed the identity of the Security Server ID.

- Enter the fully distinguished name to bind to the LDAP directory in the field “Bind Distinguished Name.” This need not be the same as the entry in “Security Server ID” (but it can be).

- Enter the password corresponding to the name used in the “Bind Distinguished Name.”

The settings as we entered them are shown in Figure 312. We chose to use a different ID for the Bind Distinguished Name than the Security Server ID.

![Figure 312. Entering Bind Distinguished Name in WebSphere Global Security Settings](image)

Click **Finish**. Once you have received confirmation of the security reconfiguration, stop (for restart) your server; the console will exit automatically once you accept the warning that it will exit once the WebSphere server has shut down.

The implications of this change are that all future access to the WebSphere Server that needs to be authenticated will require that users specify their **fully distinguished name** to authenticate and that any settings in the “User Filter” field of the LDAP Advanced Properties (as described in Appendix C, “Using WebSphere advanced LDAP properties” on page 463) will be ignored.

However, if you are using SSO, this change may well be transparent to your end users if they first authenticate with Domino using session-based
authentication and have an LTPA token when they access the WebSphere server since they will, of course, not be challenged by WebSphere for authentication credentials.

We did not require that authenticated access to the LDAP directory require an SSL session. In reality, we would require this if we also required authenticated access to the LDAP directory. This would require setting up a certificate authority and issuing SSL certificates to both the Domino server (with the LDAP task) and the WebSphere server, enabling both to load their certificates, and finally requiring SSL access to the LDAP directory.
Appendix E. Installing the IBM WebSphere 3.5 banking example

To use the servlets and EJBs of the banking example that comes with IBM WebSphere 3.5, you have to install them correctly so that you can access them.

We used the Account and Transfer EJBs from this example in our own sample application. The installation of our sample is described in Chapter 10, “Installation of the banking application” on page 401.

If you only want to try installing and securing a WebSphere application that uses EJBs, you can follow the instructions in this appendix.

Note: If you want to install both our redbook sample application and the WebSphere banking example described in this appendix, you should use the same EJB container for both applications.

E.1 Creating a database for banking account data and connecting to it

The Account EJB stores the banking account data in a relational database. First the database is created on the database server. If the database server and the WebSphere server the account EJB will run on are not the same machine, you need to establish a connection from the WebSphere server to the database server after this.

E.1.1 Create ACCOUNTS database on the database server

Start the IBM DB/2 Control Center on you database server. Expand the name of your database host and the Instances and DB2. Right-click on Databases and select Create -> Database Using Wizard. Fill out the fields on the first page as shown in Figure 313 on page 480 and click Finish. We entered the name ACCOUNTS for the database; we will use this name later in this appendix.
Connect to ACCOUNTS database from WebSphere server

If your database server and WebSphere server run on the same machine, you can skip this section.

To set up the connection to a remote database server, follow these steps:

1. Start the IBM DB/2 Client Configuration Assistant on your WebSphere server.
2. Click Add and select Search the network.
3. Click Next>> and select your database server. If you have already connected to this server, it is displayed below Known Systems, otherwise you find it below Other Systems.
4. Expand your database server and select the ACCOUNTS database, as shown in Figure 314 on page 481.
5. Click Finish and close the client configuration assistant.

### E.2 Deploying the banking example EJBs

The IBM WebSphere 3.5 banking example contains the two EJBs Account and Transfer.

Account is an **entity** bean that represents the data of one banking account. For this it needs a WebSphere data source, which is a connection to a database.

Transfer is a **session** bean that represents the transaction of moving a certain amount from one account to another. The Transfer EJB uses the Account EJB to read and modify the account data.
E.2.1 Create a data source for the Account EJB

To establish a connection from the Account EJB to the database you created in the previous section, create a WebSphere data source using the following steps:

1. Start the IBM WebSphere Administrative Console and select **Console -> Tasks -> Create Data source.** In most cases you do not need a new JDBC driver because WebSphere already installed one for the WebSphere administration database, so you select **Use an already installed JDBC driver.**

2. Click **Next** and enter the WebSphere data source name and the real name of the database. We used **ACCOUNTS** for both, as you can see in Figure 315.

![Figure 315. Creating a new data source for WebSphere](image)

E.2.2 Create a container for the account example EJBs

WebSphere 3.5 stores all EJBs in containers. We decided to create a separate container for the EJBs that belong to the banking example. Use the following steps to create an EJB container:

1. Expand the host you want to install the EJBs on and right-click **Default Server.**

2. Select **Create -> EJBContainer** and enter the name of the new container, as displayed in Figure 316 on page 483.
If you define a data source on container level, it is valid for all EJBs in this container, but you cannot specify a “create table” flag that tells WebSphere to create a new table if necessary, on this level. This is the reason why we decided to leave the data source tab in the EJB container empty.

### E.2.3 Deploy and create the Account EJB

Use the following steps to create an EJB:

1. Right-click on an EJB Container and select **Create -> EnterpriseBean**.
2. Click **Browse** and find the deployable .jar file, which contains the EJB. The files for all WebSphere examples are by default located in C:\WebSphere\AppServer\deployableEJBs.

   Note that there is a deployedEJBs and a deployableEJBs directory. The Account EJB is in account.jar. If you double-click the .jar file, all EJBs it contains are listed.

3. Double-click account.jar and select the EJB it contains. When you click on **Select** now, you are asked if you want to deploy the EJB and if you want to
activate the WebSphere Workload Management (WLM) for the bean, as shown in Figure 317.

4. For this example application you do not need WLM, so click **Deploy only** and all fields of the General Tab in the Create EnterpriseBean dialog box are filled out, as you can see in Figure 318.

5. Specify the data source the account bean will use. Click the DataSource tab and then click **Change**.

6. Select the ACCOUNTS data source you created before and enter the name and password of the database user you want to use for this EJB.
The security for EJBs is handled by WebSphere and not by the database system. All users access the database using the username and password you specify here.

7. To ensure that WebSphere creates a table for the EJB in case it doesn’t already exist, check the Create table box. The DataSource tab of the Create EnterpriseBean dialog box now looks like Figure 319.

![Create EnterpriseBean dialog box (Data Source Tab)](image)

Figure 319. Create EnterpriseBean dialog box (Data Source Tab)

8. Click OK and the account EJB is created.

**E.2.4 Deploy and create the Transfer EJB**

Create the Transfer EJB in the same way as the Account EJB. By default you find it in C:\WebSphere\AppServer\deployableEJBs\Transfer.jar.

Since the Transfer bean is a session bean, you do not need to specify a data source and can leave all fields on the DataSource tab empty.
E.3 Applying security for the banking example

Once you have switched on global security for your WebSphere server you cannot invoke any EJB that is not secured. You even cannot invoke a secured EJB from an unprotected resource since there is no security context to be passed in this case.

If you have not enabled global security for your WebSphere server yet, you can skip this section and continue with E.4, “Testing the banking example application” on page 498.

E.3.1 Create servlets as resources of a Web application

Since it is necessary to call all secured EJBs from secured servlets or JSPs and is it only possible to secure the resources of Web applications, you must first create the servlets you want to use in the banking application as Web resources.

In an environment without security the servlets are called via an invoker servlet. But since this invoker servlet is only used when a specific servlet is called for the first time after the WebSphere server was started, it is not sufficient to activate the security for the invoker servlet.

E.3.1.1 Create the CreateAccount servlet

Follow these steps to create the CreateAccount servlet:

1. In the WebSphere administrative console, expand your host and then the Default Server and the Default Servlet Engine. Right-click the name of your WebSphere sample application.
   
   By default this application is called WSamplesDB2_app. Since we were running a multi-server environment and all application names and virtual host names must be unique in one WebSphere administrative domain, we added our host names to the names of all applications and virtual host, so our application is called WSamplesDB2_app_freja.

2. Select Create -> Servlet, then enter CreateAccount as the name of the servlet. The servlet class is called:

   WebSphereSamples.AccountAndTransfer.CreateAccount

3. Click the Add button to enter the URL that will be used to call the servlet. This URL is hard-coded in some actions of the example application, so you should add the following string to the Web application path:

   servlet/WebSphereSamples.AccountAndTransfer.CreateAccount
4. Click **Ok**; the Create servlet dialog box will look like Figure 320.

![Create Servlet dialog box](image)

*Figure 320. Create Servlet dialog box*

5. Click **OK** to create the Web resource.

**E.3.1.2 Create the TransferFunds servlet**

Create the **TransferFunds** servlet in the same way as the **CreateAccount** servlet before. The class name for this servlet is:

`WebSphereSamples.AccountAndTransfer.TransferFunds`

and the servlet Web path is:

`servlet/WebSphereSamples.AccountAndTransfer.TransferFunds`

**E.3.2 Create an Enterprise Application**

In IBM WebSphere 3.5 the only way to apply security to any resources is to create an Enterprise Application. Do this with the following steps:

1. In the WebSphere administrative console, create an Enterprise Application by selecting **Console -> Tasks -> Create Enterprise**
**Application.** Enter a name for the application as shown in Figure 321 and click **Next>**.

![Application Name](image)

*Figure 321. Creating an Enterprise Application*

2. Select the resources that will be part of the Enterprise Application.
   a. Expand **EnterpriseBeans** and select the **AccountHome** bean, as shown in Figure 322 on page 489.
Appendix E. Installing the IBM WebSphere 3.5 banking example

4. Click Next>; the dialog box should look like Figure 323 on page 490.

Figure 322. Selecting a resource for an Enterprise Application

b. Click the Add button to add the EJB to the Enterprise Application.

c. Select the TransferHome EJB and click Add again.

3. You still need to add the WebSphere samples Web application to the Enterprise Application. Expand Web Applications and select the name of your sample application, by default WSsample_app. Then click Add again.

4. Click Next>; the dialog box should look like Figure 323 on page 490.
5. Click **Finish** and the Enterprise Application for the banking example is created.

**E.3.3 Configure application security**

It is necessary to enable application security for an Enterprise Application to make sure that user name and password are requested when a resource is called by an unauthenticated user. If no application security was applied, the application will raise the error “Authentication failed” instead of displaying the dialog box for user name and password.

Use the following steps to enable application security.

1. Select **Console -> Tasks -> Configure Application Security** in the WebSphere administrative console. Expand **Enterprise Applications** and select the application you want to switch on security for, in our case the one you created in E.3.2, “Create an Enterprise Application” on page 487.

2. Click **Next>**; the dialog box for the application security settings is displayed as shown in Figure 324 on page 491.
Figure 324. Enterprise Application security settings

These settings are copied from the global WebSphere server security settings and normally you do not have to change them.

3. Click Finish to save the application security settings.

E.3.4 Configure resource security

To define security permissions for Web resources you assign their methods to method groups. You then can assign people or groups from your directory to these method groups.

Methods that are not assigned to any method group cannot be called.

E.3.4.1 Configure resource security for the Account EJB
1. In the WebSphere administrative console, select Console -> Tasks -> Configure Resource Security. Expand EnterpriseBeans and select the AccountHome bean as shown in Figure 325 on page 492.
2. As soon as you click **Next**>, you are asked if you want to apply the default groups as displayed in Figure 326.

3. Click **Yes** for every method of the bean a method group is assigned, as shown in Figure 327 on page 493.
Appendix E. Installing the IBM WebSphere 3.5 banking example

4. Click **Finish** to save the method groups.

### E.3.4.2 Configure resource security for the Transfer EJB

Create the method groups for the Transfer EJB by selecting **Console -> Tasks -> Configure Resource security**, and selecting **TransferHome** below **EnterpriseBeans**. Then repeat the same process as you did for the Account bean.

### E.3.4.3 Configure resource security for the CreateAccount servlet

1. Select **Console -> Tasks -> Configure Resource security**. Expand **Virtual Hosts** and the name of the virtual host you are using for the banking example. By default this is default_host.

2. Select the entry for the **CreateAccount** servlet that you created in E.3.1.1, “Create the CreateAccount servlet” on page 486 as shown in Figure 328 on page 494.
3. Click **Next** and answer the question if the default method groups should be used with **Yes**. The following table shows the default method groups for servlets and JSPs.

<table>
<thead>
<tr>
<th>HTTP request</th>
<th>Default method group</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_DELETE</td>
<td>Remove Methods</td>
</tr>
<tr>
<td>HTTP_GET</td>
<td>Read Methods</td>
</tr>
<tr>
<td>HTTP_POST</td>
<td>Read Methods</td>
</tr>
<tr>
<td>HTTP_PUT</td>
<td>Write Methods</td>
</tr>
</tbody>
</table>

4. Click **Finish**; the default method groups will be created for the CreateAccount servlet.

**E.3.4.4 Configure resource security for the TransferFunds servlet**

Create the default method groups for the **TransferFunds** servlet in the same way as for the **CreateAccount** servlet.
E.3.5 Configuring security permissions

Now you assign users or groups to the method groups. Since all six default method groups are used by the two EJBs, you should select users for all of them.

1. In the WebSphere Administrative Console select **Console -> Tasks -> Configure Security Permissions** and expand **Enterprise Applications**.

2. Select the Enterprise Application you created in E.3.2, “Create an Enterprise Application” on page 487, as shown in Figure 329.

![Assign Permissions Wizard](image)

Figure 329. Selecting an Enterprise application to assign people and groups to its method groups

3. After selecting your application, click **Next>**. Select the first method group, as you can see in Figure 330 on page 496, and click **Next>** again.

If you want to specify the same access rights for every method group (as we do in this example), you can select multiple groups by pressing Ctrl and selecting the group name.
4. You can include every user in the method group for this application by selecting **Everyone**; you can grant access to known users by selecting **All Authenticated Users**; or, you can select **Selection** and then add people and groups from your LDAP directory. In this example application you can select **All Authenticated Users**, as shown in Figure 331 on page 497.
Figure 331. Selecting the users for one method group

5. Click **Next** again and then **Finish**. Repeat this until you have specified security for all method groups. The last dialog box should look like Figure 332 in the end.

Figure 332. Authenticated users for the default method groups
E.4 Testing the banking example application

Started the banking example enterprise application by right-clicking it and selecting **Start**. Now you can test the application.

First, create an account. Start your Web browser and call the URL:
http://YOUR_HOSTNAME/WebSphereSamples/AccountAndTransfer/create.html

Fill out the account number and the initial balance and click the **Create** button. Your screen should look like Figure 333.

![Create a new Account](image)

**Figure 333. Create account servlet after it called the account bean**

Note that the message below the table says “Created account” and does not report any error. To be able to test the transfer bean, you should create a second account.

You can test the Transfer servlet and EJB by calling the URL:
http://YOUR_HOSTNAME/WebSphereSamples/AccountAndTransfer/transfer.html

Enter any amount smaller than the balance of the account you are transferring the money from, and the two numbers of the accounts you created before. If you click the **Transfer** button, your screen should look like Figure 334 on page 499 and display no error message.
Figure 334. Transfer funds servlet after it called the Transfer EJB
Domino and WebSphere Together
Appendix F. Using JDBC to access Domino

As an alternative to the Domino classes, you can use JDBC to access Domino from servlets, JSPs or EJBs. But when doing this, be aware that every user of your servlet accesses the Domino databases using the same Notes ID. You should use WebSphere security or build application-based security if you plan to access Domino R5 via JDBC. There is no way of passing the user credentials from the Java program to the Domino server if you use JDBC.

F.1 Installing the JDBC driver for Domino

Lotus provides a JDBC driver for Domino R5. You can download it at: http://www.lotus.com/developers/devbase.nsf/homedata/homejdbc

After you have downloaded the Domino R5 JDBC driver, install it on your WebSphere server. After the installation you find the Java archive JdbcDomino.jar and the DLLs that are used for the JDBC access in the subdirectory lib of the directory you installed the JDBC driver to.

You must add the JdbcDomino.jar file to your classpath or the directory containing the DLLs to your library paths to access Domino data from WebSphere via JDBC.

We added the Java archive to the following class path variable in the admin.config file:

com.ibm.ejs.sm.adminserver.classpath.

Here is what we added:

C:\Lotus/Domino/JdbcSql/lib/JdbcDomino.jar

Note: If you installed a Notes client after you installed Domino on your server, JDBC will use the Notes ID you were using in this client to access Domino. Make sure that this ID is not password-protected because the servlets and JSPs do not respond if they encounter a password-protected ID.

You can try to run some of the simple examples supplied with the JDBC driver for Domino to verify that the JDBC and Domino combination works before adding WebSphere to the equation.

In this section we explain how to write a servlet that reads Domino R5 data via JDBC. Our example will display all person entries in the Domino R5 directory.
F.2 Creating a servlet that uses JDBC to access Domino R5 data

Create a new servlet class as described in 6.3.1, "Creating the BankingServletTemplate class" on page 216. In addition to the servlet classes, import java.sql.*.

Create a variable to store the JDBC connection. This connection will be created when the servlet is called for the first time in a session and is kept until the WebSphere servlet engine is stopped.

Your VA Java screen now should look like Figure 335.

![Figure 335. Definition of the ReadNames class](image)

We now add the init, destroy, and doGet methods to the class.
F.2.1 The init and destroy methods of the servlet

If you create a method called init in a servlet, the servlet engine calls this method when the servlet is started. We will open the JDBC Connection in the init method as displayed in Figure 336.

```java
public void init(ServletConfig objConfig) throws ServletException {
    super.init(objConfig);
    try {
        Class.forName("lotus.jdbc.domino.DominoDriver");
        objCon = DriverManager.getConnection
            ("jdbc:domino:/names.nsf/yourservername");
    } catch (Exception e) {
        System.out.println (e.getMessage());
    }
}
```

Figure 336. The init method of the ReadNames servlet

In this code we connect to the Domino database we want to work with. Replace yourservername by the abbreviated Domino name of your Domino R5 server. We used gefion.

The destroy method is called just before the servlet is stopped. We will use it to close the JDBC connection as you can see in Figure 337.

```java
public void destroy() {
    try {
        objCon.close();
    } catch (Exception e) {
        System.out.println (e.getMessage());
    }
}
```

Figure 337. The destroy method of the ReadNames servlet

Finally, we will create the method that will do the major part of the work.

F.2.2 The doGet method of the servlet

We now create a method called doGet; this method will respond to GET requests to this servlet.
In this method we connect to Domino using JDBC, open the Domino Directory and display the peoples view. The result looks very similar to the result of the *ReadNamesRemote* servlet described in Appendix B, “The ReadNamesRemote Servlet” on page 459.

```java
public void doGet(HttpServletRequest req, HttpServletResponse res) throws javax.servlet.ServletException, java.io.IOException {
    res.setContentType("text/html");
    res.setHeader("Pragma", "no-cache");
    res.setHeader("Cache-control", "no-cache");
    java.io.PrintWriter pw = res.getWriter();
    pw.println("<HTML>");
    pw.println("<HEAD>");
    pw.println("<TITLE>Persons</TITLE>");
    pw.println("<HEAD>");
    pw.println("<BODY>");
    pw.println("<H1 ALIGN="CENTER">Persons in Domino directory</H1>");
    pw.println("<TABLE>");
    try {
        Statement objGetNames = objCon.createStatement();
        ResultSet objResNames = objGetNames.executeQuery
                                 ("SELECT * FROM People");
        String sLine = ""
        while (objResNames.next()) {
            sLine = "<TR>";
            for (int iCol = 1; iCol <= objResNames.getMetaData().getColumnCount(); iCol++) {
                Object objCell = objResNames.getObject(iCol);
                if (objResNames.wasNull()) {
                    sLine = sLine + "<TD> </TD>";
                } else {
                    sLine = sLine + "<TD>" + objCell.toString() + "</TD>";
                }
            }
            sLine = sLine + "</TR>";
            pw.println(sLine);
        }
    } catch (Exception e) {
        pw.println(e.getMessage());
    }
    pw.println("</TABLE></BODY>");
    pw.println("</HTML>");
}
```

*Figure 338. The doGet method of the ReadNames servlet*
Now we export the Java code into the WebSphere servlet directory or the servlet directory of an application and call its URL. Our browser displays the people view of the Domino directory as shown in Figure 339.

![Figure 339. Output from ReadNames servlet](image)

We can use JDBC from JSPs and EJBs in a similar way.
Appendix G. HTML output from viewnav.jsp

For reference purposes, the following listing is an example of the complete HTML generated by viewnav.jsp for the navigation tree shown in Figure 340. The source JSP code is shown and discussed in 8.6.4, “JSP source for viewnav.jsp” on page 348.

Figure 340. navigation tree created by viewnav.jsp

```html
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<link rel="stylesheet" type="text/css"
href="http://freja.lotus.com/intranet/intra_01.nsf/viref_foFiles$ByName/Net
tscapeNavigator6/$File/ns6.css">
<script language="JavaScript1.3">
var NUM_DAYS = 10;

function on_load() {
    // select current division in <select>
    var division = "Communication";
    var divisions = document.forms["frm"].divisionList.options;
    var i;

    for (i = 0; i < divisions.length; i++) {
        if (divisions[i].value == division) {
            divisions[i].selected = true;
        }
    }
}
```

© Copyright IBM Corp. 2001
function setCookie(name, value, expire) {
    document.cookie = name + "=" + value
    + ((expire == null) ? "" : (; expires="" + expire.toGMTString()));
}

function newDivision( slct ) {
    // determine cookie expiration date
    var date = new Date();
    date.setTime(date.getTime() + (86400 * 1000 * NUM_DAYS));

    // get new division
    var division = slct.options[ slct.selectedIndex ].value;

    // set the cookie
    setCookie("IntranetDivision", division, date);

    // refresh menu
    window.location.reload();

    // call the new division's home page in the middle frame
    parent.frames["Middle"].location="http://freja.lotus.com/intranet/intra_01.nsf/viref_foFrameset$ByName/ContentFrameset?OpenDocument";
}
</script>
</head>
<body onload="on_load()">
<form name="frm">
<select name="divisionList" onchange="newDivision(this)"
    <option value='Communication'>Communication</option>
    <option value='General Services'>General Services</option>
    <option value='ISPM'>ISPM</option>
</select>
</form>
<hr>
<div class="head1">
    <nobr>Intranet</nobr>
</div>
<div class="doc2">
    <nobr>
        <a href='http://freja.lotus.com/intranet/intra_01.nsf/fo_AreaContentList?OpenForm&CSTE-4TJBSV'

508  Domino and WebSphere Together
Media

Human Resources

Marketing Communication

External Supplier

Lotus

Software Group

Domino.Server

Media
Appendix H. JSP custom tags that create scripting variables

In all of the sample custom JSP tags presented in Chapter 8, “JavaServer Pages” on page 269, the final result of the tags was to insert values into the HTML. Another useful feature of custom tags is that they can generate objects that are accessible from server-side script. This is shown in the following code fragment:

```xml
<redbook:varTag id="someString" />  
...  
<%= someString %>
...  
<% if (someString.equals("Domino")) { %>
...
```

In this case, the tag handler class must create the object and store it as an attribute in the appropriate scope (page, request, session, or application). Also, we have to create a special Tag Extra Info class which describes the variables created in the tag handler class. The following code sample would support the JSP fragment above:

```java
public class VarTag extends TagSupport {
    private String id;

    public int doStartTag() {
        return Tag.SKIP_BODY;
    }

    public int doEndTag() {
        // create the object to be referred to as a scripting variable
        String s = getString(); // somehow create the String
        pageContext.setAttribute(id, s);

        return Tag.EVAL_PAGE;
    }

    public void setId(String id) {
        this.id = id;
    }
}

public class VarTEI extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData data) {
        VariableInfo vi1 = new VariableInfo(
            data.getAttributeString("id"),
```
H.1 Tag handler class

To create scripting variables in the tag handler class, objects must be stored as attributes at the desired scope level. In the example above, the following method call accomplished this:

```java
pageContext.setAttribute(id, s);
```

This line stores the object referred to by `s` as an attribute at the page scope using the key string `id`. There is another version of the `setAttribute()` method that takes a third argument, which specifies at which scope to store the object. The third argument is an `int`, which can have the values `PageContext.PAGE_SCOPE`, `PageContext.REQUEST_SCOPE`, `PageContext.SESSION_SCOPE` and `PageContext.APPLICATION_SCOPE`. If this argument is omitted, the object is stored at the page scope level by default.

The reason we create and store the object in the `doEndTag()` method will be discussed in the next section.

H.2 Tag Extra Info class

The Tag Extra Info class describes the scripting variables created by a custom tag handler to the JSP processor servlet. Classes of this type should extend the `javax.servlet.jsp.tagext.TagExtraInfo` base class. The method `getVariableInfo()` returns an array of `VariableInfo` objects to the JSP processor. Each of these `VariableInfo` objects describes a scripting variable created by the tag handler class. The constructor for the `javax.servlet.jsp.tagext.VariableInfo` class has the following signature:

```java
VariableInfo(String varName, String className, boolean declare, int scope);
```

The `declare` argument specifies whether the variable should be declared or not and should always be `true` when creating new objects. The `scope` argument does not refer to page, request, session, or application scope. In
In this case, “scope” means visibility in the JSP page. This argument can have the values shown in Table 21.

Table 21. Scope property of the VariableInfo class

<table>
<thead>
<tr>
<th>Value</th>
<th>Visibility in JSP page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VariableInfo.NESTED</td>
<td>Only between start and end tags</td>
</tr>
<tr>
<td>VariableInfo.AT_BEGIN</td>
<td>After start tag</td>
</tr>
<tr>
<td>VariableInfo.AT_END</td>
<td>After end tag</td>
</tr>
</tbody>
</table>

If AT_END is specified, then the variable can only be created in the doEndTag() method, otherwise the variable can be created and referenced in any of the tag handler methods.

H.3 TLD file

In the TLD file where the custom tag that creates a scripting variable is defined, you must declare the tag’s Tag Extra Info class as follows:

```xml
<tag>
  <name>varTag</name>
  <teiclass>com.lotus.redbook.banking.VarTEI</teiclass>
  ...
</tag>
```

H.4 <redbook:dominoUserName> example

We have created a sample custom tag that creates a String containing the current user’s Domino name and makes it available for scripting in the JSP page.

The following is an example of how it would be used:

```xml
<redbook:dominoUserName id="user" form="full" />
<h2>User: <%= user %></h2>
...
<redbook:viewNav database="db.nsf" view="viewname"
  category="<%= user %>">
  ...
  <% if (user.equals("xxx")) { %>
  ...
  <%
```

The form attribute can equal “full,” meaning the tag returns the fully distinguished name, or “common” for the user’s common name.
In our example, the tag handler class creates a Domino session and recycles it once the user’s name has been obtained. This represents a high overhead cost for retrieving this one piece of information. In a full application you would probably choose to store the user’s name as an attribute of the HttpSession.

The Java source code and the TLD file for the `<redbook:dominoUserName>` example have been included in the additional material in 5955jsp-username-taglib.zip. Specifically, this example is comprised of the following files:

- UserName.jar
- UserNameSource.jar
- username.tld

For information on how to get 5955jsp-username-taglib.zip, refer to Appendix K, “Using the additional Web material” on page 521.
Appendix I. Variations of the People view using JSP custom tags

To illustrate how our JSP ViewNavigator tag library allows total separation between data storage in a Domino database and how that data is presented in a browser, we created three JSP page variations of the People view in Domino Directory. Our JSP samples show content from the People view formatted as:

- An HTML table
- An Excel spreadsheet
- A tabbed table created using JavaScript

Note that the output isn’t necessarily very pretty in our samples, but it illustrates how page designers with DHTML and JavaScript skills can display Domino data in any way they want without having any knowledge of Domino programming whatsoever.

In Figure 341 you can see the People view in Domino Directory as we see it when accessing Domino directly from a browser.

![Figure 341. People view accessing Domino from browser](image)

In Figure 342 on page 516 we use our tag library in a JSP to produce an HTML table with the contents of the People view.
The table is admittedly a bit rough, but can easily be shined up with HTML, cascading style sheet, and so on. The source for our JSP follows.

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ taglib uri="/tld/redbook-taglib2.tld" prefix="redbook" %>
<HTML>
<HEAD>
<TITLE>Comparing Apples and Dominos</TITLE>
<LINK REL=STYLE SHEET HREF="JSP-Styles.css" TYPE="text/css">
</HEAD>
<BODY>
<CENTER><H2>Comparing Apples and Dominos</H2>
<% String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
    response.setContentType("application/vnd.ms-excel");
} %>
<TABLE BORDER=1>
<TR><TH>Name</TH> <TH>Phone</TH> <TH>Company</TH> <TH>email</TH>
<redbook:viewNav database="names.nsf" view="People">
<redbook:viewEntry type="document">
<TR><redbook:viewEntryColumn column="1" /></TD><redbook:viewEntryColumn column="2" /></TD><redbook:viewEntryColumn column="3" /></TD><redbook:viewEntryColumn column="4" />
</TR>
</redbook:viewEntry>
</redbook:viewNav>
</TABLE>
</CENTER></BODY></HTML>
```

**Figure 342. People view as HTML table created by JSP custom tags**
We have highlighted the code that uses our custom tag library.

Note that this is exactly the same JSP that allows us to present the contents of the People view as an Excel spreadsheet, as shown in Figure 343.

Figure 343. People view as Excel spreadsheet created by JSP custom tags

The only difference is that we pass the parameter ?format=excel when we invoke the JSP.

Finally, in Figure 344 on page 518 you can see an example where we have taken a subset of the People view (four documents in our case) and shown them in a tabbed table that is created using JavaScript.
All required data has been passed to the browser so the user can switch between the different tabs without the need for any server round-trips.

The JSP for the tabbed table is available for download together with the other sample JSP. See Appendix K, “Using the additional Web material” on page 521 for information about how to get the files.
Appendix J. Domino Collaboration Objects for Java

The upcoming version of the Domino Toolkit for Java/CORBA for Domino R5.0.8 is planned to include new Domino Collaboration Objects for Java. This version of the toolkit was not available at the time of this writing, so the information in this appendix is subject to change.

The Domino Collaboration Objects are high-level abstractions of Lotus Domino services. Each Domino Collaboration Object represents a Domino service, such as login service, mail service, or calendar-entry service. The goal is to make it easier for Web developers to tap into Domino messaging and calendaring by simplifying the necessary back-end classes into a few component JavaBeans. This enables developers who are not familiar with Domino to rapidly integrate Domino services into their Java applications.

Lotus has already released Domino Collaboration Objects for COM Release 1.0, which supports Microsoft COM based applications like VisualBasic. DCO for COM is available at http://www.lotus.com/dco.

The intent of the Domino Collaboration Objects is not to extend Domino object functionality. Rather, the goal is to provide a functional interface to Domino that is intuitive to Java developers and that has few methods and properties. The Java developer who uses these components may avoid the need to learn the Domino objects in detail. None of the Java components will supply any user interface, although some of the COM components do.

Some of the currently planned Java components are:

- CalendarEntry
  - Search existing calendar entries
  - Create new calendar entries
- Mail
  - Compose and send an e-mail
- Login
  - Local via client ID
  - Remote via CORBA

The Domino Collaboration Objects for Java can be used from:

- Local Notes Client
- Remote Applet / Application
- Servlet or JSP
The servlets (and compiled JSPs) can be running under:

- Domino's servlet engine
- WebSphere's servlet engine
- Tomcat's servlet engine
- and probably other servlet engines as well

Again, these objects were not available at the time of writing this book. You can check for the latest version of the Domino Toolkit for Java/CORBA and more information about the Domino Collaboration Objects for Java at:

http://www.lotus.com/developer
Appendix K. Using the additional Web material

Additional Web material is referenced in this redbook and can be found on the IBM Redbooks Web site. The material is described in Table 22.

Table 22. Additional Web material

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5955rdnm.zip</td>
<td>Zipped file containing the ReadNamesRemote.class and the source file ReadNamesRemote.java. This is used in Chapter 4, “WebSphere - Domino security and single sign-on” on page 109 to verify correct SSO setup.</td>
</tr>
<tr>
<td>5955java.zip</td>
<td>Zipped file containing the Java source code (BankingSource.jar) for all servlets plus the JSP tag library, the Domino agent and the RMI server we use in our sample application.</td>
</tr>
<tr>
<td>5955ejb.zip</td>
<td>Zipped file containing the Java source code (PersonalizationSource.jar) for the Personalization EJB.</td>
</tr>
<tr>
<td>5955jsp-dbprops.zip</td>
<td>Zipped file containing the files used in the sample in 8.3, “Using a bean in a JSP to display Domino database properties” on page 280.</td>
</tr>
<tr>
<td>5955jsp-example-taglib.zip</td>
<td>Zipped file containing the files used in the sample in 8.4.2, “Developing custom JSP tags” on page 297.</td>
</tr>
<tr>
<td>5955jsp-sessiontaglib.zip</td>
<td>Zipped file containing the files used to create a JSP tag library version of the ReadNames servlet as discussed in 8.5.6, “Where to create the Domino session” on page 337</td>
</tr>
<tr>
<td>5955deploy.zip</td>
<td>Zipped file containing the code needed to deploy our sample application as described in Chapter 10, “Installation of the banking application” on page 401.</td>
</tr>
<tr>
<td>5955jsp-username-taglib.zip</td>
<td>Zipped file containing the files discussed in Appendix H, “JSP custom tags that create scripting variables” on page 511.</td>
</tr>
<tr>
<td>5955jsp-ppl-view.zip</td>
<td>Zipped file containing sample files discussed in Appendix I, “Variations of the People view using JSP custom tags” on page 515.</td>
</tr>
</tbody>
</table>
K.1 How to get the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG245955

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the Additional materials and open the directory that corresponds with the redbook form number.
Appendix L. Special notices

This publication is intended to help architects and developers to understand how Domino and WebSphere integrate from a technical angle. The information in this publication is not intended as the specification of any programming interfaces that are provided by WebSphere Application Server or the Domino server family. See the PUBLICATIONS section of the IBM Programming Announcement for WebSphere Application Server, and Lotus Domino R5 for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee
that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

<table>
<thead>
<tr>
<th>Term</th>
<th>Trademark Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>e (logo)® @</td>
<td>Redbooks</td>
</tr>
<tr>
<td>IBM ©</td>
<td>Redbooks Logo</td>
</tr>
<tr>
<td>AS/400</td>
<td>AIX</td>
</tr>
<tr>
<td>DB2</td>
<td>CICS</td>
</tr>
<tr>
<td>MQSeries</td>
<td>DB2 Universal Database</td>
</tr>
<tr>
<td>SecureWay</td>
<td>S/390</td>
</tr>
<tr>
<td>WebSphere</td>
<td>VisualAge</td>
</tr>
</tbody>
</table>

The following terms are trademarks of Lotus Development Corporation in the United States and/or other countries:

<table>
<thead>
<tr>
<th>Term</th>
<th>Trademark Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotus ©</td>
<td>Lotus Notes ©</td>
</tr>
<tr>
<td>Lotus Domino ©</td>
<td>LotusScript ©</td>
</tr>
<tr>
<td>Domino Workflow ©</td>
<td>Domino.Doc</td>
</tr>
<tr>
<td>Lotus Smartsuite ©</td>
<td>Lotus QuickPlace</td>
</tr>
<tr>
<td>People Places and Things ©</td>
<td>Lotus Sametime</td>
</tr>
<tr>
<td>SUPER.HUMAN.SOFTWARE</td>
<td></td>
</tr>
</tbody>
</table>

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere., The Power To Manage., Anything. Anywhere., TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjobenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.
PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.
Domino and WebSphere Together
Appendix M. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

M.1 IBM Redbooks

For information on ordering these publications see “How to get IBM Redbooks” on page 531.

- *WebSphere V3.5 Handbook*, SG24-6161
- *Programming with VisualAge for Java Version 3.5*, SG24-5264
- *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754
- *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755
- *Enterprise JavaBeans Development Using VisualAge for Java*, SG24-5429
- *Using VisualAge for Java to Develop Domino Applications*, SG24-5424
- *XML Powered by Domino How to use XML with Lotus Domino*, SG24-6207
- *Performance Considerations for Domino Applications*, SG24-5602
- *Using Domino Workflow*, SG24-5963
- *Customizing QuickPlace*, SG24-6000
- *B2B Collaborative Commerce with Sametime, QuickPlace and WebSphere Commerce Suite*, SG24-6218
- *How about Version 3.5? VisualAge for Java and WebSphere Studio Provide Great New Function*, SG24-6131
- *WebSphere Scalability: WLM and Clustering Using WebSphere Application Server Advanced Edition*, SG24-6153
• Lotus Domino R5 Clustering with IBM e(logo)server xSeries and Netfinity Servers, SG24-5141
• Getting the Most From Your Domino Directory, SG24-5986
• Lotus Notes and Domino R5.0 Security Infrastructure Revealed, SG24-5341
• Connecting Domino to the Enterprise Using Java, SG24-5425
• Lotus Domino R5.0 Enterprise Integration: Architecture and Products, SG24-5593
• IBM WebSphere and VisualAge for Java Database Integration with DB2, Oracle, and SQL Server, SG24-5471
• Lotus Sametime 2.0 Deployment Guide, SG24-6206
• Lotus Sametime Application Development Guide, SG24-5651
• Linux Web Hosting with WebSphere, DB2, and Domino, SG24-6007
• Developing an e-business Application Using Lotus Domino for AS/400, SG24-6052
• Lotus Domino R5 for Linux on IBM Netfinity Servers, SG24-5968
• Lotus Domino R5 for Sun Solaris, SG24-5969
• Lotus Domino for AS/400 R5: Implementation, SG24-5592
• Lotus Domino R5 for IBM RS/6000, SG24-5138
• Lotus Domino for S/390: Running a Large Domino System, SG24-5984

M.2 IBM Redpapers

Redpapers are only available in softcopy format at:
http://ibm.com/redbooks
• Domino Certification Authority and SSL Certificates, REDP0046
• IBM WebSphere Development Tools for AS/400: An Introduction, REDP0503
• Notes and Domino Connectivity - A Collection of Examples, REDP0115
• Lotus QuickPlace for AS/400: Setup and Management Considerations, REDP0045
• WebSphere’s OSE Remote, REDP0040
M.3 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at ibm.com/redbooks for information about all the CD-ROMs offered, updates and formats.

<table>
<thead>
<tr>
<th>CD-ROM Title</th>
<th>Collection Kit Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM System/390 Redbooks Collection</td>
<td>SK2T-2177</td>
</tr>
<tr>
<td>IBM Networking Redbooks Collection</td>
<td>SK2T-6022</td>
</tr>
<tr>
<td>IBM Transaction Processing and Data Management Redbooks Collection</td>
<td>SK2T-8038</td>
</tr>
<tr>
<td>IBM Lotus Redbooks Collection</td>
<td>SK2T-8039</td>
</tr>
<tr>
<td>Tivoli Redbooks Collection</td>
<td>SK2T-8044</td>
</tr>
<tr>
<td>IBM AS/400 Redbooks Collection</td>
<td>SK2T-2849</td>
</tr>
<tr>
<td>IBM Netfinity Hardware and Software Redbooks Collection</td>
<td>SK2T-8046</td>
</tr>
<tr>
<td>IBM RS/6000 Redbooks Collection</td>
<td>SK2T-8043</td>
</tr>
<tr>
<td>IBM Application Development Redbooks Collection</td>
<td>SK2T-8037</td>
</tr>
<tr>
<td>IBM Enterprise Storage and Systems Management Solutions</td>
<td>SK3T-3694</td>
</tr>
</tbody>
</table>

M.4 Other resources

These publications are also relevant as further information sources:

- **WebSphere Application Server Support page** - contains technical notes, troubleshooting help, service packs, E-fixes and more available online at:
  http://www.ibm.com/software/webservers/appserv/support.html

- **Lotus Knowledge Base** - contains Tech Notes and Papers, available online at:
  http://support.lotus.com/

M.5 Referenced Web sites

These Web sites are also relevant as further information sources:

- http://ibm.com/websphere/
  Entry point to information about the IBM WebSphere software platform for e-business

  Lotus’ primary destination for the latest developer information and resources. Contains articles about new and current technologies along
with relevant tips and techniques to help you build dynamic collaborative
e-business applications.

- http://notes.net/

Notes.net from Iris (the developers of Notes and Domino) is a technical
Web site with discussion forums, documentation and the Webzine Iris
Today with many good articles about technical details of Domino.


The IBM developerWorks Web site is designed for software developers,
and features links to a host of developer tools, resources, and programs.

- http://support.lotus.com/

Lotus Support’s Web site, where you can search using keywords or
browse the Lotus Knowledge Base and locate helpful and informative tech
notes and technical papers for the entire Lotus Product family. This source
of information contains the latest technical information updated hourly.
How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** [ibm.com/redbooks](http://ibm.com/redbooks)
  
  Search for, view, download, or order hardcopy/CD-ROM Redbooks from the Redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

  Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**
  
  Send orders by e-mail including information from the IBM Redbooks fax order form to:

  **e-mail address**

  In United States or Canada: pubscan@us.ibm.com
  Outside North America: Contact information is in the “How to Order” section at this site: [http://www.elink.ibmlink.ibm.com/pbl/pbl](http://www.elink.ibmlink.ibm.com/pbl/pbl)

- **Telephone Orders**

  **United States (toll free)**: 1-800-879-2755
  **Canada (toll free)**: 1-800-IBM-4YOU

- **Fax Orders**

  **United States (toll free)**: 1-800-445-9269
  **Canada**: 1-403-267-4455
  **Outside North America**: Fax phone number is in the “How to Order” section at this site: [http://www.elink.ibmlink.ibm.com/pbl/pbl](http://www.elink.ibmlink.ibm.com/pbl/pbl)

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the Redbooks Web site.

---

**IBM Intranet for Employees**

# IBM Redbooks fax order form

Please send me the following:

<table>
<thead>
<tr>
<th>Title</th>
<th>Order Number</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First name | Last name

Company

Address

City | Postal code | Country

Telephone number | Telefax number | VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date | Card issued to | Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.
Index

Numerics
3-tiered application 19

A
Account EJB 361
Account.jar 406
ACL 112, 407
   installation setup 73
Act as part of the operating system
   user right 46
Active Directory
   warning 125
Add feature
   VisualAge for Java
      205
admin.config 106, 449
Administrative domain 10
Administrator’s console 58
Agent
   calling EJB 386
Aliases
   servlets 235
Application security
   configure 161
Authentication
   all authenticated users 168
Authoring database 191

B
Banking example
   Redbook 183
   WebSphere 479
bootstrap.properties 102

C
Cert.id 72
Class path
   in admin.config 106
CLIENT
   Run-as Mode 432
Collaborative Commerce 6
Connection methodologies
   table 81
Content manager 192

Cookie
   LTPA token 119
CSS 349, 352
Custom tags 295

D
Data source 419
DB2
   create database for EJB 479
   installation 48
Default server 149
Default servlet engine 150
default_app 16
default_host
   aliases 62
Development model
   Domino and WebSphere 32
diop_iors.txt 126
Distinguished name
   format LDAP/Domino 174
Domino
   ACL setup 73
   advanced configuration 70
   AgentRunner 201
   agents and servlets 34
   calling EJB from agent 386
   configure OSE remote 93
   configuring DSAPI plug-in 82, 92
   configuring IIOP 123
   configuring LDAP 123
   configuring OSE remote 96
   connecting to WebSphere 76
   distinguished name format 174
   enabling plug-in tracing 102
   enabling SSO 135
   import LTPA keys 140
   install WebSphere components 89
   installation 66
   installing administration client 75
   installing Domino server 66
   installing Web Services 68
   invoking servlets 233
   Java/COM restrictions 125
   JDBC access 502
   login form 146
   manual update of DSAPI plug-in files 95
   parsing URL to send to WebSphere 83
WebSphere FixPak 64
Invoker servlet 234
IOR 126
IP forwarding 80
IPSec 78

J
J2EE
  technologies 27
Java
  source for JSP 272
Java/COM restrictions 125
JavaBeans 277
JavaServer Pages
  overview 20
JDBC
  driver 415
JdbcDomino.jar 501
JNDI 12
JSP 269, 277
  controller servlet 344
  Domino session 358
  enable v1.1 149
  introduction 20
  keep Java source 272
  output from viewnav.jsp 507
  overview 20
  syntax 273
  tag library 511
  temporary servlet 21
  using tag libraries 310
VisualAge for Java 201
WebSphere Studio 366
JSP tag library
  examples 515
  scripting variables 511
  TLD file 513

distinguished name format 174
group filter 466
group ID map 467
realm 142
securing server 471
user filter 464
user ID map 467
user short name 134
WebSphere configuration 129
LdapSearch 123
Lightweight Third Party Authentication (LTPA) 129
Limits
  DB2 user ID 42
  passing data via URL 238
Logging
  enabling plug-in tracing 102
  WebSphere 59
Login
  flow of control 192
Login form
  $$LoginUserForm 146
LTPA
  generate keys 131
  import keys 140
LTPA token 119
  accessing in Java 460
  domain 139
  expiration 129, 140
  login flow 192

M
Method groups 115, 162
  default 116
  rules for assignment 116

N
Nanny process 12
NCSO.jar 106
NCSOW.jar 31
  adding to WebSphere classpath 106
Notes.jar 31

O
OSE remote 77
  configuring for Domino 93
data flow figure 84
INET sockets 94
install on Domino 89
queues refresh interval 97

P
Package
VisualAge for Java 218
PanelBuilder servlet 257
Permissions
WebSphere security 165
Personalization
user-driven vs. rule-driven 185
Personalization database 190
Personalization EJB 373
Personalization.jar 406
Project
VisualAge for Java 213
Published documents database 189

Q
queues.properties 63, 83

R
Realm 128, 142
Recycle session 286
Redbook Banking sample
activating transfer agent 447
AgentTransfer agent 390
CallDominoJsp servlet 344
content manager 192
database architecture 185
Domino design 189
DominoLogin servlet 254
flow of control 192
frameset 184
helper classes 241
installation 401
overview 183
PanelBuilder servlet 187, 257
Personalization EJB 188, 373
SaveQuery servlet 188, 263
Search servlet 187, 261
Web application path 412
redbook-taglib.tld 406
Regen Plugin Co button 63

Resource security
configure 162
Reverse proxy 80
RMI server 387
Role
content manager 192
rules.properties 63, 83, 95
Run-as Mode 432

S
Sample application
ReadNames servlet 147
sas.clients.props 100
Scripting variables
JSP taglib 511
Search servlet 261, 263
Security
Domino 112
Java/COM restrictions 125
method groups 115
shared secret 119
user permissions 117
WebSphere 113
WebSphere application 161
WebSphere global 127
WebSphere permissions 165
WebSphere resources 162
Server.id 72
Servlet redirector 78
thick 80
thin 79
Servlet redirectors
testing with Domino 455
thin setup 449
Servlets
add to Web application 152
aliases in WebSphere 235
and Domino Web agents 34
API 2.2 209
introduction 19
invoking from Domino 233
overview 19
passing data by form 238
passing data by URL 236
ReadNamesRemote source 459
servlet engine transport 459
showCfg 60
snoop 60

536 Domino and WebSphere Together
URLs 233
using JDBC 502
Session authentication
Domino 144
showCfG servlet 60
SOAP 28
SSO 175
enabling in Domino 135
LTPA token 119
verifying 170
what is 109

T
Tag libraries 295
deploy 313
descriptors (TLD) 308
developing 297
Domino session 337
scripting variables 339
using in JSP 310
Thick servlet redirector 80
Thin servlet redirector 79
thinRedirectorStart.bat 454
TLD 308
deploy 317
elements 309
ViewNavigator tags 335
topframe.jsp 406
Transfer Funds agent 447
Transfer.jar 406
Troubleshooting 175

U
UDDI 28
Uniform Resource Identifier 17
Uniform Resource Locator 17
URI 17
URL 17
  parsing by plug-in 83
  servlet 233
User
  WASAdmin 407
User ID
  length limit for DB2 42
User id
  WASAdmin 122
User.id 72

V
vhosts.properties 63, 82, 95
viewnav.jsp 348, 406
ViewNavigator tags 319
  TLD file 335
  viewEntry 326
  viewEntryColumn 330
  viewEntryIndent 333
  viewNav 320
virtual host 16
Web path 150
VisualAge for Java
  create class 216
  Domino AgentRunner 201
  servlet API 2.2 209
  create EJB 368
  deploy EJB 370
  Domino Access Builder 201
  EJB/JSP Development Environment 201
  installation 197

W
Web application
  add servlet 152
  add servlets 424
  application path 15
  classpath 15
  create 422
  creating 148
  default_app 16
  document root 15
  WebSphere 15
Web services
  open standards 27
  technologies 28
Web SSO Configuration 137
WebSphere
  add servlet 152
  adding NCSOW.jar to classpath 106
  admin.config 106, 449
  administrative domain 10
  administrative server 11
  administrator’s console 58
  advanced edition 9
  advanced LDAP properties 463
  application components 18
  banking example 479
challenge type 128
configure application security 161
configuring DSAPI plug-in 82
connecting to Domino 76
connecting to LDAP 476
DataSource 419
default_host aliases 62
e-fixes 66
EJB 382
enable JSP 1.1 149
enterprise application 158
FixPak installation 64
global security 127
installation 53
logs 59
method groups 115
naming services 12
nanny process 12
resource security 162
security 113
security permissions 165
servlet aliasing 235
servlet engine transport 94
servlet redirectors 78
standard edition 9
table with attributes 4
user registry 129
virtual host 16
WAS database 11
Web application 15
WebSphere Studio 366
Working Together 5
WSDL 28

X
XML
from Domino 241
IBM Redbooks review

Your feedback is valued by the Redbook authors. In particular we are interested in situations where a Redbook "made the difference" in a task or problem you encountered. Using one of the following methods, please review the Redbook, addressing value, subject matter, structure, depth and quality as appropriate.

- Use the online Contact us review redbook form found at [ibm.com/redbooks](http://ibm.com/redbooks)
- Fax this form to: USA International Access Code + 1 845 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

<table>
<thead>
<tr>
<th>Document Number</th>
<th>Redbook Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>SG24-5955-01</td>
<td>Domino and WebSphere Together - Second Edition</td>
</tr>
</tbody>
</table>

Review

What other subjects would you like to see IBM Redbooks address?

Please rate your overall satisfaction:

| O Very Good | O Good | O Average | O Poor |

Please identify yourself as belonging to one of the following groups:

| O Customer | O Business Partner | O Solution Developer |
| O IBM, Lotus or Tivoli Employee | O None of the above |

Your email address: The data you provide here may be used to provide you with information from IBM or our business partners about our products, services or activities.

| O Please do not use the information collected here for future marketing or promotional contacts or other communications beyond the scope of this transaction. |

Questions about IBM’s privacy policy? The following link explains how we protect your personal information. [ibm.com/privacy/yourprivacy/](http://ibm.com/privacy/yourprivacy/)
Domino and WebSphere Together
Second Edition

Installation and setup including SSO

Development using servlets, JSPs, EJBs and Domino

Application deployment and security

In this IBM Redbook we explain how you can use the combined capabilities of Lotus Domino and IBM WebSphere to provide a complete and integrated platform for solutions like collaborative commerce.

In the first part of the book we show you how to install and configure Lotus Domino R5.0.6a and IBM WebSphere 3.5.3. This includes setup of Single Sign On (SSO) between the different servers. We describe this for Windows 2000, but you should be able to use our description to guide you through setup on other platforms as well.

In the second part of the book we look at how the WebSphere Java server components—servlets, JavaServer Pages (JSP), and Enterprise JavaBeans (EJB)—can work together with the functionality of Domino in a Web application, from a developer’s viewpoint. We use one common sample application to illustrate how the different technologies can work together. In the last chapter we show how to deploy and secure our sample application. Among the topics we cover in appendices are advanced LDAP properties for WebSphere, securing the LDAP server, and JSP custom Domino tags that support scripting.

All examples are supplied as additional material for download from the IBM Redbooks Web site. These include general ways to handle application login and token expiry, forwarding to JSPs when using a Domino session, and so on.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks