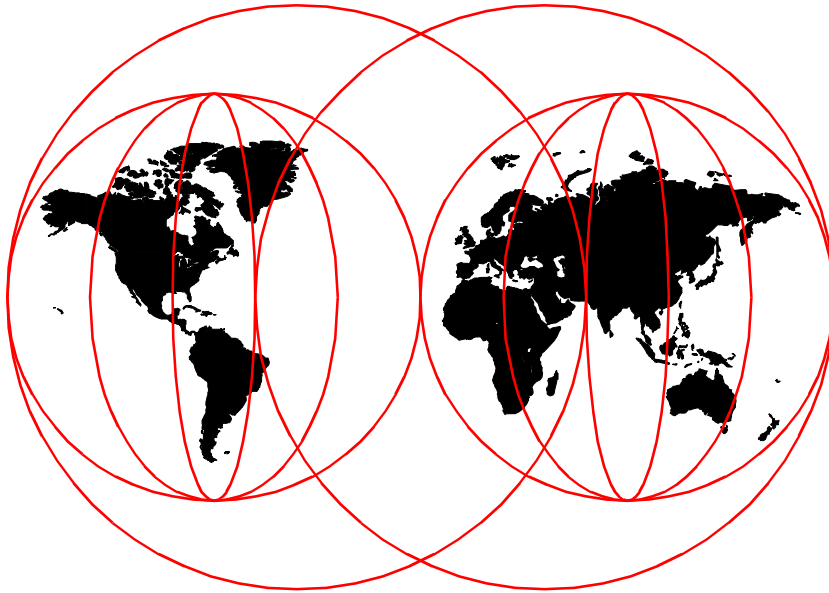


Developing an e-business Application for the IBM WebSphere Application Server

John Akerley, Murtuza Hashim, Alexander Koutsoumbos, Angelo Maffione



International Technical Support Organization

www.redbooks.ibm.com

SG24-5423-00



International Technical Support Organization

**Developing an e-business Application for the
IBM WebSphere Application Server**

September 1999

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix C, "Special Notices" on page 195.

Sample Code on the Internet

The sample code for this redbook is available as sg245423.zip on:

`ftp://www.redbooks.ibm.com/redbooks/SG245423/`

Download sg245423.zip and read the README.TXT file included in the file. Any updates to the book will also be found here.

First Edition (September 1999)

Note

This book is based on a pre-General Announcement version of a product and may not apply when the product becomes generally available. We recommend that you consult the product documentation or follow-on versions of this redbook for more current information.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

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 1999. 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	xiii
What You Should Know	xiii
The Team That Wrote This Redbook	xiv
Product Service and Support	xv
Redbook Code and Updates	xv
Comments Welcome	xvi
Chapter 1. Introduction	1
Chapter 2. A Web Programming Primer	3
2.1 The Web Programming Model	3
2.2 Dynamic Page Generation	4
2.3 Servlets	6
2.3.1 Accessing Servlets	6
2.3.2 The Java Servlet API	7
2.3.3 The Servlet Life Cycle	7
2.4 JavaServer Pages	10
2.4.1 JavaServer Pages Specification	10
2.4.2 JavaServer Pages Elements	10
2.4.3 JavaServer Pages API	15
2.4.4 How JavaServer Pages Work	15
2.5 Maintaining State in Web Applications	16
2.5.1 Web Server Authentication	16
2.5.2 Hidden Form Fields	17
2.5.3 Cookies	17
2.5.4 URL Rewriting	18
2.5.5 Servlet Session Management	18
2.6 Web Security	19
2.6.1 Authentication	19
2.6.2 Confidentiality	20
2.6.3 Integrity	20
2.6.4 Non-repudiation	20
2.7 Caching	21
Chapter 3. Designing the Home Banking Application	23
3.1 Application Requirements	23
3.2 System Requirements	24
3.3 Use Cases	24
3.4 Application Prototype	26
3.5 Analysis Object Model	28
3.6 Subsystem Design	29

3.7	Security Model	31
3.8	HBA Architecture and Design	32
3.8.1	Access to the Business Model	32
3.8.2	Controlling the Interaction Between the Client and Server	34
3.8.3	What Goes into a JavaServer Page?	36
3.9	Error Handling	37
3.10	HBA Subsystems	38
Chapter 4. Tool Usage in the Home Banking Application		47
4.1	The Tool Suite	47
4.2	Design and Analysis Tool: Rational Rose 98 Java Edition	50
4.3	Web Site Prototyping Environment: NetObjects Fusion	50
4.3.1	Prototyping the Site	51
4.4	Web Development Environment: WebSphere Studio	57
4.4.1	Page Designer	60
4.4.2	Importing the Site	62
4.4.3	Restructuring the Site	64
4.4.4	Adding Dynamic Pages to the Site	67
4.4.5	Publishing the Site	71
4.5	Java Development Environment: VisualAge for Java	71
4.5.1	Developing Servlets with VisualAge for Java	72
4.5.2	WebSphere Test Environment	73
4.5.3	JSP Execution Monitor	77
4.6	Application Server: WebSphere Application Server	85
4.6.1	WebSphere Application Server Architecture	86
4.6.2	WebSphere Implementation of JavaServer Pages	87
4.6.3	Managing Your WebSphere Environment	87
Chapter 5. Implementing the Home Banking Application		99
5.1	Implementing the Domain Firewall	99
5.2	Implementing the Business Model	103
5.3	Implementing the Web Application	104
5.3.1	General Implementation Issues	105
5.4	SubSystem Implementation	109
5.5	Application Manager	109
5.5.1	Application Manager Interaction	111
5.5.2	Application Manager Servlets	112
5.6	Login	115
5.6.1	Login Interaction	118
5.6.2	Login Servlets	119
5.6.3	Login JavaServer Pages and HTML Pages	122
5.7	Account Information	122
5.7.1	Account Information Interaction	125

5.7.2 Account Information Servlets	127
5.7.3 Account Information JavaServer Pages	129
5.8 Bill Payment	131
5.8.1 Bill Payment Interaction	132
5.8.2 Bill Payment Servlets	135
5.8.3 Bill Payment JavaServer Pages	143
5.9 Transfer Funds	144
5.9.1 Funds Transfer Interaction	146
5.9.2 Transfer Funds Servlets	148
5.9.3 Transfer Funds JavaServer Pages	149
5.10 Payee	149
5.10.1 Payee Interaction	151
5.10.2 Payee Servlets	154
5.10.3 Payee JavaServer Pages	158
5.11 User	159
5.11.1 User Interaction	160
5.11.2 User Servlets	162
5.11.3 User JavaServer Pages	166
5.12 Utility Classes	166
5.12.1 CacheControl	166
5.12.2 Formatter	167
5.12.3 XMLConfigUtil	168
Chapter 6. Deploying the Home Banking Application	169
6.1 Installing the Servers	169
6.2 Configuring the Servers	169
6.2.1 Configuring the Web Servers	169
6.2.2 Deploying the HBA Application Classes	177
6.2.3 Deploying the HBA Web Site	179
6.2.4 Configuring the WebSphere Application Server	179
Appendix A. HBA Use Cases	185
Appendix B. Working with the HBA Implementation	191
B.1 Deployment	191
B.2 Development	193
Appendix C. Special Notices	195
Appendix D. Related Publications	197
D.1 International Technical Support Organization Publications	197
D.2 Redbooks on CD-ROMs	198
D.3 Other Publications	198
D.4 Product Documentation	200

How to Get ITSO Redbooks	201
IBM Redbook Fax Order Form	202
Glossary	203
List of Abbreviations	211
Index	213
ITSO Redbook Evaluation	217

Figures

1.	Components of a Web Application	3
2.	Servlet Execution Model	8
3.	How JavaServer Pages Work	16
4.	HBA Use Case Model in Rational Rose	25
5.	Main Page of the HBA	26
6.	HBA Login Page	27
7.	HBA Accounts Page	28
8.	Analysis Object Model in Rational Rose	29
9.	HBA Application Flow	30
10.	HBA Security Architecture	31
11.	Separation of the Model from the Application	33
12.	JavaServer Page as Controller	35
13.	Servlet as Controller	36
14.	HBA Application Manager	38
15.	HBA Logout	39
16.	HBA Authentication Sequence	40
17.	HBA Account History	41
18.	HBA Account Balance	41
19.	HBA Bill Payment	42
20.	HBA Payee Setup	42
21.	HBA Add Payee	43
22.	HBA Delete Payee	44
23.	HBA Transfer Funds	45
24.	HBA Change Password	46
25.	HBA Tool Usage	48
26.	Tool Usage with an SCM Tool	49
27.	Tool Usage Life Cycle	50
28.	Site Navigation Bar, or Menu, of the HBA Application	52
29.	NetObjects Fusion Visual Page Editor	53
30.	Extra Links on an HBA Page	54
31.	Fusion Publishing Wizard	55
32.	Fusion Generated Site in Windows NT Explorer	56
33.	Page Designer—Normal View	60
34.	Page Designer—HTML Source View	61
35.	JSP Support in the Page Designer	62
36.	Importing the Prototype Site	63
37.	Relations View of the Imported Site	63
38.	Changing File Extensions	65
39.	Defining Publishing Targets	66
40.	Publish Setup	66
41.	WebSphere Studio Files View after Site Restructure	67

42.	Adding the SERVLET Tag	68
43.	Previewing the Account History Page in the Page Designer	70
44.	Editing the Account History Page in the Page Designer	70
45.	The WebSphere Test Environment	74
46.	Launching the WebSphere Test Environment	75
47.	WebSphere Test Environment Window	76
48.	WebSphere Test Environment Output to Console Window	76
49.	Launching the JSP Execution Monitor	78
50.	Options Dialog for JSP Execution Monitor	78
51.	Loading a JSP for Monitoring	79
52.	The JSP Execution Monitor	80
53.	JSP Syntax Error in the JSP Execution Monitor	82
54.	Stepping Through Syntax Errors in the JSP Execution Monitor	83
55.	JSP Generated Servlets	84
56.	WebSphere Application Server Architecture	86
57.	WebSphere Administration Console	88
58.	WebSphere Application Server Manager Introduction	89
59.	Servlet Configuration under WebSphere	90
60.	Servlet Aliases in WebSphere	91
61.	Servlet Filtering in WebSphere	92
62.	JVM Debug Settings in WebSphere	93
63.	Active Session Monitor in WebSphere	94
64.	Resource Monitor in WebSphere	95
65.	Database Connection Monitor in WebSphere	96
66.	Connection Management in WebSphere	97
67.	Session Management in WebSphere	98
68.	Selected Elements of the Bank Domain Firewall	102
69.	Selected Elements of the Rose Model of the Bank Implementation	104
70.	Complete HBA Implementation	105
71.	Application Manager - User Recognition	109
72.	BankServlet init Method Sequence	111
73.	Session Management JSP/BankServlet Interaction Diagram	112
74.	Login Subsystem	115
75.	Login Screen	116
76.	Accounts Page	117
77.	Unsuccessful Login Page	117
78.	LoginServlet Interaction Diagram	118
79.	Account Information Page	123
80.	Account Balance Page	124
81.	Account History Page	124
82.	Account Information Architecture	125
83.	Account Information Interaction	126
84.	Account Balance and History Interaction	126

85. Pay Bill Page	131
86. Bill Paid Page	132
87. Bill Payment Architecture: Choose Bill Payment	133
88. Bill Payment Architecture: Pay Bill	133
89. Displaying the Pay Bill or Transfer Funds JavaServer Page	134
90. Bill Payment Interaction Diagram	135
91. Transfer Funds Page	145
92. Funds Transferred Page.	146
93. Transfer Funds Architecture: Choose Transfer Funds	147
94. Transfer Funds Architecture: Transfer Funds.	147
95. Payee Setup Page	150
96. Add Payee Page.	150
97. Delete Payee Page.	151
98. Add/Delete Payee Servlet Architecture	152
99. Payee Servlet doGet Interaction.	152
100. PayeeServlet doPost Sequence.	153
101. Change Password Page.	160
102. Change Password Architecture	161
103. Change Password Interaction	162
104. Netscape Administration Server on Windows NT.	171
105. Netscape Enterprise Server (Create Server Menu)	172
106. Web Server Menu	173
107. Setting the Document Root Directory	174
108. Applying the Document Root Directory Changes.	175
109. WebSphere Bank Application Packages	177
110. VisualAge SmartGuide	178
111. WebSphere Administration Page	180
112. Adding bank.jar to the Classpath	181
113. Servlet Configuration Facility	182
114. Add a New Servlet Dialog.	183

X Developing an e-business Application for IBM WebSphere

Tables

1. Domain Firewall and Command Pattern Comparison	33
2. BankServlet Methods	113
3. BankServlet Collaborators	113
4. LoginServlet Methods	119
5. LoginServlet Collaborators	119
6. AccountServlet Methods	127
7. AccountServlet Collaborators	127
8. MoneyTransferServlet Methods	136
9. MoneyTransferServlet Collaborators	136
10. BillPaymentServlet Methods	142
11. BillPaymentServlet Collaborators	142
12. TransferFundsServlet Methods	148
13. TransferFundsServlet Collaborators	148
14. PayeeServlet Methods	154
15. PayeeServlet Collaborators	154
16. ChangePasswordServlet Methods	163
17. ChangePasswordServlet Collaborators	163
18. WebSphere Studio Code Folders	179

Preface

It seems that e-business is one of the most often used terms in the computer industry lately. In this book you will follow along with the process of a small team designing and developing the quintessential e-Business application: Home Banking through the Internet.

The Home Banking Application (HBA) demonstrates the use of IBM e-Business products in the development and deployment of the application. First, we introduce this book in the context of Web development, then provide an overview of the applicable Web technologies. Next, we describe the design of the Home Banking Application and the tools we used to build it. Finally, we show how to implement the application, and how to install and configure the application on several platforms and Web servers.

What You Should Know

You should have a working knowledge of Java and Web technologies, including HTML, browsers and Web servers. Familiarity with VisualAge for Java will also help when reading the “WebSphere Test Environment” on page 73, but is not required. You do not need to have any experience with the WebSphere Application Server or WebSphere Studio, but access to the documentation for these products may be helpful as you read the book.

It will also help if you are familiar with the diagrams used in object modeling, especially those using the Unified Modeling Language (UML).

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 San Jose Center.

John Akerley is a consultant at the IBM International Technical Support Organization, San Jose Center, California where he teaches and writes about Java and Web development tools and techniques. John previously worked on the VisualAge for Java certification team at the IBM Toronto Lab, where he helped create certification programs, taught, wrote, and consulted on VisualAge for Java.

Murtuza Hashim is an e-business IT Specialist in IBM Global Services US. He has four years of experience in the software engineering field. His areas of expertise include object-oriented design and development, Web development and enterprise e-business solutions. Murtuza holds a Masters degree in Software Engineering and a Bachelors degree in Systems Engineering. He is also a Sun Certified Java Programmer.

Alexander Koutsoumbos is a Technical Consultant for IBM Australia. His areas of expertise include object technology, distributed computing, and e-business applications. He has presented on topics ranging from Java applications development to VisualAge for Java at conferences and to IBM customers.

Angelo Maffione is an I/T specialist at the Java Technology Center - IBM Semea Sud - Bari. As employee of IBM Global Services, he is involved in projects for customers dealing with Java-Internet solutions and architectures. Angelo holds a degree with honors in Computer Science from the University of Bari, Italy. Before joining IBM three years ago, Angelo worked for the Computer Science Department at the University of Bari, Italy as a researcher. He received an IBM Outstanding Technical Achievement Award in 1996.

Product Service and Support

IBM WebSphere and VisualAge for Java Service and Support is staffed by knowledgeable developers who handle everything from how-tos to complex technical problems. The most common way of contacting Service and Support is through their Web sites: <http://www.software.ibm.com/webservers> and <http://www.software.ibm.com/ad/vajava>. The sites have links to newsgroups, fixes, announcements, and other information. Check these sites periodically for information.

WebSphere Service and Support monitors several newsgroups:

```
ibm.software.websphere.studio
ibm.software.websphere.http-servers
ibm.software.websphere.application-server
```

VisualAge for Java Service and Support monitors several VisualAge for Java newsgroups:

```
ibm.software.vajava.beans
ibm.software.vajava.enterprise
ibm.software.vajava.ide
ibm.software.vajava.install
ibm.software.vajava.language
ibm.software.vajava.non-technical
```

You can find these newsgroups at:

```
news.software.ibm.com
```

There is also a wealth of good material on the VisualAge Developer Domain site:

```
www.software.ibm.com/vadd
```

IBM employees can also use the internal forums for VisualAge for Java and the WebSphere Application Server:

```
ibm.ibmpc.vajava
ibm.ibmpc.webspher
```

Redbook Code and Updates

The source code described in this book, as well as any updates to the book can be found at:

```
ftp://www.redbooks.ibm.com/redbooks/SG245423/
```

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 “ITSO Redbook Evaluation” on page 217 to the fax number shown on the form.
- Use the online evaluation form found at <http://www.redbooks.ibm.com>
- Send your comments in an Internet note to redbook@us.ibm.com

Chapter 1. Introduction

The explosive growth of the World Wide Web over the last few years continues unabated. The Web has evolved from sites that serve static HTML pages to a global arena for recreation, information, collaboration, and business transactions.

This book will take you through the process of developing an e-business, or Web, application: Internet Banking. The book will show this process using Java technologies: servlets and JavaServer Pages as well as IBM Web development and deployment software: VisualAge for Java, IBM WebSphere Application Server and IBM WebSphere Studio.

This book is not a complete reference for either the tools or technologies used. It documents one team's approach to developing a Web application. Use this book in combination with the resources listed in Appendix D, "Related Publications" on page 197 as you build your own applications.

The remainder of this book is organized as follows:

- Chapter 2, "A Web Programming Primer" provides an overview of the applicable Web technologies.
- Chapter 3, "Designing the Home Banking Application" describes the design of the application.
- Chapter 4, "Tool Usage in the Home Banking Application" describes the tools we used to build the HBA.
- Chapter 5, "Implementing the Home Banking Application" describes the implementation of the application.
- Chapter 6, "Deploying the Home Banking Application" details the steps required to install and configure the application on several platforms and Web servers.

Appendix A, "HBA Use Cases" lists the use cases defined for the HBA.

Appendix B, "Working with the HBA Implementation" explains how to work with the code developed for the Home Banking Application.

The process of building servlet-based systems is maturing. Building systems using JavaServer Pages and servlets is a fairly new way of building Web applications, and there are several design and implementation approaches. The book will try to show you these different approaches with a discussion of the benefits of each. IBM has developed a complete framework for developing e-business applications: The IBM Application Framework for

e-business, commonly referred to as EBAF. Although this book does not discuss EBAF, the approaches and tools used are compatible with the EBAF directions.

The banking scenario used in this book will be documented in several other books and workshops developed by the International Technical Support Organization. The approaches used to develop the HBA in this book are an attempt to create a codebase which can evolve to new technologies, for example, an Enterprise JavaBeans or CORBA based bank implementation.

This book does not attempt to create a complete banking application—it does not cover persistence, concurrency, locking, and the connection with a bank's legacy system. This book also does not address more complex client scenarios (for example, using JavaScript or applets). See Appendix D, "Related Publications" on page 197 and refer to www.redbooks.ibm.com for books on these issues.

When you have finished this book you should have a good idea how to build a basic e-business application.

Chapter 2. A Web Programming Primer

This chapter introduces the e-business and Web development technologies you should be familiar with as you read this book. You should already be familiar with Web concepts such as HTTP, browsers, and Web servers, as well as the Java programming language.

2.1 The Web Programming Model

A Web application is any application that uses Web technologies, including Web browsers, Web servers and Internet protocols. Web applications typically connect to other servers such as database or transaction based systems (Figure 1).

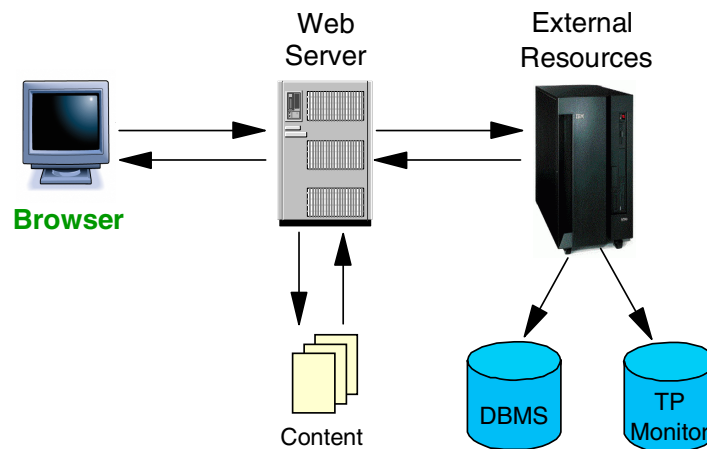


Figure 1. Components of a Web Application

The Web programming model uses a multitier architecture, meaning that applications are partitioned into components. The first tier is the Web browser. The second, or middle, tier includes a Web server (and an optional application server) that assembles Web pages from static and dynamic content and delivers them to clients. In our application, the middle tier logic is implemented in Java, using servlets and JavaServer Pages.

The third tier provides services such as database and transactional capabilities. Typically, these are mature business systems which organizations want to integrate with the Web. In this book we will describe the first two tiers. There are several resources in Appendix D, "Related Publications" on page 197 that discuss the third tier.

2.2 Dynamic Page Generation

Dynamic pages are used to provide Web application output to Web browsers. Dynamic pages are served based on a client request, for example, to view stock prices or trade stocks on the Web. Dynamic pages require Web servers to do more than send the contents of a static HTML page to the browser.

The main technologies supporting dynamic pages are:

Common Gateway Interface (CGI)

CGI is the original means of generating dynamic content for Web servers. In the CGI model a new process is created for each request from the browser. This, while simple to implement and supported by most Web servers, performed poorly because a new process had to be launched for each HTTP request that accessed a CGI program, limiting the number of concurrent requests a server could handle. Additionally, a CGI program cannot interact with the Web server once it has begun execution because it runs in a separate process.

Scripting Languages

Several companies have created server-side scripting environments, including Net.Data from IBM, Active Server Pages (ASP) from MicroSoft, and ColdFusion from Allaire. These technologies are quite popular and allow Web site builders to embed dynamic content as scripts directly into Web pages. The scripts are then interpreted by the server when the page is served. The downside to these technologies is that they are limited to a particular group of products or operating systems and the developer must learn the scripting language.

Server plug-in technologies

There are several plug-in technologies supported by various Web servers. These technologies provide very good performance but are closely coupled to the Web server and can be difficult to program. The plug-in technologies include the Netscape NSAPI and MicroSoft ISAPI.

Servlets

Servlets are the Java solution to dynamic content and are covered in detail in 2.3, "Servlets" on page 6. Servlets have the following features:

- Portability

Servlets are written in Java, making them portable across platforms and across different Web servers, because the Servlet API defines a standard interface between a servlet and a Web server.

- Persistence and Performance

A servlet is loaded once by a Web server, and invoked for each client request. This means that the servlet can maintain system resources (like a database connection) between requests, and there is no overhead of instantiating a new servlet on each request. Servlets can be loaded dynamically or when the Web server is started.

- Java Based

Because servlets are written in Java, they inherit all the benefits of Java, including a strong type system, object-orientation, and modularity. Through garbage collection and the absence of pointer manipulation, servlets avoid many memory management problems that can plague other applications.

JavaServer Pages (JSP)

JSP is a new Java-based scripting technology. JavaServer Pages are described in detail in 2.4, “JavaServer Pages” on page 10. JavaServer Pages have the following features:

- Separation of content presentation and generation

Responsibility for content and data can be delegated to server side components, with JavaServer Pages being responsible for extracting that content and merging it with an HTML document.

- Better Model/View/Controller architecture

JavaServer Pages provide better support for Model/View/Controller (MVC) architecture in a Web application than do servlets. Prior to JavaServer Pages, servlets were responsible for both the control logic and dynamic content generation. This dual role of both controller and view makes the application more difficult to maintain.

- Separation of roles in the development team

Having the business logic encapsulated in components, the control logic handled by servlets, and the dynamic page content handled by JavaServer Pages makes it easier to demarcate roles in a Web team. The JavaServer Page, being a separate file, can be maintained by an HTML author, with a programmer being responsible for the servlets and JavaBeans. The HTML author can interact with the JavaBeans and servlets through tags, much like adding an applet tag in an HTML document.

- Portability and familiarity

By using Java as the scripting language, JavaBeans as the component architecture, and standards like HTML for the presentation, JavaServer Pages are very portable across platforms and Web servers. By using Java

as the programming model and HTML for presentation, JavaServer Pages build on existing skill sets.

- Java based

Because JavaServer Pages are based on Java, they inherit all the benefits of Java, including a strong type system, object-orientation and modularity, and strong memory management.

2.3 Servlets

Servlets are server side Java programs that run inside a Java enabled Web server or application server. Java servlets are to a Web server what Java applets are to Web browsers. Servlets are loaded and executed within a Web server, and applets are loaded and executed within a Web browser.

Servlets are defined by the Java Servlet API, which defines a standard interface between a servlet and a Java enabled server. This makes them portable across these servers.

2.3.1 Accessing Servlets

Servlets are accessed from a Web browser in several ways:

- HTML forms: Servlets are commonly the target of the Submit button in HTML forms. User input is passed to the servlet using the POST or GET methods.
- Hypertext links: Servlets can be the target of a hypertext link in the same way as any other URL. Following the link invokes the service or doGet method of the servlet. Servlets can also be invoked using other requests such as PUT and DELETE.
- SERVLET tag: Some Web servers support the HTML SERVLET tag or support servlets as server side includes using the `<!-- include -->` syntax. The servlet's service or doGet method is invoked and the output is placed in the HTML page, replacing the SERVLET tag. Note that in the JSP 0.92 and 1.0 specifications the only include directive is: `<%@ include file=relativeurlspec>`.
- Other servlets: Servlets can access other loaded servlets using:

```
getServletContext().getServlet("servletname");
```

Note that the Java Servlet API 2.1 deprecates this method and provides the RequestDispatcher interface, which provides methods to forward requests to other servlets and to include output from other servlets.

2.3.2 The Java Servlet API

The Java Servlet API defines a standard interface between a Web server and a servlet. Client requests are made to the Web server, which then invokes the servlet to service the requests through this interface. The API is composed of two packages:

- javax.servlet
- javax.servlet.http

The javax.servlet package contains classes to support generic, protocol-independent servlets. This means that servlets can be used for any protocol that supports a request/response paradigm. Examples of such protocols are FTP, SMTP, and POP. The javax.servlet.http package contains classes to support HTTP servlets. For complete information see the JavaDoc for the Java Servlet API and the resources in Appendix D, “Related Publications” on page 197.

Similar to an applet, a servlet does not have a main method, it has a set of methods, or entry points, which are invoked by the server. A servlet is created from a Java class by implementing the servlet interface. Typically this is done by extending either GenericServlet for protocol-independent servlets, or the HttpServlet class for HTTP-specific servlets.

You may hear the Servlet API described as the JSDK or Java Servlet Development Kit. The JSDK is a reference implementation of the Servlet API. In this book we worked with the Servlet API 2.0 as implemented in the IBM WebSphere Application Server 2.02.

2.3.3 The Servlet Life Cycle

A client of a servlet-based application does not usually communicate directly with a servlet, but requests the servlet through a Web server that invokes the servlet through the Servlet API. The server’s role is to initialize, invoke the service method (or doGet or doPost), and destroy each servlet instance. Typically, there is one instance of each servlet, with multiple threads created to handle multiple client requests (Figure 2). This characteristic makes servlets very efficient.

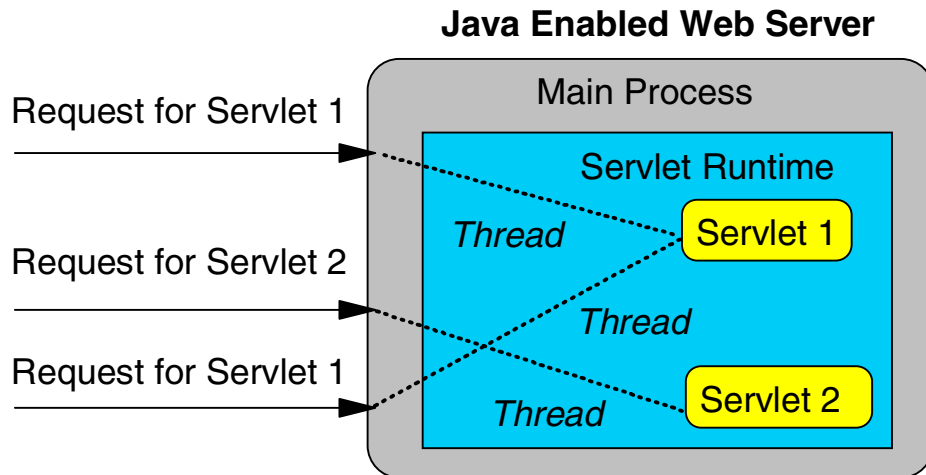


Figure 2. Servlet Execution Model

Servlets can be dynamically loaded when their services are first requested, or the Web server can be configured so that specific servlets are loaded when the Web server initializes.

Once loaded, the Web server communicates with a servlet through the Servlet interface, which defines five methods: `init`, `service`, `destroy`, `getServletConfig` and `getServletInfo`.

init

This method is called when the servlet is first loaded. A subclass of `GenericServlet` or `HttpServlet` only needs to implement this method if it needs to perform setup tasks that should be performed once rather than during each client request. An example of this is initializing a connection to a database or loading default data. The `init` method is guaranteed to be called once, and to complete before any requests are handled.

service

Each time a client request is made, this method is called and it is passed a `ServletResponse` and `ServletRequest` object. The `service` method is responsible for constructing a response for the client request.

A subclass of `HttpServlet` does not implement this method. When the server calls the `HttpServlet` `service` method, it determines whether the request is a GET or POST, and calls the appropriate `doGet` or `doPost` methods that a servlet developer provides implementations for:

doPost

Invoked whenever an HTTP POST request is issued through an HTML form. The parameters associated with the POST request are communicated from the browser to the server as a separate HTTP request. A doPost method should be used whenever modifications on the server will take place.

doGet

Invoked in response to an HTTP GET method from a URL request or an HTML form. An HTTP GET method is the default when a URL is specified in a Web browser. In contrast to the doPost method, doGet should be used when modifications will not be made on the server or when the parameters are not sensitive data. The parameters associated with a GET request are appended to the URL and passed in the HTTP request.

The response from the servlet can be of several types:

- An output stream which the browser interprets based on the content-type, typically an HTML page.
- An HTTP error response.
- A redirection to another URL, servlet, or JavaServer Page.

destroy

The destroy method is called when the Web server unloads the servlet. A subclass of GenericServlet or HttpServlet only needs to implement this method if it needs to perform cleanup operations, such as releasing a database connection or closing files.

getServletConfig

The getServletConfig method returns a ServletConfig instance that can be used to return the initialization parameters and the ServletContext. The ServletContext interface provides information about the servlet's environment, and access to the log.

getServletInfo

The getServletInfo method is an optional method that provides information about the servlet, such as its author, version, and copyright.

The service, doGet and doPost methods are invoked with a request and response object that provide information about the request and the means of communicating the response to the browser. These classes are: javax.servlet.ServletResponse and javax.servlet.ServletRequest for GenericServlets; javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse for HttpServlets.

2.4 JavaServer Pages

JavaServer Pages is a server side scripting technology that allows for dynamic generation of the response on the server. Using JavaServer Pages, you can embed a scripting language inside an HTML page and access business logic through scriptlets or JavaBeans.

A traditional servlet uses an output stream to write HTML code to the Web server for display in a browser. Programmers who write servlet code in Java are, however, not always user interface designers, and may not produce good-looking Web pages. Using JavaServer Pages, you can separate the tasks of programming servlets from that of designing HTML pages. You will see examples of JSP usage in Chapter 5, "Implementing the Home Banking Application" on page 99.

2.4.1 JavaServer Pages Specification

JavaServer Pages is a new technology. At the time of writing, the level 1.0 specification was under review and the current specification was level 0.92; however, not many implementations of this level exist. For example, the IBM WebSphere Application Server supports a modified version of the JSP 0.91 specification, and will move to the JSP 1.0 or higher specification in the future.

This book uses the JSP syntax as defined in the WebSphere Application Server 2.0.

2.4.2 JavaServer Pages Elements

Directives

Directive are placed at the start of a JSP, before any other JSP tags. The general form of a directive is:

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

Here is an example of a directive:

```
<%@ import="java.io.*;java.util.*;itso.bank.model.*" %>
```

Declarations

Use declarations to declare variables and methods for later use in the JSP. The general syntax is:

```
<script runat=server>  
// code for class-wide variables and methods  
</script>
```

For example:

```
<SCRIPT RUNAT=server>
int i=0;
String name="Hello";
private void foo() { ...code... }
</SCRIPT>
```

Scriptlets

Scriptlets consist of Java code that is copied, as-is, into the generated servlet. The general form of a scriptlet is:

```
<% ..... java code ..... %>
```

Scriptlets can also refer to the implicit variables *request* (the servlet request object), *response* (the servlet response object), *out* (the output writer for the generated HTML), and *in* (the servlet input reader). For example:

```
<% out.println("Some <b>bold</b> text"); %>
```

Expressions

An expression is a place holder for a Java variable or expression that is evaluated and the result is placed in the output page. Typically the expression refers to a property or method of a JavaBean, or to a previously defined variable. The general form of an expression is:

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

Tags

JSP tags provide the ultimate separation between your Java code and HTML pages. The tags are used to access properties of JavaBeans. In WebSphere Application Server 2.0, the three tags are BEAN, INSERT and REPEAT. The BEAN tag is part of the JSP 0.91 specification. The INSERT and REPEAT tags are extensions created by IBM for the WebSphere Application Server. In the JSP 0.92 specification, the BEAN tag is changed to USEBEAN and the capabilities provided by the INSERT and REPEAT tags are provided by the DISPLAY and LOOP tags. In the 1.0 specification, the USEBEAN tag becomes jsp:usebean, DISPLAY becomes jsp:getProperty, and the LOOP tag has been removed until a standard extension mechanism can be agreed upon.

Two more promising additions in the 0.92 specification are the ERRORPAGE directive, which provides a consistent way to handle exceptions; and the INCLUDEIF and EXCLUDEIF tags, which provide for conditional display of text using the tags.

At the time of writing, the JSP 1.0 specification was available at <http://java.sun.com/products/jsp> for review.

For a complete reference of the tags and parameters, see the **Create dynamic Web pages**→**Using JSP** section in the WebSphere Application Server online documentation.

BEAN

The BEAN tag creates a reference to a JavaBean to allow subsequent access to the properties and methods of the bean. For example:

```
<BEAN name="customer" type="itso.bank.CustomerView">
</BEAN>
```

This tag defines customer as a reference to an object of the itso.bank.CustomerView type. Once the bean is defined by the BEAN tag, it can be accessed in the JavaServer Page using a scriptlet, expression, or the INSERT tag.

The complete BEAN tag syntax is:

```
<BEAN name="Bean_name" varname="local_Bean_name" type
="class_or_interface_name" introspect="yes|no" beanName="ser_filename"
create="yes|no" scope="request|session|userprofile" > <param
property_name="value"></BEAN>
```

where the attributes are:

- name

This name is used to look up the bean in the appropriate scope (specified by the scope attribute). For example, this might be the session key value with which the Bean is stored. The value is case-sensitive.

- varname

This is the name used to refer to the Bean elsewhere within the JSP file. This attribute is optional. The default value is the value of the name attribute. The value is case-sensitive.

- type

This is the name of the Bean class file. This name is used to declare the Bean instance in the code. The default value is Object. The value is case-sensitive.

- introspect

When the value of this attribute is yes, the JSP processor examines all request properties and calls the set property methods that match the request properties. The default value of this attribute is yes.

- beanName

This is the name of the Bean's .class or the serialized file (.ser file) that contains the Bean. This attribute is used only when the Bean is not present in the specified scope and the create attribute is set to yes. The value is case-sensitive. The path of the file must be specified in the Application Server Java classpath unless the file is in the applicationserver_root\servlets directory.

- create

When the value of this attribute is yes, the JSP processor creates an instance of the Bean if the bean is not found within the specified scope. The default value is yes.

- scope

This indicates lifetime of the Bean. This attribute is optional and the default value is request. The valid values are:

- request - The Bean is set in the request mode by a servlet that invokes the JSP file using the APIs described in the JavaServer Pages API. If the Bean is not part of the request context, the Bean is created and stored in the request context unless the create attribute is set to no.
- session - If the Bean is present in the current session, the Bean is reused. If the Bean is not present, it is created and stored as part of the session if the create attribute is set to yes.
- userprofile - The user profile is retrieved from the servlet request object, cast to the specified type, and introspected. If a type is not specified, the default type is com.ibm.servlet.personalization.userprofile.UserProfile. The create attribute is ignored.

- param

A list of property and value pairs. The properties are automatically set in the Bean using introspection. The properties are set once when the Bean is instantiated.

INSERT

Use the INSERT tag to insert JavaBean properties from a bean in a previously declared BEAN tag into the output page or from request parameters or attributes.

```
<insert requestparm=pvalue requestattr=avalue bean=name  
  property=property_name(optional_index).subproperty_name(optional_index)  
  default=value_when_null>  
</insert>
```

where:

- **requestparm**
The parameter to access within the request object. This attribute is case-sensitive and cannot be used with the Bean and property attributes.
- **requestattr**
The attribute to access within the request object. The attribute would have been set using the setAttribute method. This attribute is case-sensitive and cannot be used with the Bean and property attributes.
- **bean**
The name of the JavaBean declared by a BEAN tag within the JSP file. The value of this attribute is case-sensitive. When the Bean attribute is specified but the property attribute is not specified, the entire Bean is used in the substitution. For example, if the Bean is type String and the property is not specified, the value of the string is substituted.
- **property**
The property of the Bean to access for substitution. The value of the attribute is case-sensitive and is the locale-independent name of the property. This attribute cannot be used with the requestparm and requestattr attributes.
- **default**
An optional string to display when the value of the Bean property is null. If the string contains more than one word, the string must be enclosed within a pair of double quotes (such as "HelpDesk number"). The value of this attribute is case-sensitive. If a value is not specified, an empty string is substituted when the value of the property is null.

REPEAT

The repeat tag retrieves subsequent values in a loop, until an `ArrayOutOfBoundsException` stops the processing.

```
<repeat index=name start=starting_index end=ending_index>  
</repeat>
```

where:

- **index**
An optional name used to identify the index of this repeat block. The value is case-sensitive.
- **start**
An optional starting index value for this repeat block. The default is 0.
- **end**
An optional ending index value for this repeat block. The maximum value is 2,147,483,647. If the value of the end attribute is less than the value of the start attribute, the end attribute is ignored.

2.4.3 JavaServer Pages API

The JavaServer Pages API defines the communication between Java code (typically servlets or other JavaServer Pages) and a JavaServer Page. There are two Java types in the API:

- `com.sun.server.http.HttpServiceResponse`: Extends `sun.servlet.http.HttpResponse` and provides a new method: `callPage` to invoke a JavaServer Page.
- `com.sun.server.http.HttpServiceRequest`: Extends `sun.servlet.http.HttpServletRequest` and provides a new method, `setAttribute`, to set attributes in the request object. These attributes can be accessed in the JavaServer Page using the BEAN tag.

Information can also be passed to the JavaServer Page using the `putValue` method of the `HttpSession` object to associate objects with the session. These objects are accessible for the life of the session, while objects set in the request are only accessible for the life of the request.

2.4.4 How JavaServer Pages Work

The first time a JavaServer Page is invoked (or whenever it is changed) it is parsed into a Java source file containing a servlet, then compiled and initialized. Once the servlet is initialized, the service method is invoked. For all subsequent requests, the service method of the existing servlet is invoked,

and the output of the servlet, the combination of the static and dynamic elements (created through JSP elements) is sent to the browser as shown in Figure 3. A JavaServer Page has an extension of .jsp in order for it to be identified by the server as a JSP file.

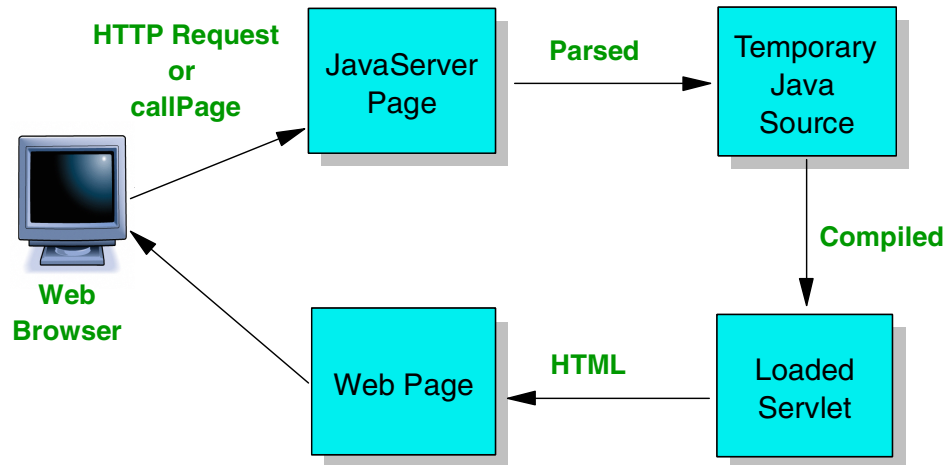


Figure 3. How JavaServer Pages Work

2.5 Maintaining State in Web Applications

HTTP is a stateless protocol, which means that it does not maintain state across client requests. In most cases, HTTP creates a new connection for each request, which means that there is no way for a server to recognize that a series of requests have come from the same client. For many Web applications, the ability to maintain information across client requests is a core requirement.

There are a number of approaches that add state to HTTP, including Web server authentication, hidden form fields, cookies, and URL rewriting. In all these approaches, the Web application must manage the state information. Servlet Session Management manages state at a higher level and supports cookies and URL rewriting through a Session Management API.

2.5.1 Web Server Authentication

The majority of Web servers support user authentication that restricts access to resources to users who have logged in using a username and password maintained by the Web server. In addition to limiting access to resources, authentication can be used to track a client session. When a user has logged

in, the browser retains the username and password and resends them with every request to the same realm. A realm is the combination of a location or resource and the hostname of the server.

This approach is simple to implement and is supported on most Web servers. In addition, as long as the browser is not restarted, it will attempt to use the same username and password any time it is directed at that realm.

The major drawback with this approach is that each user is required to have a unique user ID and password and log on every time they visit the site. They may not appreciate this step. Users expect and appreciate a login procedure when requesting sensitive information, but see it as intrusive and restrictive otherwise. In addition, while all logins from a user are considered a session, they may actually come from different machines.

2.5.2 Hidden Form Fields

Hidden form fields, as their name implies, are fields in an HTML form that are not displayed in the client's browser. They are sent to the server whenever the HTML form containing them is submitted. Hidden form fields can be used to maintain state information by placing a session ID in a hidden form field each time the a response is sent to the client.

This approach is supported in most browsers, does not require any server setup, and does not require a user to be logged in. The major drawback with this approach is that it only works with dynamically generated forms. If your Web site is interspersed with static and dynamically generated pages, this approach may not be appropriate.

2.5.3 Cookies

A cookie is a piece of data passed between a Web server and a Web browser. The Web server sends a cookie that contains data it requires the next time the browser accesses the server. This data may uniquely identify the user to the Web server as well as store other information. When the browser receives the cookie, it stores it and sends it back to the server when requested.

The major drawback with cookies is that not all browsers accept cookies. Typically, a browser rejects cookies because the user does not want cookies. Many users are suspicious of anything being stored on their machine from what they believe may be an untrustworthy source.

2.5.4 URL Rewriting

Another way of tracking user information is to append state information to the hyperlinks in each page sent back to the browser. This technique is known as URL rewriting. When the browser makes a new request of the server, the URL request contains information about the client.

Some disadvantages of URL rewriting are: the user must follow a strict path through the site, they cannot deviate from pages which have the encoded URLs, and the developer must be careful to rewrite all the URLs that are sent back to the client.

2.5.5 Servlet Session Management

All the above approaches address the need to add state to the HTTP protocol, and each approach has its advantages and disadvantages. The Java Servlet API contains types designed to handle session management at a higher level, allowing developers to focus on building Web applications.

With servlet session management, each user can be associated with HttpSession objects that are used to store or retrieve information about that user. This object maintains information about a single session. Other Java objects can be added to a session using the putValue method and retrieved using getValue. A session object can be created and retrieved using the getSession method.

Session management can be implemented using cookies or URL rewriting. A unique identifier for the session object is placed in the cookie or added to the URL. This ID is then used to retrieve the session object.

Session Life Cycle

A session is a series of connections from the same browser over a fixed period of time. A session can be terminated automatically by the server after some fixed time period, or be terminated manually by the Servlet by calling the invalidate method.

For a complete discussion of Session Management, see the Java Servlet API JavaDoc and tutorial and Appendix D, "Related Publications" on page 197.

2.6 Web Security

Security on the Web is a huge topic. As companies decide to provide services to their customers on the Internet, they must take measures to implement security. This section introduces some key security concepts that were applicable to the HBA application.

In order to be resistant to attack, companies must take measures to secure their Web servers, the information that travels between the Web server and the user, and possibly also the users' computers. There are several very different areas to be considered when securing your Web applications:

- Server Location:

Securing the Web server involves securing the physical surroundings of the server and network, the computer on which the Web server runs, and securing the Web server itself.

- Client:

Securing the user's computer involves controlling what software the user is allowed to run as well as the levels of software that they have installed on their computer. The key software to secure is the client or the browser that the user uses to access the Internet. Securing the client completely is only possible in an Intranet environment, although a level of security can be enforced in an Internet environment, for example, by ensuring that clients support SSL. Securing the client may also involve running anti-virus software on the client machine.

- Application

Securing the Web application is the main focus of this section. The main security issues for Web applications include knowing:

- Who you are communicating with (authentication)
- That the transaction is private (confidentiality)
- That the data has not been tampered with (integrity)
- That the participants will not later deny the transaction (non-repudiation)

These four items will be discussed in the rest of this section.

2.6.1 Authentication

Authentication is the identification of the client or server. In a Web application, authentication can be handled by the Web server or by the application itself. Most Web servers provide authentication and access control to limit access to

known users. However, many Web applications are gateways to existing business applications. In these cases, the authentication information may already exist in another system, and authentication can be handled by the Web application itself, or by integrating the Web server's authentication with the existing system.

Authentication can be based on a user ID and one or more passwords, or on digital certificates (see "Digital Certificates" on page 21).

2.6.2 Confidentiality

Confidentiality is provided by encryption. Secure Sockets Layer (SSL) is the most widely used technology for encryption on the Web.

SSL is a layer that sits between the TCP/IP protocol and the application layer. While the standard TCP/IP protocol simply sends a stream of data between two computers, SSL adds numerous features to that data, such as encryption of the data using a variety of algorithms, authentication, and non-repudiation of the server (using digital signatures), authentication and non-repudiation of the client (using digital signatures), and data integrity (through the use of message authentication codes).

When two programs talk to each other using SSL, these programs use the strongest cryptographic protocol that they have in common. These protocols include the Data Encryption Standard (DES) and other symmetric protocols. SSL allows for authentication of both the client and the server through digital certificates and digitally signed challenges.

There are two levels of SSL security based on the length of encryption keys: export and domestic grade. Export grade is used internationally, domestic grade can only be used in North America.

2.6.3 Integrity

Both sides of a transaction must be sure that the information they receive has not been tampered with in any way. This integrity is also provided by the SSL protocol through message authentication codes (MAC). Each SSL transmission has a MAC appended that ensures that the transmission has not been tampered with.

2.6.4 Non-repudiation

It is important that both sides of a transaction agree that the transaction has taken place. Non-repudiation is provided by digital certificates which prove that the certificate holder was involved in the transaction.

Digital Certificates

Digital certificates are a mechanism for authenticating and securing the information that is transmitted between two entities. They consist of a private key and a public key. The private key is used to encrypt a transmission and is only held by the signer. The public key is used to decrypt the transmission and verify the signature. Only the public key can decrypt a transmission signed with the private key and vice-versa.

Client-Side and Server Side Digital Certificates

A client certificate's purpose is to verify the identity of an individual and can eliminate the need to remember usernames and passwords. Client certificates can be used with SSL, but are not mandatory. The downside to client certificates is that each user must obtain one and always have it when accessing the Web application.

A server certificate must be implemented by a Web server that implements SSL. When a browser connects to the Web server using the SSL protocol, the server sends the browser its public key in a certificate. The certificate is used to authenticate the identity of the server and to distribute the server's public key, which is used to encrypt the initial information that is sent to the server by the client.

2.7 Caching

In Web applications, a cache is a place where things are stored temporarily so that data does not have to be retrieved each time from the original source. There are several places where data can be cached in a Web application:

- **Browser:** Most Web browsers maintain a cache of the pages that have been accessed. These cached pages are stored on the client machine's hard drive and are not usually refreshed when the browser is restarted. A user can control this caching through browser settings and by manually reloading pages.
- **Proxy Server:** A Proxy server can be used in organizations wishing to make access to the Web more efficient for their users. Pages are cached by the Proxy server, and users from the organization will receive the cached page rather than the page from the actual URL.
- **Web application:** A Web application can perform its own caching. For example, an application might cache values rather than access a database on each request.
- **Legacy Application:** Many applications, especially databases, maintain very sophisticated caching systems designed to increase performance.

Caching can have a positive effect on the performance of a Web application, but can also complicate the design. Some dynamic information has a definite lifetime, and a cached version may not be not valid. Other information is valid only for the time when it is originally created.

Design the caching strategies for the pages in your Web application when you design the application. Do not cache pages when the data is not valid for a length of time, or else set realistic timeouts for the data on the page.

Preventing Web Page Caching:

To prevent Web browsers and Proxy servers from caching dynamically generated Web pages (meaning dynamic output that results from processing JSP files, SHTML files, and servlets), use the following code to set headers in the HTTP Response (in the JavaServer Page or the servlet):

```
response.setHeader("Pragma", "No-cache");  
response.setDateHeader("Expires", 0);  
response.setHeader("Cache-Control", "no-cache");
```

Setting the HTTP headers is a more effective method of controlling browser caching than using the <META> tag equivalents. For example, <META HTTP-EQUIV="Pragma" CONTENT="No-cache"> is the equivalent of the first HTTP header setting. Setting the HTTP headers is the recommended method, because some browsers do not treat the <META> tags in the same way as the equivalent HTTP header settings. On some browsers, the <META> tag equivalents do not work when the callPage method is used to load a JSP file that contains the <META> tags.

There may be instances when you want to permit a page to be cached, but you do not want a proxy server to permit multiple users to have access to the cached page. For example, suppose your servlet does not use session tracking and it generates a Web page that contains user input. To maintain that personalization, you would want to prevent other users from having access to the cached page. To prevent a proxy server from sharing cached pages, use the following code:

```
response.setHeader("Cache-Control", "private");
```

This header can be combined with the three previous headers for preventing caching.

Chapter 3. Designing the Home Banking Application

In this chapter we will explain how we designed the HBA, including:

- Application Requirements
- System Requirements
- Use Cases
- Application Prototype
- Analysis Object Model
- Subsystem Design
- Subsystem Design
- Security Model
- HBA Architecture and Design
- Error Handling

Finally, we will introduce each subsystem of the HBA in 3.10, “HBA Subsystems” on page 38.

3.1 Application Requirements

Customers using the HBA must be able to:

- Access their checking or savings account balances
- Access their checking or savings account histories
- Transfer funds between checking and savings accounts
- Pay bills from their checking and savings accounts
- Manage the list of payees to which they can pay bills
- Change their password

Customers must be able to do this securely and from any Internet-connected computer. We also decided that we would require customers to authenticate a second time before performing a transaction that modified an account balance.

3.2 System Requirements

The initial system requirements, based on the application requirements, are:

HBA customers need:

- A user ID provided by the bank
- A login password or PIN (personal identification number) and a transaction password
- At least one active bank account
- An Internet browser that supports the Secure Sockets Layer (SSL)
- Access to the Internet

A provider of the services implemented by the HBA needs:

- A Web and application server
- A digital certificate recognized by the client browsers

3.3 Use Cases

A set of Use Cases was created for the HBA. The complete Use Case model is shown in Figure 4. The individual Use Cases were used as inputs to the object model and architecture design. The individual Use Cases are in Appendix A, “HBA Use Cases” on page 185. The Use Case model is very important in a Web application. It can map very closely to the Web pages that will make up the site and can serve as a storyboard to walk through the site. The Use Case model can also drive the design of the application as described in 3.8, “HBA Architecture and Design” on page 32.

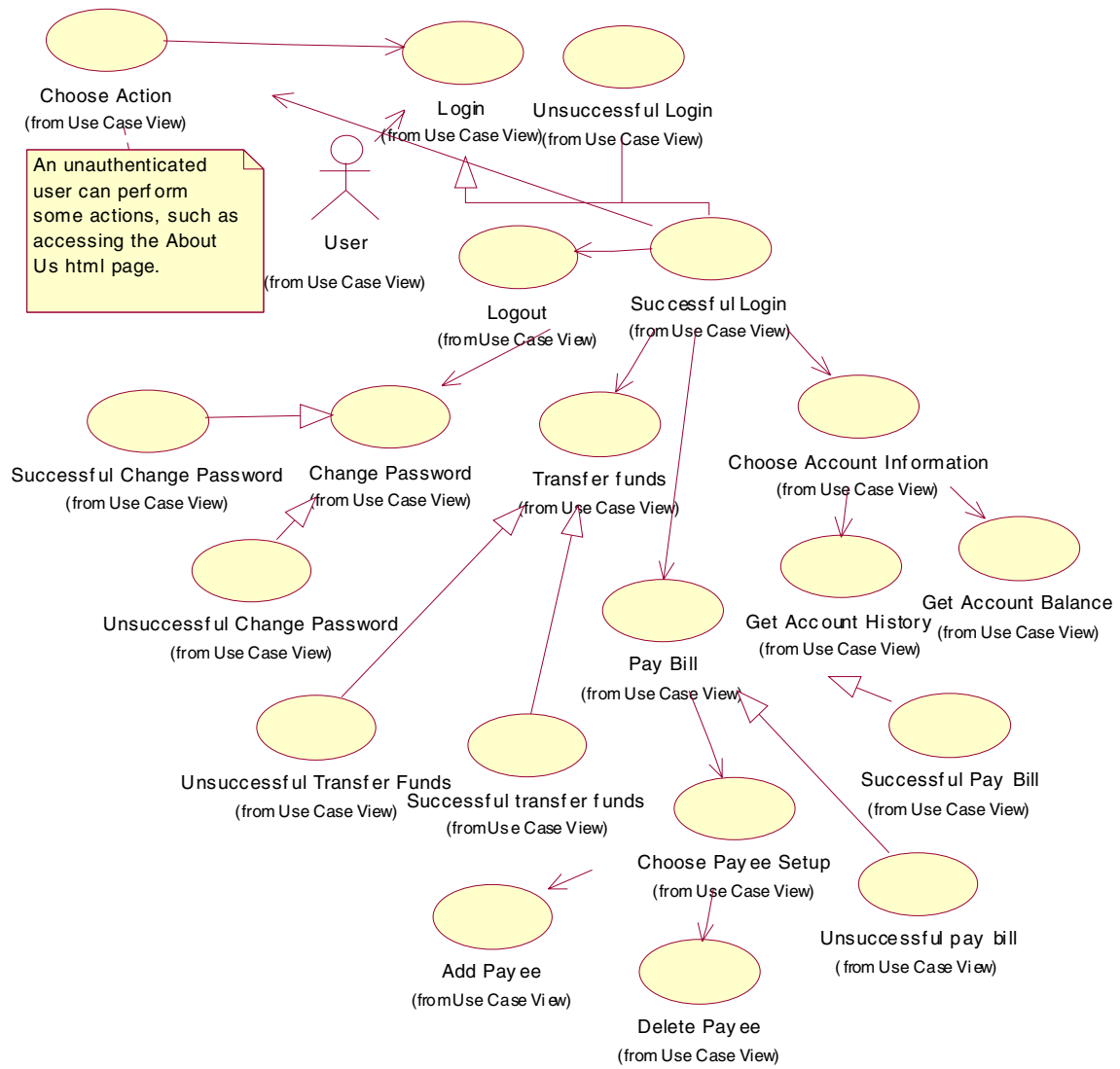


Figure 4. HBA Use Case Model in Rational Rose

3.4 Application Prototype

The prototype is used to explore the look and feel of the HBA. It was created quickly using NetObjects Fusion. Using NetObjects Fusion for prototyping the look and feel of the site can be very productive because you can use the predefined templates and site styles to quickly show different ideas.

A user starts interacting with the HBA by entering the `http://hostname` into their browser, and they are then presented with the main page of the HBA (Figure 5).



Figure 5. Main Page of the HBA

In order to access their accounts, the user must go to the Login page to identify themselves by means of their UserId and Password (Figure 6).

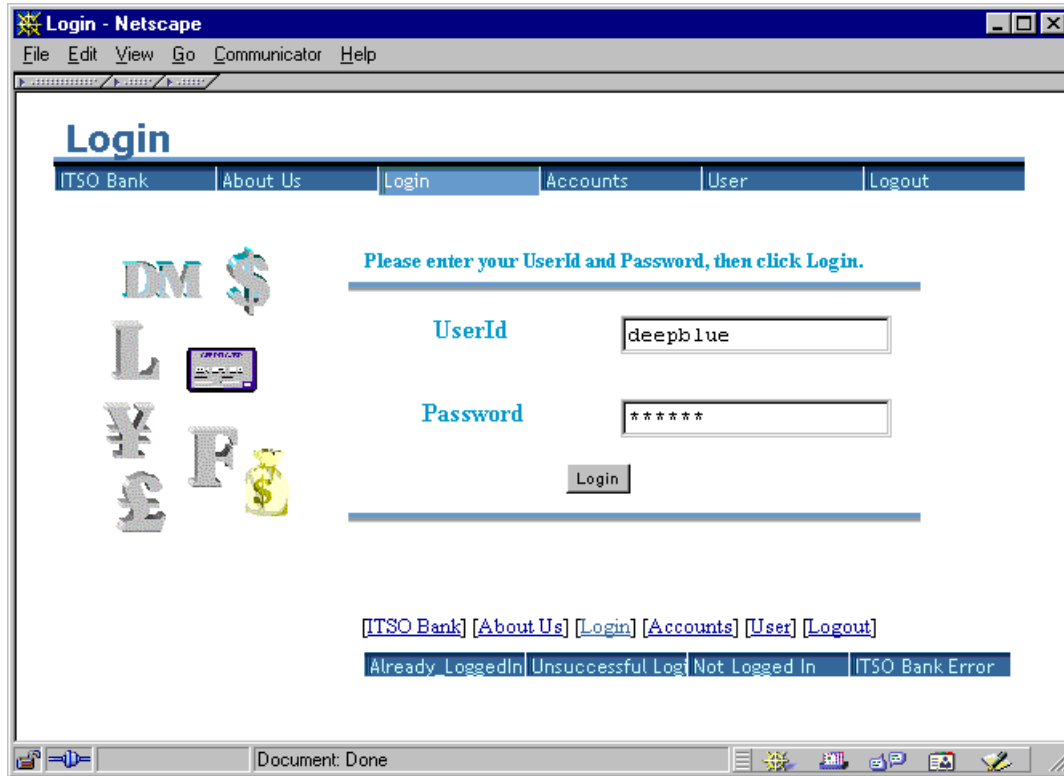


Figure 6. HBA Login Page

Once they have logged in, they have full access to their accounts (Figure 7).

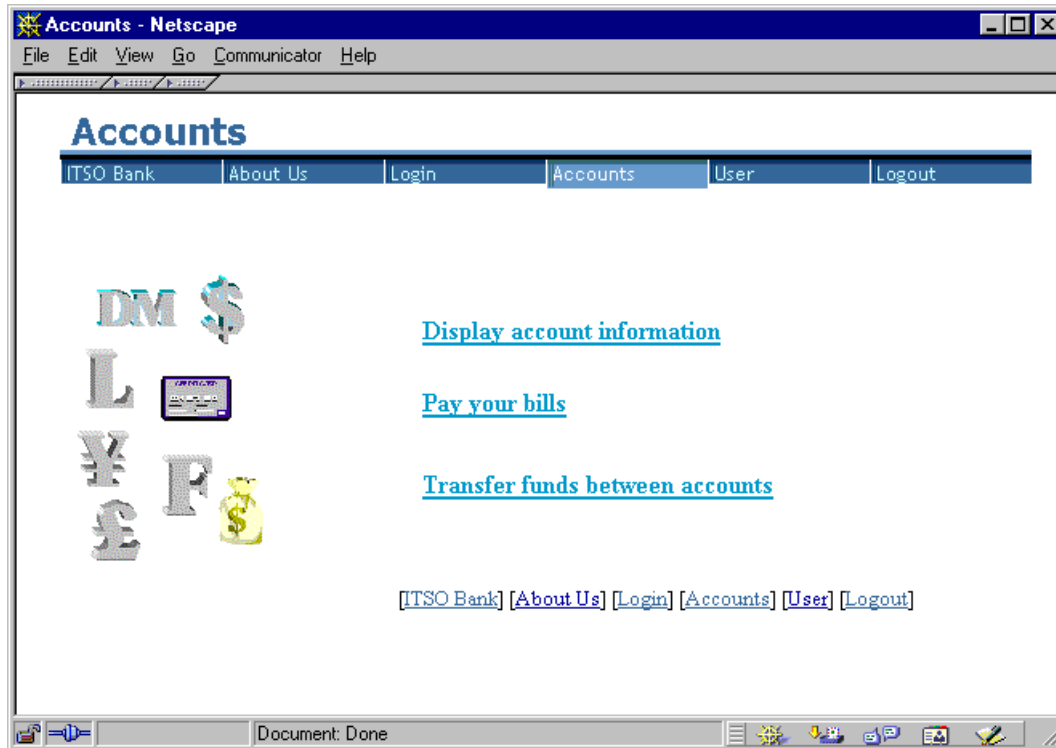


Figure 7. HBA Accounts Page

3.5 Analysis Object Model

In many Web applications the analysis model would be based on the model of an existing legacy application. Especially in the case of an Internet banking system, it is unlikely the system would not be built on an existing bank's infrastructure. Our application does not connect to an existing bank system, and this book does not address the issues involved in connecting to a legacy application. See Appendix D, "Related Publications" on page 197 for these resources.

Because we did not start with an existing application, we created an analysis model of a bank's business model. The object model contains four main objects: Bank, BankAccount, Customer, and TransactionRecord (Figure 8). A Bank has many BankAccounts that may be of three types: CheckingAccount, Savings Account, and PayeeAccount. A PayeeAccount is used to have a target for bill payments by a customer. Each BankAccount may be the target of many transactions, so it may have many TransactionRecords. Each

BankAccount has an owner: a Customer. The PayeeAccount is used to refer to a corporation that the customer pays bills to. For example, the gas company could be a payee.

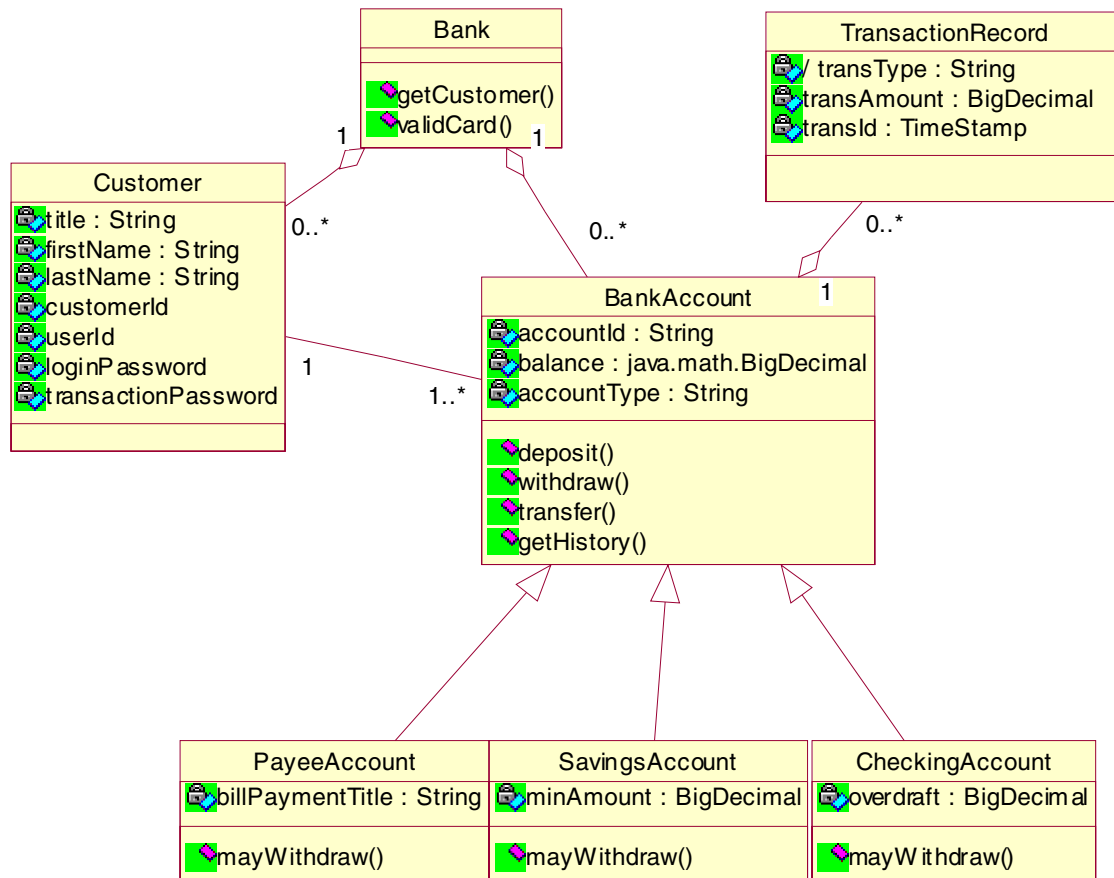


Figure 8. Analysis Object Model in Rational Rose

3.6 Subsystem Design

We have designed eight subsystems in our HBA application:

- Application Manager: Initializes the bank and provides some session management and logout functionality.
- Login: Authenticates the user.
- Account Information: Provides account balance and history.

- Pay Bill: Enables the user to pay bills to corporations (payees).
- Payee Setup: Manages the user's list of payees.
- Transfer Funds: Enables the user to transfer money between their various bank accounts.
- User: Supports password management.

Figure 9 shows the proposed flow between servlets, HTML pages and JavaServer Pages in the HBA. It is very similar to the Use Case model shown in Figure 4 on page 25.

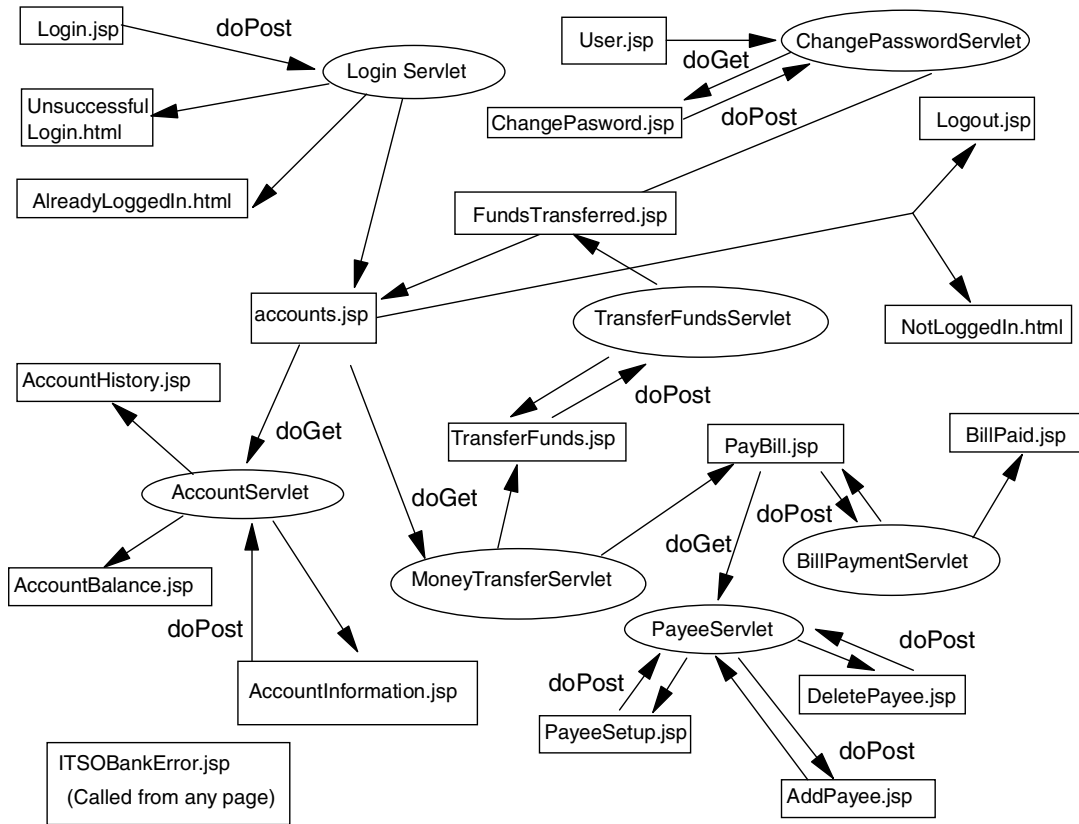


Figure 9. HBA Application Flow

The subsystems are discussed in 3.10, “HBA Subsystems” on page 38.

3.7 Security Model

The security design (Figure 10 on page 31) for the HBA is as follows:

- Access to the Web server will be controlled through passwords and a secure environment.
- The server will only serve application pages using the SSL protocol. This way, the application does not need to worry about the encryption of the transmission. Some e-business applications may require a specific level of security that the application should check, for example, the use of 128 bit keys in SSL communications.
- Users will be required to log in to the HBA and to provide a further password when initiating a transaction that modifies the balance of any account.

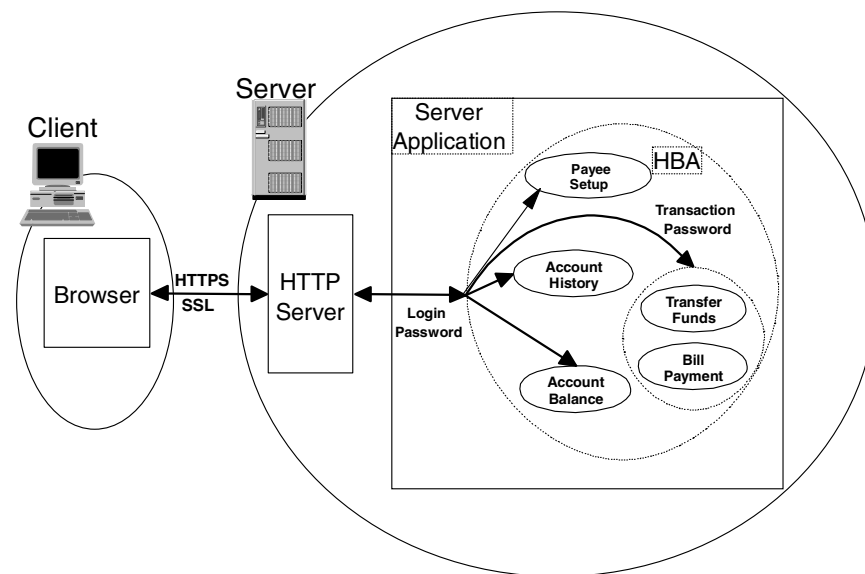


Figure 10. HBA Security Architecture

Security Used in Internet Banking Applications

An informal investigation of some of the worldwide Internet banking applications on the Web showed us that at the moment, about 20% use a single userid and password combined with a server certificate and SSL for authentication, 40% use client and server certificates, while the remaining 40% use SSL and a server certificate with a second level of authentication (similar to our transaction password).

3.8 HBA Architecture and Design

Now we have a start on the design of our application. We know the page structure and flow from the Use Case model, we have a business object model, and we have outlined the major subsystems of the site.

In the HBA design process there were several points at which design decisions must be made. The basic architecture of the HBA is defined by the scope of the project: to build a Home Banking Application using IBM tools that runs on the IBM WebSphere Application Server. Three of the major design decisions we faced were:

1. Access to the Business Model
2. Controlling the Interaction Between the Client and Server
3. What Goes into a JavaServer Page?

3.8.1 Access to the Business Model

Whether you are connecting to an existing legacy application or to a new business model, you need to design the access to the business model. In our case, we decided to create a business model layer, or domain firewall, to separate our Web application from the business model implementation.

The domain firewall is an API that abstracts the object model for the client. It is implemented in 5.1, "Implementing the Domain Firewall" on page 99 using Java interfaces. For example, in our domain firewall we have account objects and they respond to messages that a bank account understands, like `getBalance`.

All interaction with the business logic is channeled through the firewall, and the client application is isolated from any changes in the implementation of the business logic. The separation is shown in Figure 11.

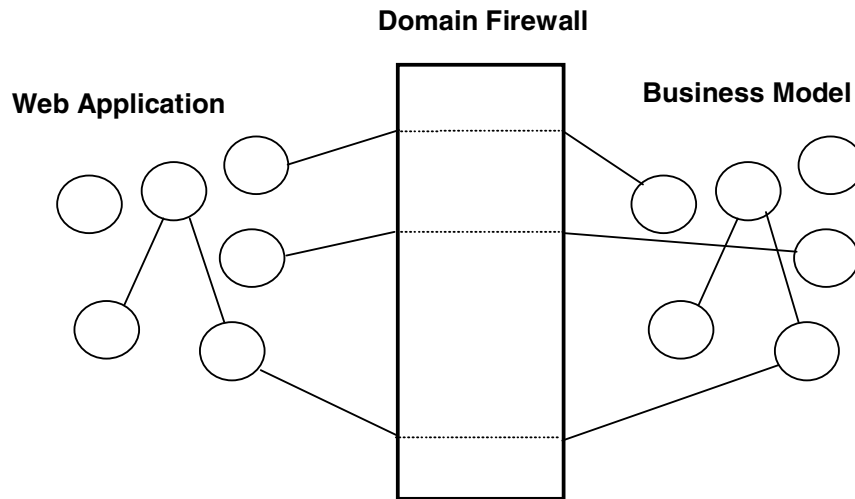


Figure 11. Separation of the Model from the Application

Another option is to use a command pattern. The Command Pattern is defined in *Design Patterns* (see Appendix D, “Related Publications” on page 197). The Command Pattern is used to package and execute commands, and is very effective when you want the commands to be executed uniformly but do not know where or how they will necessarily be executed.

In designing a system to use a command pattern, the Use Cases can map roughly to the commands. For each Use Case, you create a command which is executed in some sort of command handler. Table 1 compares the two approaches.

Table 1. Domain Firewall and Command Pattern Comparison

	Domain Firewall	Command Pattern
Pros	Object-oriented interface Understandable domain	One point of entry Easy to implement logging and undo action Matches well with transactional systems
Cons	May be complex to implement Many points of entry	Not an intuitive interface Lose object-oriented interface

3.8.2 Controlling the Interaction Between the Client and Server

All applications need a point of control to manage the flow of the application. In object-oriented programs the Model/View/Controller (MVC) paradigm has become popular. In MVC the different parts of the application are separated into:

- Model: The business logic of the application.
- View: The user interface.
- Controller: The component that manages the interaction between the model and the view and the flow of the application. In the original MVC designs, the controller managed the user input, but in many versions of MVC, it has become the controller of the flow of the application.

In servlet-only applications, the servlet was used as both the controller and the view (HTML was coded in Java or read from a file), but this has disadvantages. If the servlet is used to generate dynamic content, then any changes made to the format of the output also requires that the Servlet be recompiled. This makes the application more difficult to maintain, particularly if there are frequent changes to the format of the output.

In a JSP/servlet based Web application, the controller could be implemented directly in JavaServer Pages, or in servlets that then invoke JavaServer Pages.

If JavaServer Pages are used as the controller, browsers make requests directly to a JavaServer Page (Figure 12). After receiving the client request, the compiled JSP servlet requests information from server components, which perform any necessary computation and encapsulate the business logic. Then, the compiled JSP servlet inserts the results of the computation into the Web page, which is then rendered and interpreted by the browser as usual. In this case, controller code and view code are mixed in the same component. This may make maintenance of the application more difficult. In addition, the tools currently available for Java development do not have strong support for JSP/HTML development and vice-versa.

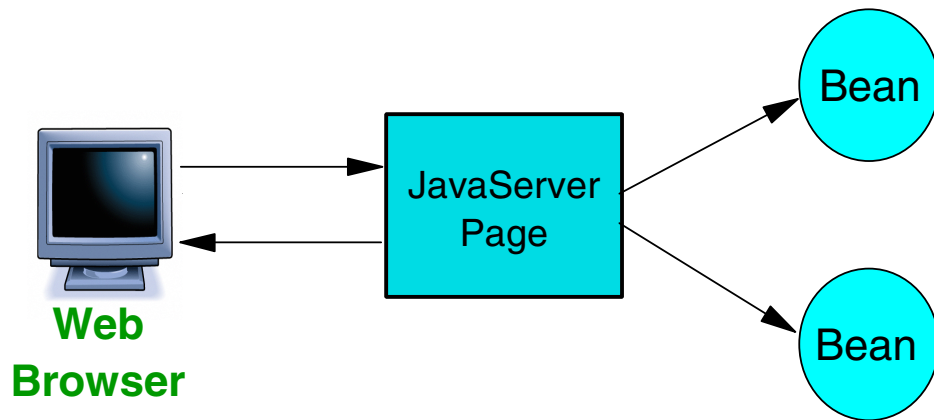


Figure 12. JavaServer Page as Controller

If servlets are used as the controller (Figure 13), browsers invoke servlets which then invoke JavaServer Pages. The Java Server pages would access only the information needed to display results. The Servlet interacts with the JavaBeans to perform any necessary computation and encapsulate the business logic, and may also create beans to store the results of the computation. The JSP then extracts whatever information it requires from the JavaBeans and merges them with the Web page. The browser then interprets and renders the Web page as usual.

Although using servlets as the controller adds another layer to the application beyond JavaServer Pages, we think it is a good idea, because:

- View and controller code are logically separated.
- You can use the best tools for Java and for HTML/JSP without trying to combine them.
- Your Java developers and HTML developers are not working on the same code.

We used a servlet as the controller except in the few cases where the JavaServer Page had no request data associated with it.

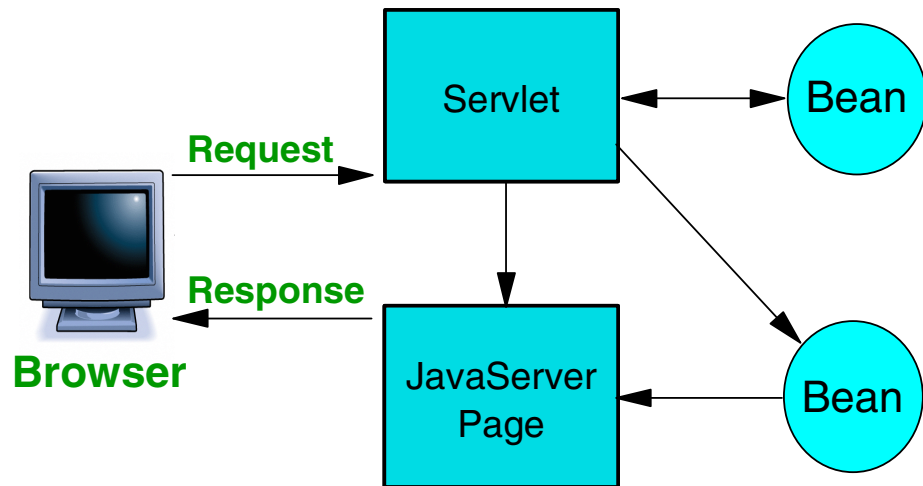


Figure 13. Servlet as Controller

3.8.3 What Goes into a JavaServer Page?

Now you know that we think the servlet should act as the controller. You still need to decide exactly what goes into a JavaServer Page. A JavaServer Page can contain Java code or bean tags to represent dynamic content.

In the first approach, the Java code is embedded directly inside the HTML document using scriptlets. We believe you should use this approach when the programming logic is relatively minor. Having any more than a trivial amount of programming logic inside the HTML document can make it more difficult to separate roles in a development team, as the HTML and Java code are tightly coupled. If scriptlets are used, business logic should still be encapsulated in JavaBean components.

In the second approach, the programming logic resides in components and bean tags are used to request information from these components, which is then inserted into the HTML document. This approach has an advantage in that it facilitates a cleaner separation of roles in a development team, and the programming logic is contained inside components which would be the responsibility of a Java programmer.

In this approach we recommend that you go a step further and create special view beans to be used in JavaServer Pages. These beans are created by the servlet and placed in the session or the request. They simply encapsulate data based on the results of the request. You may also see similar beans called adaptor beans in other documents.

As with using servlets as controllers, using View beans will add another layer of code to your application, but it will make the code cleaner and easier to maintain. You will find examples of view beans in 5.3, “Implementing the Web Application” on page 104. This separation will also make your application easier to test because you can unit test each subsystem using dummy data in the view beans.

An additional current reason to use view beans is that, using the current implementation, the type of a bean in a JSP page cannot be declared as an interface, and any methods which throw checked exceptions must be caught in the JavaServer Page.

3.9 Error Handling

Error handling in the HBA application is handled in several ways:

- User Errors

If a user enters an incorrect value, an appropriate message is displayed either on a separate page or the page on which the entry was made.

- Application Errors

If the application detects an error, for example, the bank is not available, the `callErrorPage` method is called that sends the user to a generic error page and lists the error.

In the JSP 0.92 and 1.0 specifications, the `errorpage` directive is introduced. This can be used as a generic way of handling runtime exceptions in JSP pages. In the 0.91 specification there is no general error handling mechanism, but the XML servlet configuration file (a feature of WebSphere) allows an error page to be defined. The XML servlet configuration file is discussed in “Servlets” on page 105.

In both specifications, JSP pages containing elements that can throw checked exceptions must catch these exceptions. This is another reason to create the View layer (see 3.8.3, “What Goes into a JavaServer Page?” on page 36).

3.10 HBA Subsystems

The HBA subsystems were introduced in 3.6, “Subsystem Design” on page 29. The individual subsystems are described in this section.

Application Manager

The Application Manager (Figure 14) provides session management for the HBA application by means of the BankServlet. All JavaServer Pages in the site call the BankServlet using the `SERVLET` tag. This calls a method of the BankServlet, which validates that the user has a valid login session. If the user does not have a valid login session, they are redirected to the Not Logged In page; otherwise, the page continues to be loaded.

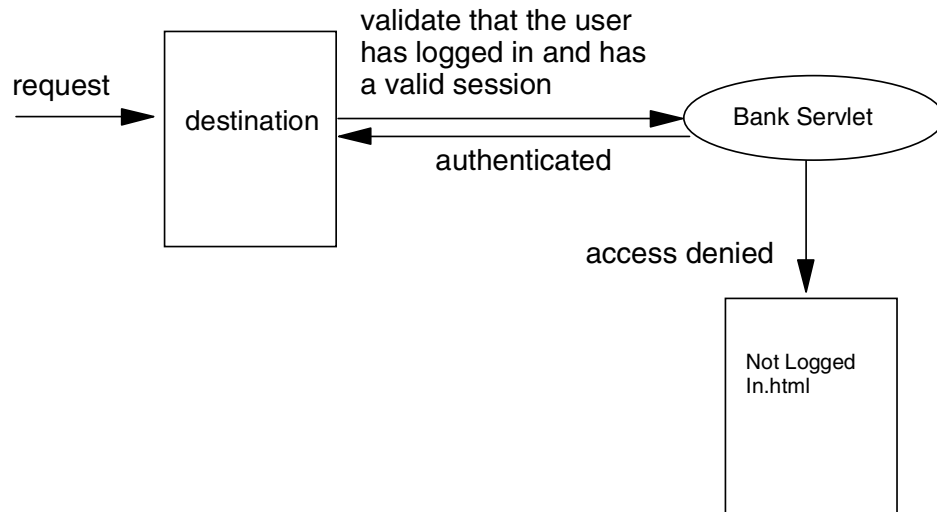


Figure 14. HBA Application Manager

The Application Manager also initializes the bank when the Web server starts and handles the logout function (Figure 15).

The user can choose to log out of the application from anywhere in the site by clicking Logout from the menu. When they click logout, the user gets sent to the Logout JSP and the user’s session is invalidated.

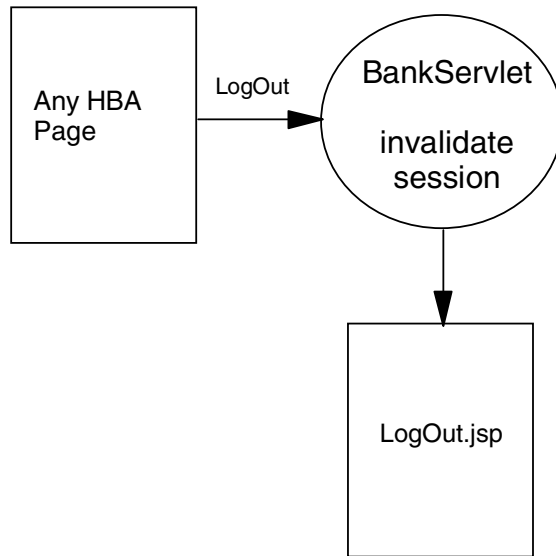


Figure 15. HBA Logout

Login

In order to enter the HBA, a user needs to access the Login page (Figure 16). When the user enters their user ID and password and clicks Login on the Login page, their request is submitted to the LoginServlet. The LoginServlet creates a new session for the user. It then gets the Customer object from the bank based on the userid and checks to see if the password is valid. Once the user has been authenticated, a CustomerView object is added to the user's session. The CustomerView object will be used throughout the HBA to provide access to the customer's data.

If the login attempt was successful, the user is sent to the Accounts page. If authentication was denied the user is redirected to the Unsuccessful Login page.

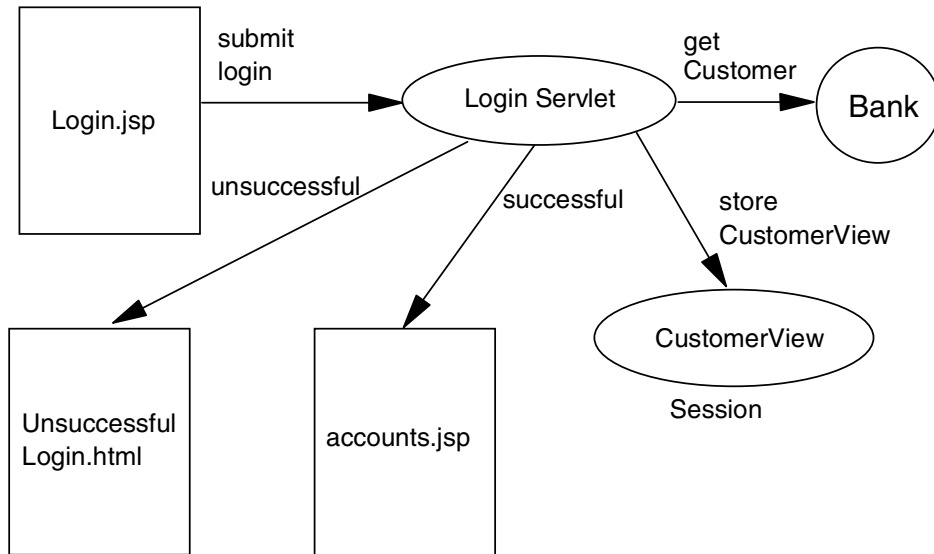


Figure 16. HBA Authentication Sequence

Account Information

The Account Information subsystem provides the Account Balance and Account History function. When the user requests Account Information, they are sent to the Account Information JSP. This page lists the customer's bank accounts along with an option to display the account history or balance.

The Account Information page is called by the AccountServlet with an AccountViewList bean which lists the customer's accounts.

Account History

When the user selects their account and clicks on Account History, the request is sent to the AccountServlet (Figure 17). The AccountServlet retrieves all the transactions for the selected account. The AccountServlet then calls the Account History JavaServer Page where the account history is displayed.

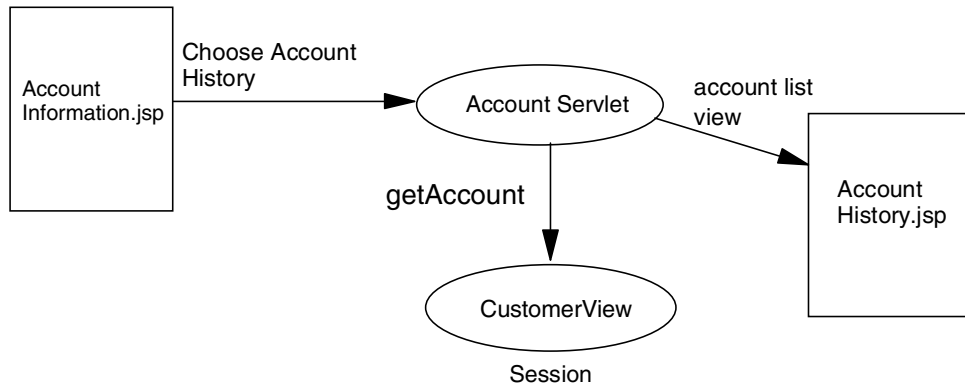


Figure 17. HBA Account History

Account Balance

When the user selects an account and chooses the account balance option from the Account Information JSP the request is sent to the AccountServlet (Figure 18). The AccountServlet retrieves the balance information and calls the Account Balance JSP.

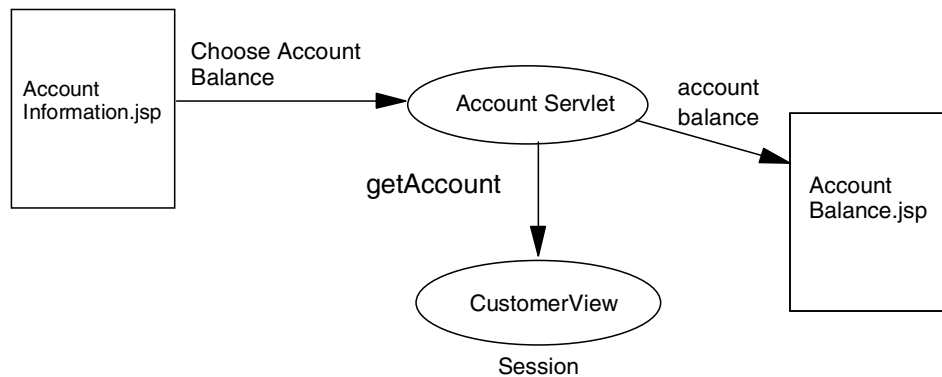


Figure 18. HBA Account Balance

Pay Bill

When the user selects the Pay Bill option from the Accounts JavaServer Page they are sent to the Pay Bill JavaServer Page (Figure 19). The Pay Bill JSP retrieves the account and payee information to display the accounts and payees. The user then selects an account and a payee; enters an amount and the transaction password; and submits the request. The request is sent to the BillPayment servlet that validates the password, performs the

transaction and invokes the Bill Paid JSP. If the validation fails, the user is sent back to the Pay Bill JSP with an error message.

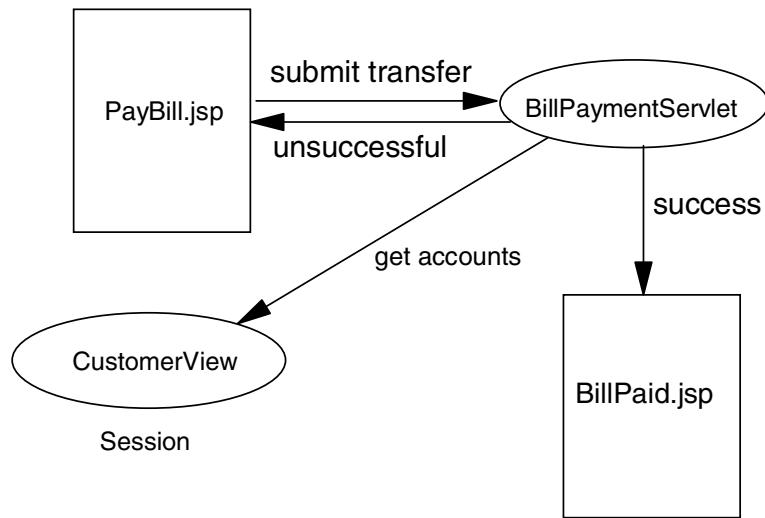


Figure 19. HBA Bill Payment

Payee Setup

When the user selects Payee Setup from the Pay Bill JSP they are sent to the Payee Setup JSP (Figure 20). This page lists all the payees the user currently has. From here the user can choose to add or delete a payee to or from their list.

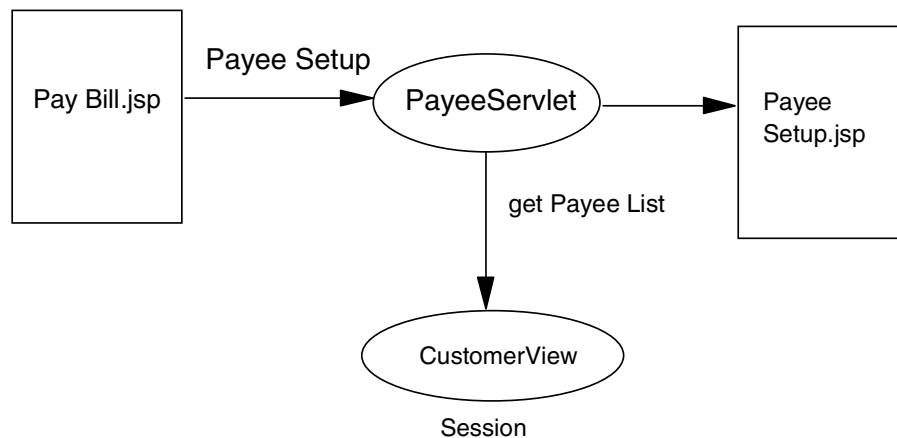


Figure 20. HBA Payee Setup

Add Payee

When the user clicks the Add Payee button they are sent to the Add Payee JSP (Figure 21) where they can select the payee to add to their accounts. The request is sent to the PayeeServlet. The PayeeServlet determines that the action is to add a payee and adds the payee to the user's list of payees. The user is then sent to the Payee Setup JSP, where all of their current Payees are listed, including the one that they just added.

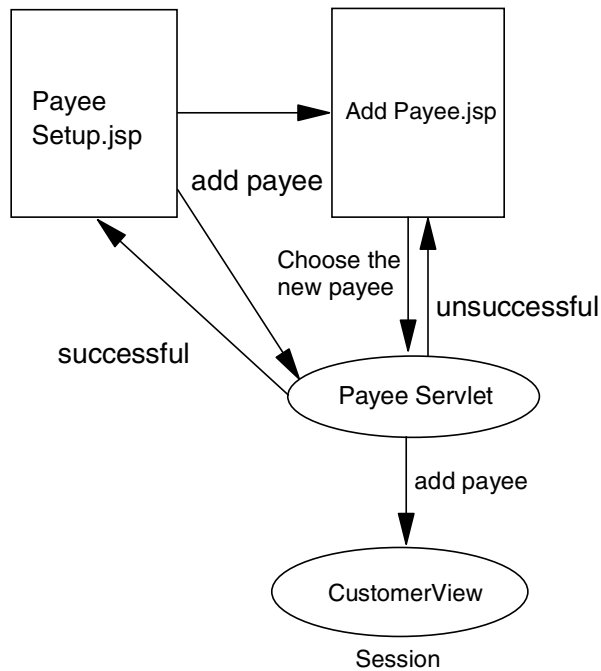


Figure 21. HBA Add Payee

Delete Payee

When the user selects a payee from the Payee Setup JSP and selects the Delete Payee action, the request is sent to the PayeeServlet. The servlet determines that the action is Delete Payee and deletes the payee from the user's Payee list. It then redirects the user to the Payee Setup JSP, where the current list of the users payees is displayed without the one that was just deleted (Figure 22).

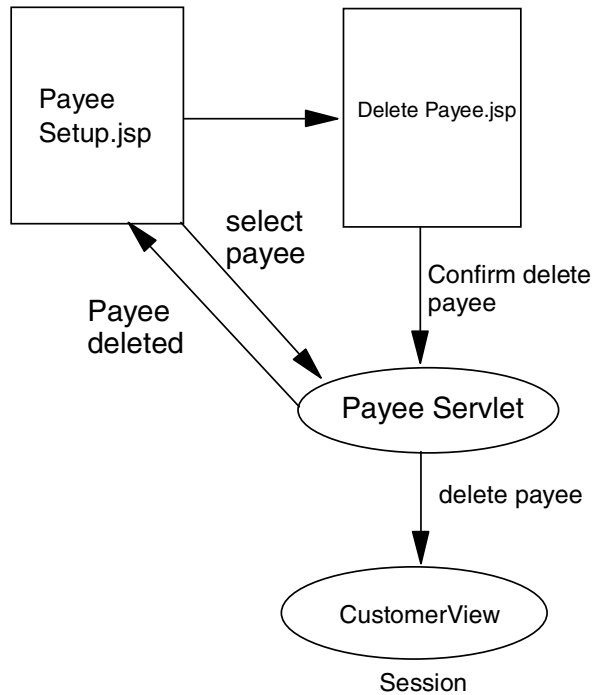


Figure 22. HBA Delete Payee

Transfer Funds

When the user selects Transfer Funds from the Accounts JSP, the request is sent to the TransferFundsServlet (Figure 23). This JSP gets the account information from the customer object to display the accounts. From here, the user selects the source and target accounts and enters the amount and the transaction password and submits the request. The servlet validates and performs the transfer and invokes the Funds Transferred JSP to display the results. If there is an error, the user is sent back to the Transfer Funds JSP with the corresponding error message.

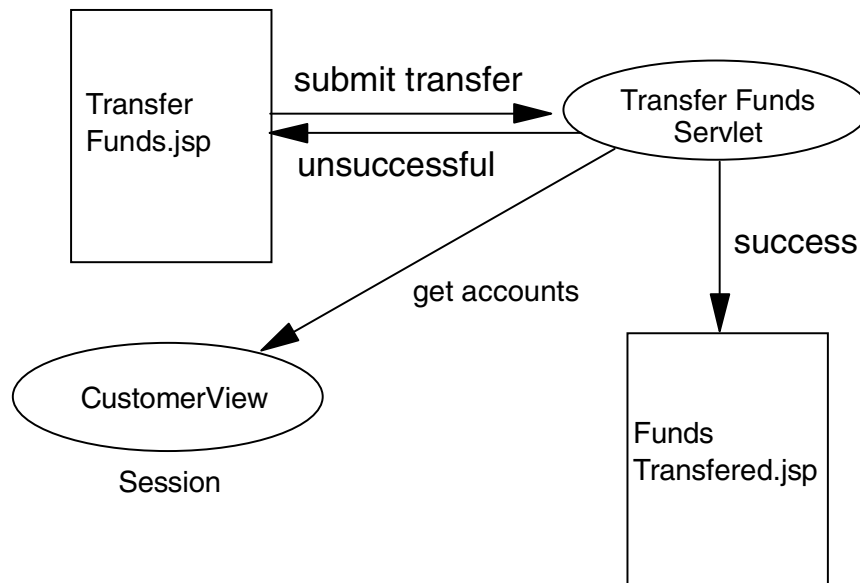


Figure 23. HBA Transfer Funds

User

The User subsystem is composed of one function: Change Password.

When the user selects Change Password from the User JSP, they are sent to the Change Password JSP (Figure 24). There they can choose to change their application login password, which grants them access to the HBA; or their transaction password, which authorizes them to perform transactions. Once they select the type of password they want to change, and enter the old, new, and confirmed new password and click Submit, the request is sent to the ChangePasswordServlet. The servlet validates, and if validated, the user is redirected to the Accounts JSP page. If the validation fails, the user is sent back to the Change Password JSP with an error message.

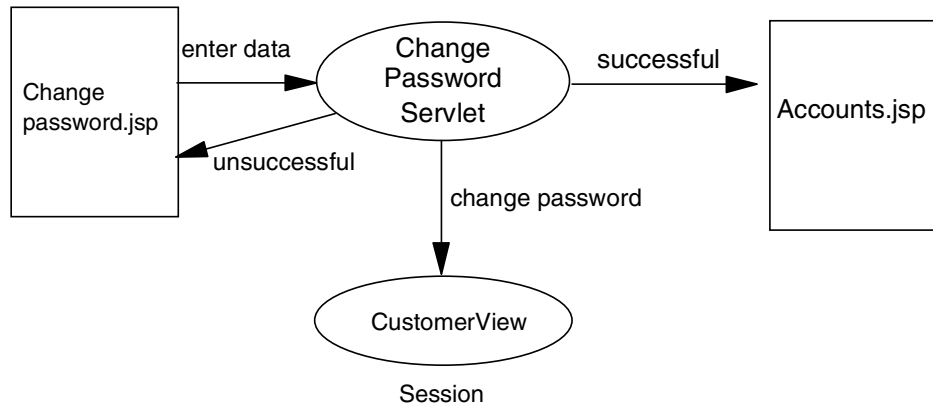


Figure 24. HBA Change Password

Chapter 4. Tool Usage in the Home Banking Application

To create a Web application you need several types of tools. In our case, we are building a JavaServer Pages and servlet based Internet banking system. This chapter describes the tools used, as well as how they were used together. The types of tools and specific tools used were:

- Design and Analysis Tool: Rational Rose 98 Java Edition
- Web Development Environment: WebSphere Studio
 - Java Development Environment: VisualAge for Java
 - Web Site Prototyping Environment: NetObjects Fusion
 - HTML and JSP Page Editor: WebSphere Studio Page Designer
- Application Server: WebSphere Application Server
- Web Servers: IBM HTTP Server and Netscape Enterprise Server

This section introduces the tools used in the HBA development and explains how we used the tools. For more information on each tool, consult the appropriate documentation or the tool's Web site.

4.1 The Tool Suite

The tools available for building Web applications using JavaServer Pages and servlets are maturing, as are the way they are used. New versions of some tools and specifications may have appeared by the time you read this book, so some descriptions may not match your environment. Figure 25 shows the way we used the toolset to create the HBA.

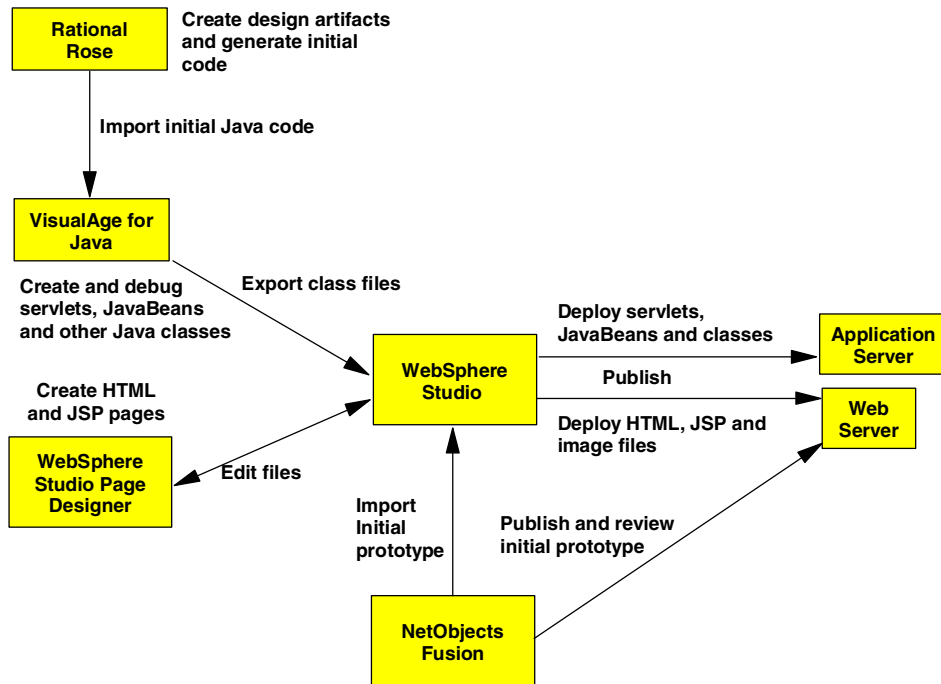


Figure 25. HBA Tool Usage

Initially NetObjects Fusion is used to quickly generate a site prototype. In our case, we also reused much of the NetObjects Fusion site in the final HBA by importing the site into WebSphere Studio. The WebSphere Studio Page Designer is used to add dynamic content and maintain the pages. VisualAge for Java is used to create and maintain all the Java code used in the HBA. WebSphere Studio manages the source control and publishing of the site.

During HBA development, we used two different source control mechanisms, VisualAge for Java and WebSphere Studio. Studio was used to maintain the HBA site, and VisualAge for Java was used to maintain the Java code as it was being developed. In a production environment, an improved scenario would be to add a Software Configuration Management (SCM) tool to manage versions of the application (Figure 26).

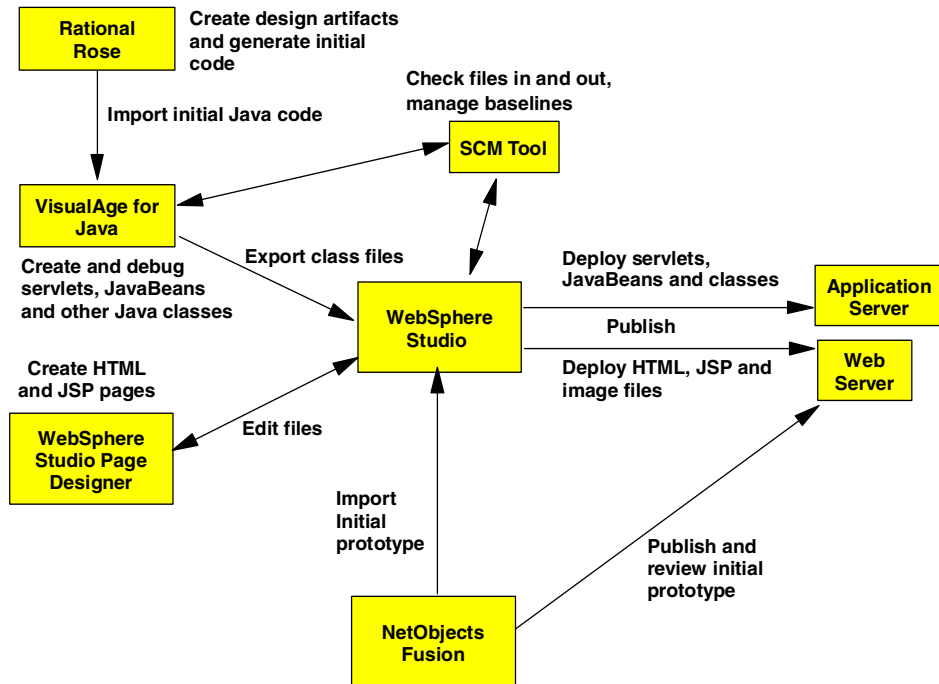


Figure 26. Tool Usage with an SCM Tool

For a description of a similar environment, see Creating WebSphere Applications with VisualAge TeamConnection in Appendix D, “Related Publications” on page 197.

Figure 27 shows the life cycle of tool usage in HBA development. Initially, the Java code is created and unit tested in VisualAge for Java WebSphere Test environment using the JSP and HTML files published by WebSphere Studio. Once the code is working correctly it is imported into WebSphere Studio and published with the rest of the site to test the final deployment using the WebSphere Application Server.

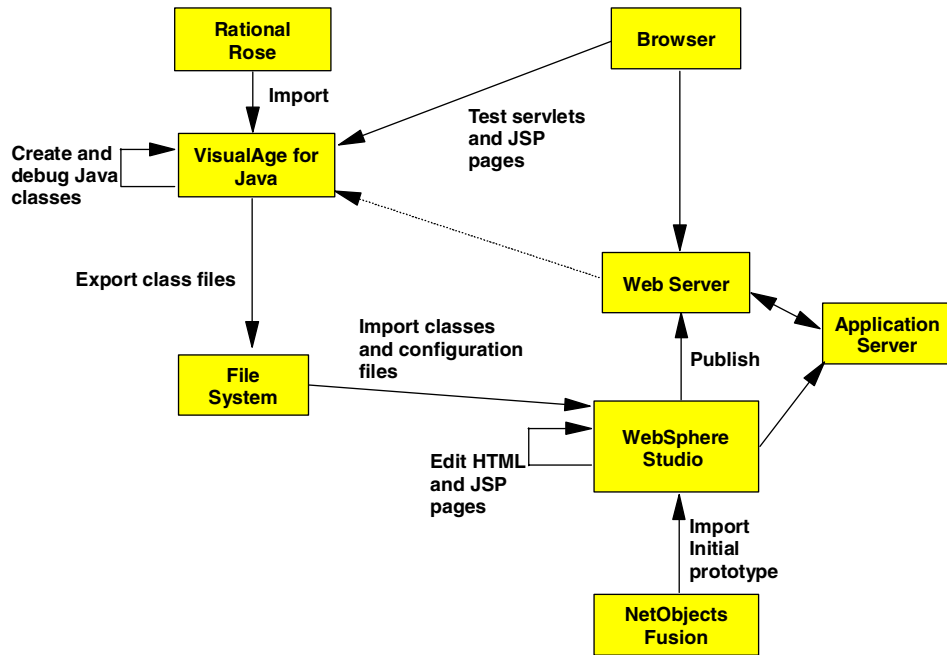


Figure 27. Tool Usage Life Cycle

4.2 Design and Analysis Tool: Rational Rose 98 Java Edition

Rational Rose 98 Java Edition, or Rational Rose, is a popular object modeling, analysis, and design tool from Rational Corporation. We used Rational Rose for the creation of use cases, interaction diagrams, domain object model and the domain firewall. We then generated a first pass at the Java code for the domain firewall and business logic using Rational Rose.

The object model, use case model and sequence diagrams used throughout the book were created in Rational Rose.

4.3 Web Site Prototyping Environment: NetObjects Fusion

We used NetObjects Fusion to build a prototype site. This prototype can be used to demonstrate the proposed site to clients or team members. In our case we also reused the site by importing it into WebSphere Studio.

NetObjects Fusion (Fusion) is a tool for building Web sites without being an HTML expert. With Fusion you can:

- Design your pages
- Publish your Web site to remote locations
- Generate HTML that is consistent across browsers
- Create a well defined environment for controlling the content of your Web site

For more information about Fusion, go to <http://www.netobjects.com>
For our project, we used NetObjects Fusion 4.0 workstation edition. It also comes in a team edition (NetObjects Fusion Authoring Server) supporting collaborative development.

Fusion is used to prototype our HBA application and generate initial HTML pages that we will modify. The prototype site does not have the complete bank functionality, and additional links are added to show pages that would be generated through servlet or JavaServer Page calls.

4.3.1 Prototyping the Site

The HBA application was developed using one of NetObjects Fusion's custom templates: Company Internet. This template quickly provided the HBA with a consistent look and feel, including the Site Navigation Bar. The pattern is quickly recognizable by a new user as shown in Figure 28.

Fusion makes it easy to make changes throughout the site. You change a property in one page and it cascades throughout the site to reflect your change in every page in the site. Fusion does this by the use of MasterBorders. Pages in a section of the site can share a MasterBorder, and if an area of the MasterBorder is changed in one page, it is reflected in all pages that use that MasterBorder.

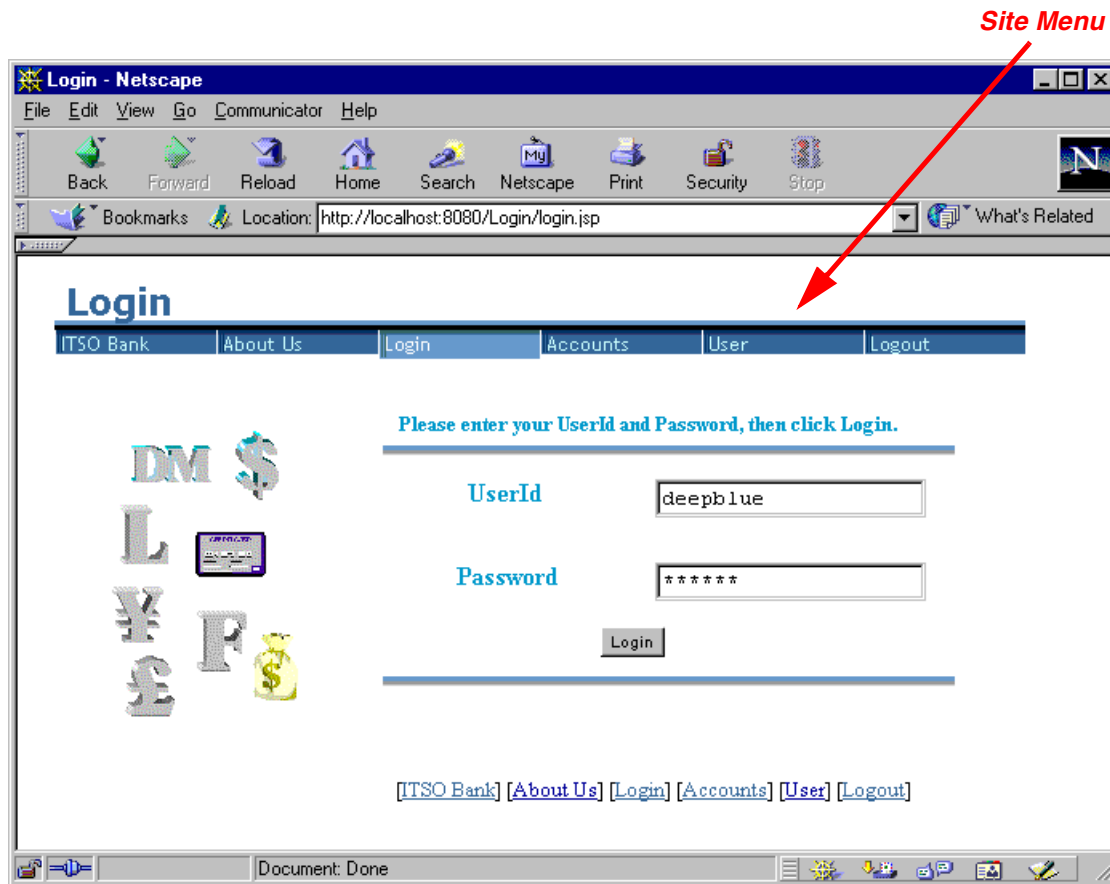


Figure 28. Site Navigation Bar, or Menu, of the HBA Application

Creating the HTML Pages

Fusion provides a visual editor for creating the pages of the site. We used this feature to prototype all the pages of our site. We simply dragged and dropped components on our pages and positioned and labeled them (Figure 29). Once we created our pages visually, we could then preview them before we published them.

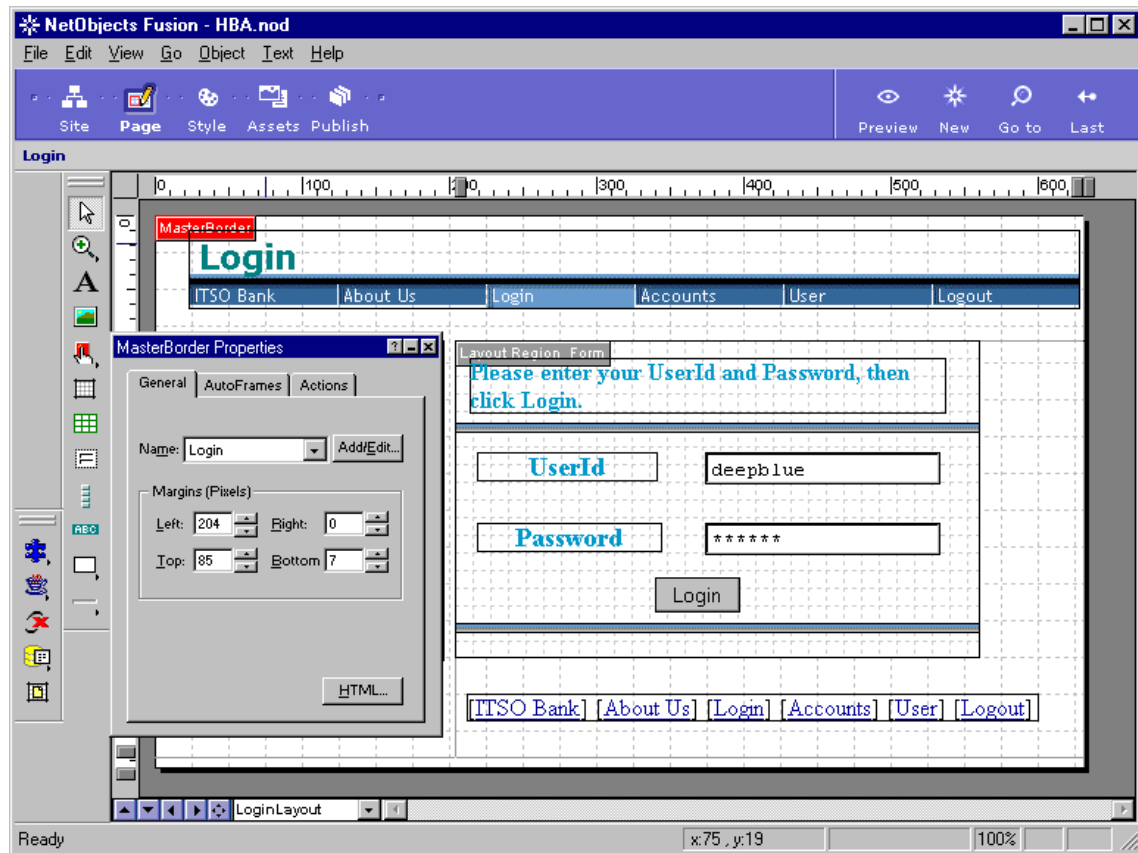


Figure 29. NetObjects Fusion Visual Page Editor

Creating the Dummy Content

In the final HBA site, much of the content will be dynamically generated using servlets and JavaServer Pages. For the prototype these tables, lists and other elements were filled in with dummy content. For example, list boxes were populated with dummy account IDs.

Creating the Prototype Links

In the HBA site, many of the JavaServer Pages are only accessed through servlets, as shown in the HBA application flow in Figure 9 on page 30. In the prototype, because there are no servlets, we need to provide a mechanism to display and review these pages. We did this by creating extra links to these pages.

Figure 30 shows the Accounts.html page with the extra links at the bottom of the page. This page will become Accounts.jsp in the final site.

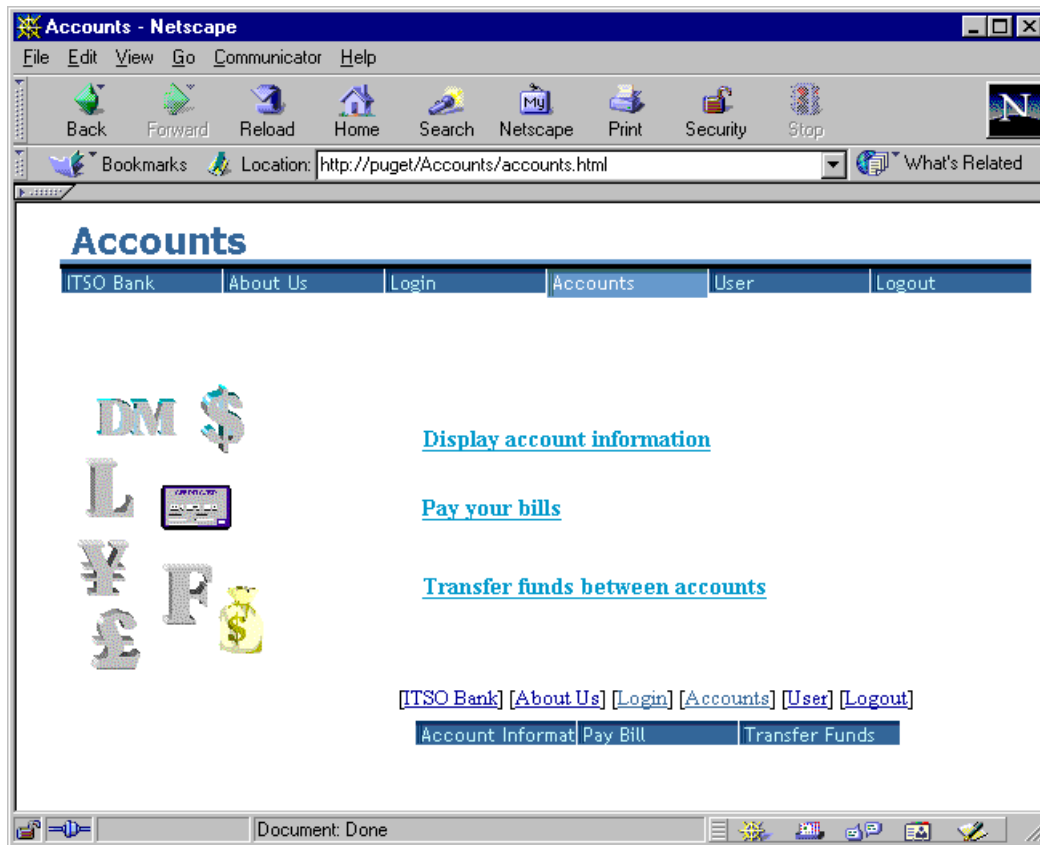


Figure 30. Extra Links on an HBA Page

Publishing the Prototype Web site

Fusion manages all the pages and resources of our Web site in its own format. Once we are ready to publish, we use the Publishing Wizard (Figure 31). We set the directory where we want to publish: a local directory, or a remote server. We click **OK**, and Fusion generates all the HTML pages and images for our site and puts them in the appropriate directories.

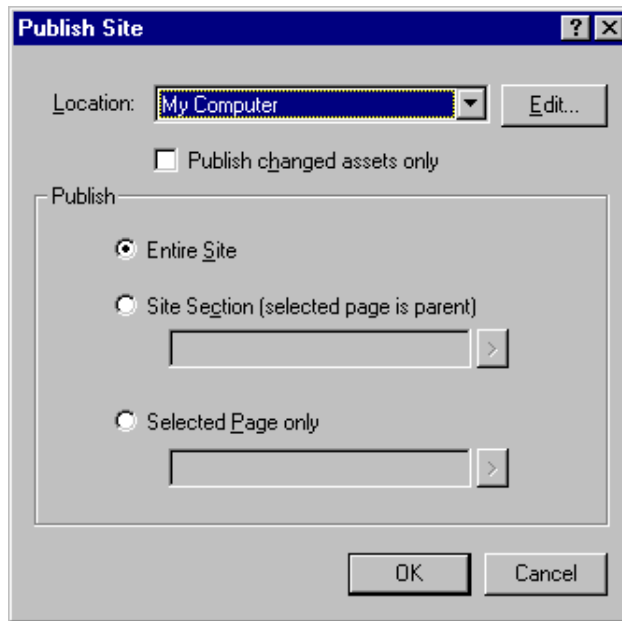


Figure 31. Fusion Publishing Wizard

The resulting files can be seen in Figure 32.

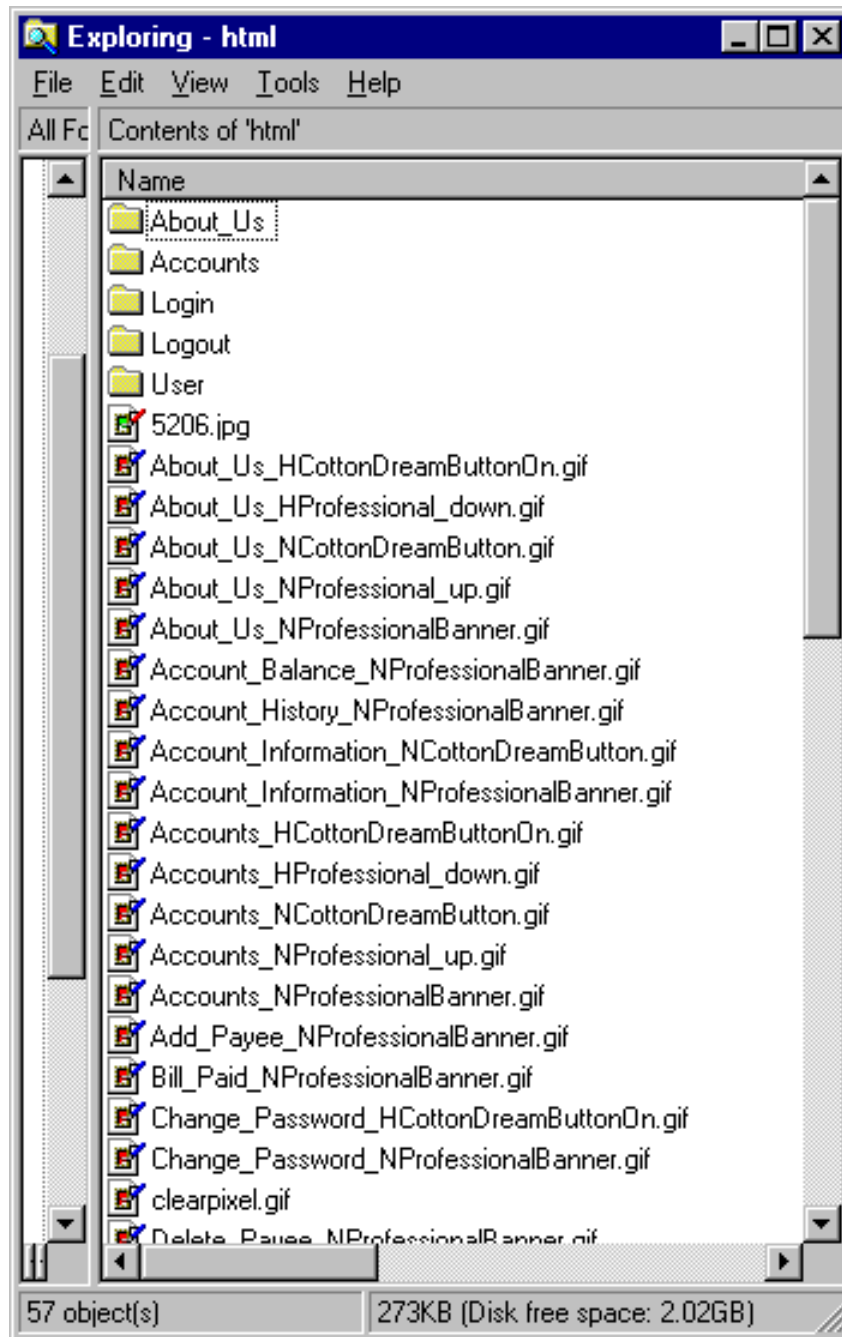


Figure 32. Fusion Generated Site in Windows NT Explorer

4.4 Web Development Environment: WebSphere Studio

WebSphere Studio is an integrated suite of tools and wizards for building Web applications. It has the following components:

- NetObjects ScriptBuilder—A language sensitive text editor that you can use for a number of scripting, markup, and programming languages.
- NetObjects Fusion—A graphical, authoring tool for designing, building, and managing entire Web sites. See 4.3, “Web Site Prototyping Environment: NetObjects Fusion” on page 50.
- VisualAge for Java, Professional Edition—IBM’s Java development environment. See 4.5, “Java Development Environment: VisualAge for Java” on page 71.
- Web Development Workbench—The workbench provides a complete application assembly environment for dynamic Web sites.
- Page Designer—An integrated HTML and JSP page editor.
- Applet Designer—A graphical tool that allows you to create multimedia applets for your Web pages.

For more information on WebSphere Studio, see www.software.ibm.com/webservers/studio and the product documentation.

We did not use the NetObjects ScriptBuilder or the Applet Designer in the HBA project, so we do not describe them here. NetObjects Fusion and VisualAge for Java are described in 4.3, “Web Site Prototyping Environment: NetObjects Fusion” on page 50 and 4.5, “Java Development Environment: VisualAge for Java” on page 71. In this section we will describe the Web Development Workbench and Page Designer. When we discuss the Web Development Workbench we will often refer to it as WebSphere Studio.

Web Development Workbench

Using the WebSphere Studio workbench, you can view, edit, and manage your site during the development and publishing processes. The workbench has the following features:

Views

The WebSphere Studio workbench provides three views of your files:

- File View: Shows all the files and folders in your site. It is shown in the left-hand side of the workbench window.
- The right-hand side of the workbench window can show one of two views:
 - Relations View: Shows links between files.

- **Publishing View:** Shows the Assembly stages. You can create an unlimited number of assembly stages in which you develop and test your site without affecting the production version.

Report Generation

Studio can generate the following reports to help you manage development of your Web site:

- **Import Report**—Lists the options you select during import and shows the status of imported files.
- **Publishing Report**—Summarizes the results of the publishing process.
- **Project Integrity Report**—Summarizes the results of a check for broken links in your Web project.
- **File Report**—Provides detailed information about a file.
- **Assembly Stage Report**—Provides detailed information about a project and its assembly stages.
- **Relations map**—Shows the relationship between a selected file, its parent files, and its children files.
- **Project map**—Shows the relationships among the files in your application.

Team Development

You can use several popular Software Configuration Management tools to provide more sophisticated version control and release management of your applications.

Site Import

When you already have an existing Web site, importing is a quick and painless way to create WebSphere Studio Web sites and populate them with files.

Link Management

There are several types of links that Studio manages or helps you manage: industry-standard links, source links, generated links, and custom links. WebSphere Studio automatically recognizes and manages industry-standard links; you must identify the other three kinds. Once you identify generated links, WebSphere Studio dynamically manages them. For example, when you rename or move a file within WebSphere Studio, the links pointing to it are automatically updated.

The WebSphere Studio link types are:

- **Inside link**—A link to another file in the site. Appears as a solid line with an arrow head at the end.

- Embedded link—A link to a file which is not a hyperlink (does not have a HREF tag, but a tag such as). Appears as a solid line with a small depression in the form of a V with a dot.
- Outside link—A link to a file outside the site. Appears as a dashed line.
- Broken link—A link to a published file whose child file is not set for publishing, or a link to a file that does not exist. Appears as a double-crossing line on either a solid line (inside link) or a dashed line (outside link).
- Self link—A link from a file to itself. Appears as an arrow that loops back to the file icon.
- Anchor link—A link from one position in a file to another position in the same file. Appears as a solid line with an anchor.
- Unverified link—A link to a file that cannot be verified using HTTP. Appears as a dashed line with a question mark at its end.
- Source Link—A link from a publishable file to the file that is used to create it. For example, you can create a source link from a .class file to the Java source. Appears as a solid line with an open square containing two arrows.
- Custom Link—A link that you create to identify a relationship unrecognized by WebSphere Studio. Appears as a solid line with a small star.
- Generated Link—A link that is generated by a user-defined rule and existing hyperlinks. Appears as a multi-headed arrow on either a solid line (inside link) or a dashed line (outside link).

Publishing Support

WebSphere Studio supports publishing your entire site to different servers and supports different publishing stages (such as test and production). You can specify the directories to which different parts of the site are published and set files to not be published.

Site and Sub-Site Archiving

You can archive your site or parts of your site when you are finished development or to produce development baselines.

Integration with Various Asset Editing Tools

You can specify default editors for each type of file in your site.

Style Sheet Support

Use cascading style sheets to provide a consistent look and feel for your site.

Code Generation Wizards

You can easily generate servlets which access databases or JavaBeans.

4.4.1 Page Designer

The Page Designer is an integrated JSP and HTML page editor. You can use the Page Designer to edit the JSP and HTML pages of your site. The editor provides Normal (WYSIWYG) or source views of the pages. Figure 33 shows the Normal view and Figure 34 shows the HTML Source view of the Page Designer.

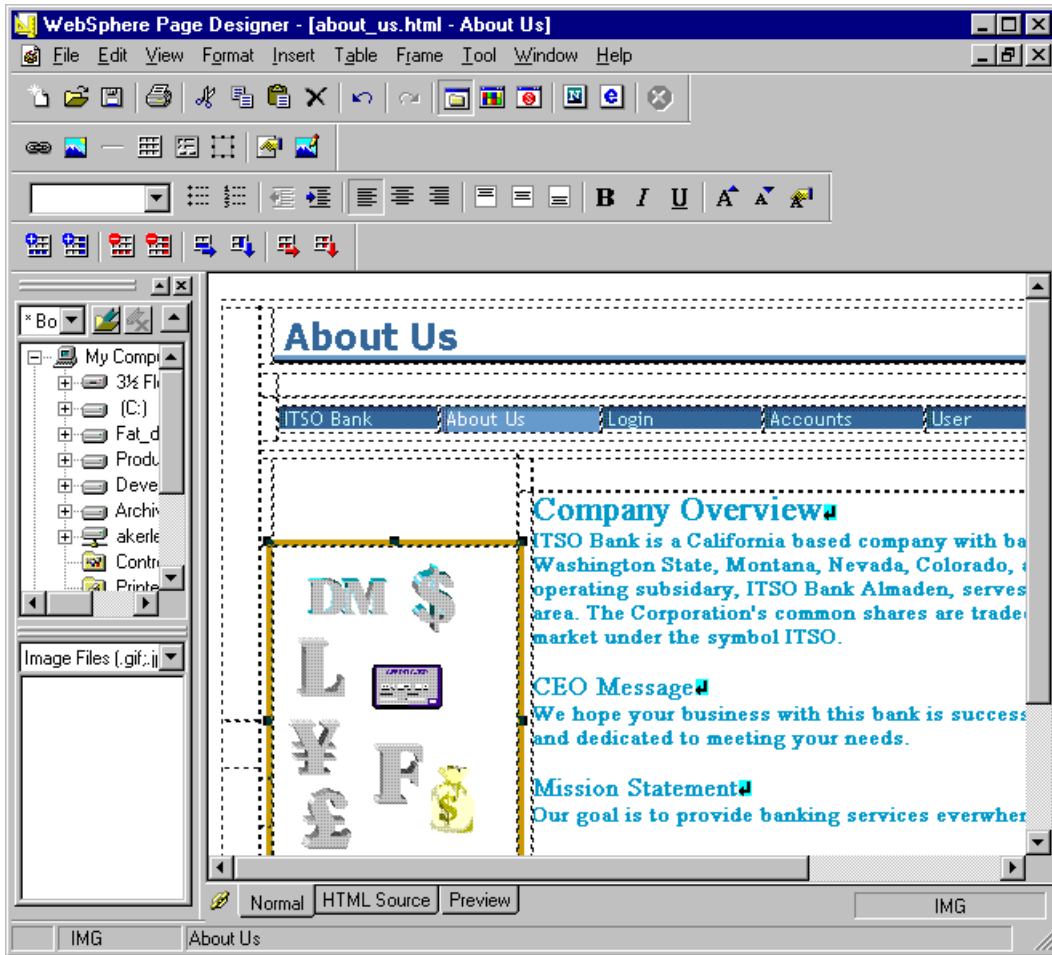


Figure 33. Page Designer—Normal View

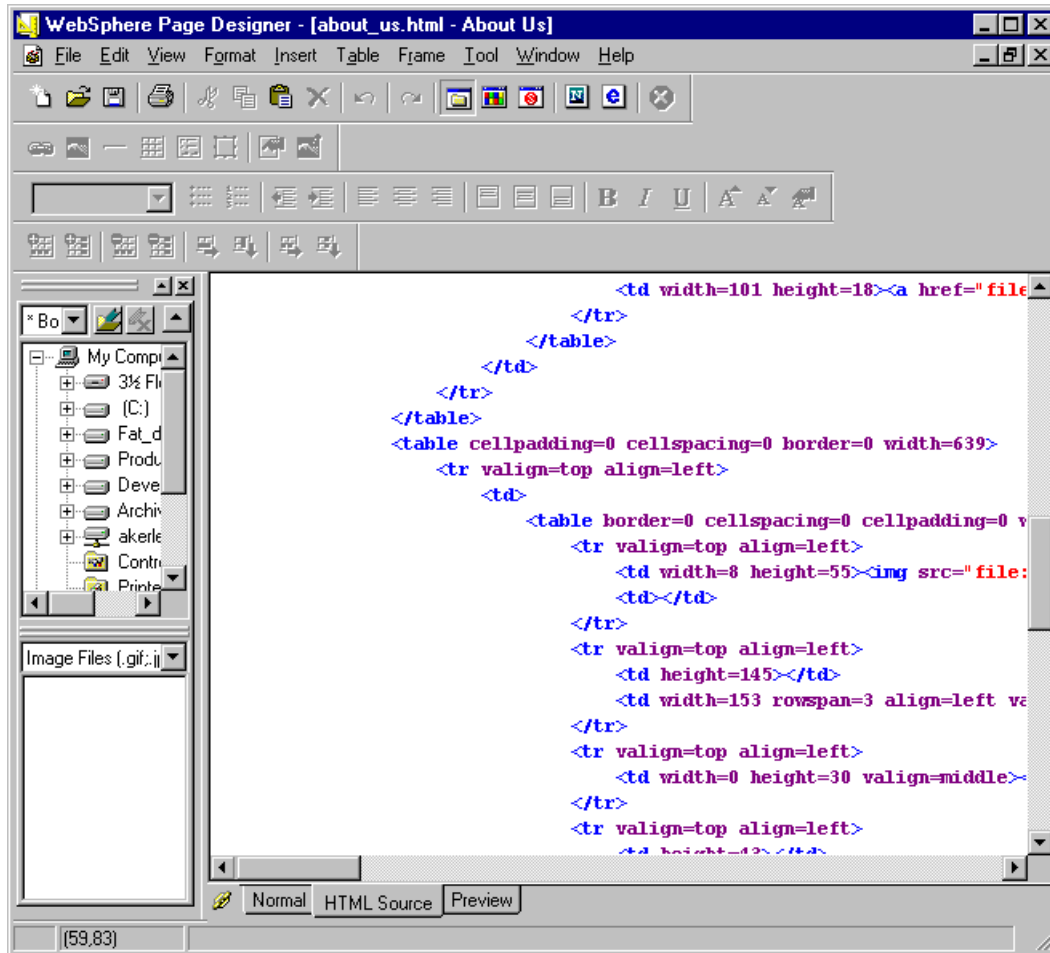


Figure 34. Page Designer—HTML Source View

You can use the Page Designer to insert JSP syntax into your page, as shown in Figure 35.

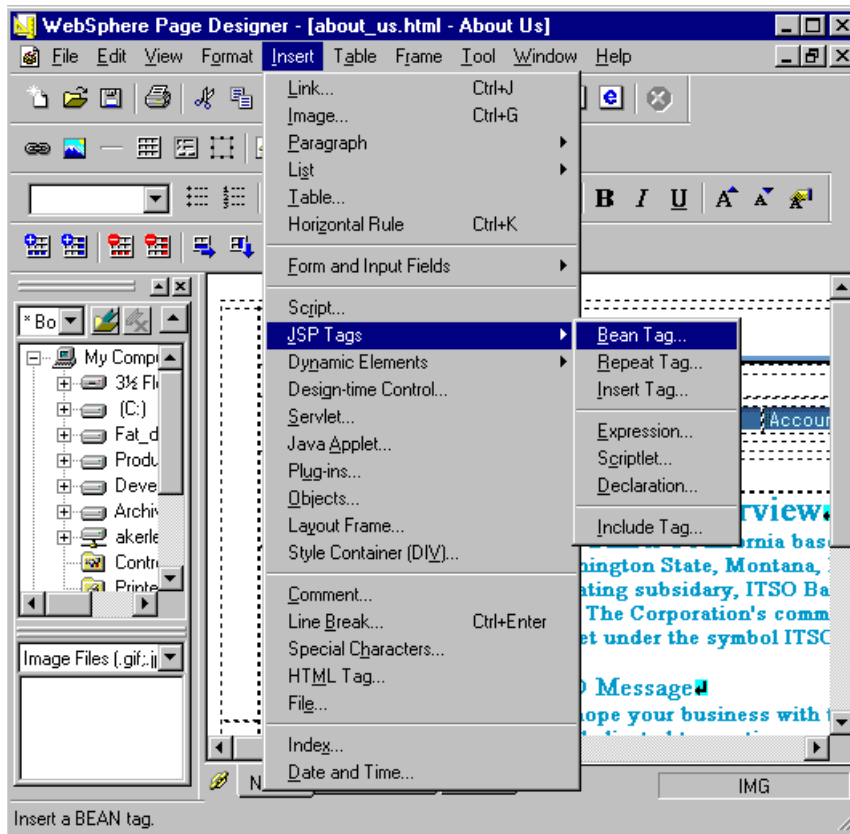


Figure 35. JSP Support in the Page Designer

The Page Designer also provides the AnimatedGIF Designer and WebArt Designer to help you produce animated GIFs and other Web graphics.

For more information on the Page Designer, see the product documentation.

4.4.2 Importing the Site

Once the HBA prototype was acceptable, we imported the site into WebSphere Studio. We used **File**→**Import Site** and specified `http://localhost` as the URL. Figure 36 shows the Import dialog and Figure 37 shows the Relations view of the imported site.

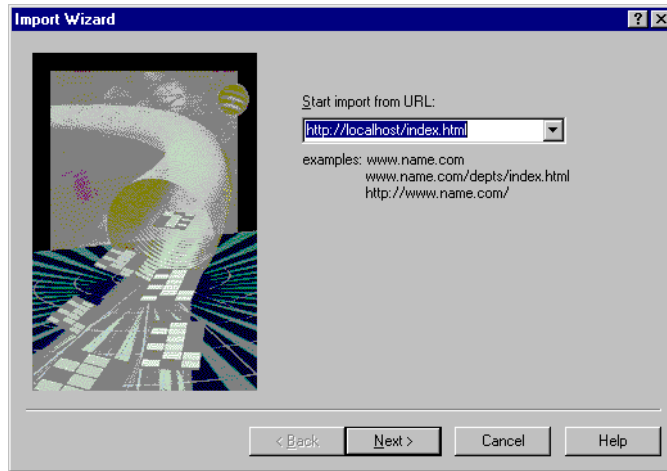


Figure 36. Importing the Prototype Site

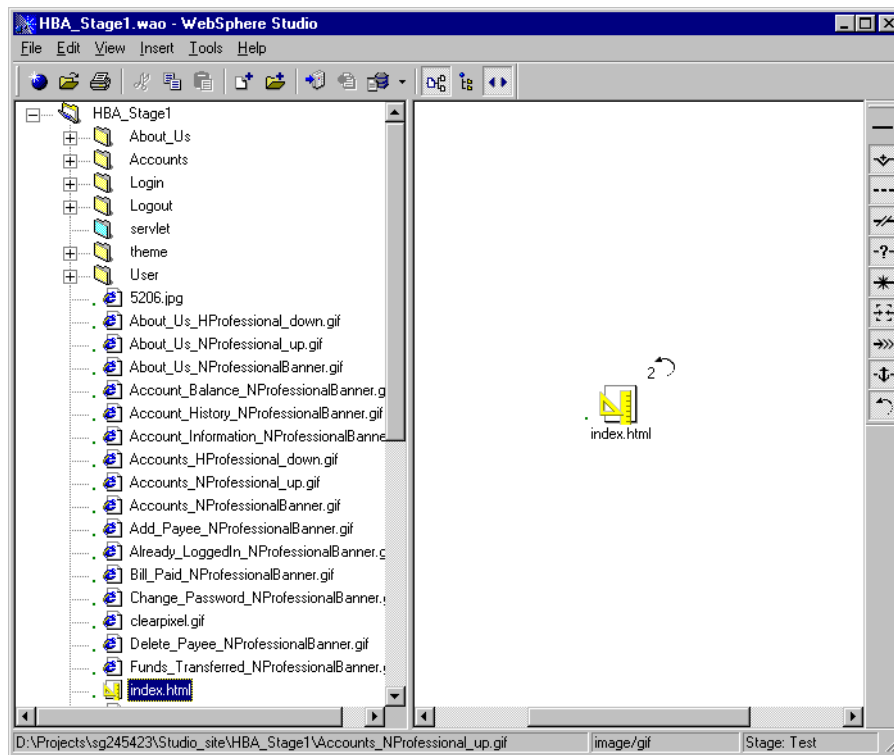


Figure 37. Relations View of the Imported Site

4.4.3 Restructuring the Site

We made several changes to the structure of the site created in NetObjects Fusion to change it to a dynamic site and make it easier to work with using WebSphere Studio:

- Move all HTML files to the top level directory
- Move images to a subdirectory
- Rename HTML to JSP files
- Remove prototype links
- Create a classes folder
- Set publishing targets
- Delete the Theme folder

Move all JSP and HTML Files to the Top Level Directory

NetObjects Fusion creates a subdirectory for each child level of the site. This results in many directories being created, sometimes holding only one file. We decided to move all the HTML pages to the top level of the site. Once we moved the files we deleted the extra folders that NetObjects Fusion had created.

Move Images to a Subdirectory

To make it easier to refer to images and keep them organized we moved all the images to an images subdirectory. We simply created a new folder in our WebSphere Studio project named images and dragged all the images to this folder. All affected links were automatically updated.

Rename HTML to JSP Files

Although we could have changed the extension within NetObjects Fusion, the prototype would then not have worked. We simply selected each file that was intended to be a JavaServer Page, and changed it to a .jsp extension (Figure 38).

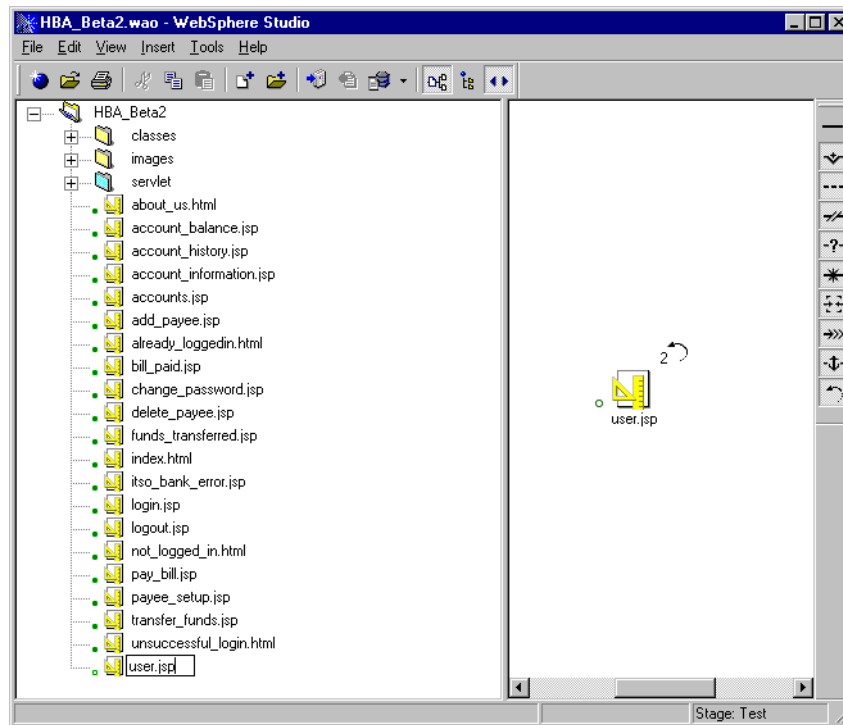


Figure 38. Changing File Extensions

Remove Prototype Links

In the prototype, extra links were created in order to see pages which would be generated through a call to a servlet or a JSP in the running application. These links were removed at this stage.

Create a Classes Folder

We created a classes folder to hold the JAR file for the site. The JAR file will contain all the Java types used in the HBA except the servlets.

Set Publishing Targets

The site is to be published to three different areas:

- Document Root directory for all image, JSP and HTML files
- WebSphere Application Server servlets directory for all the servlets and servlet configuration files
- WebSphere Application Server classes directory for the bank.jar file

To set the publishing targets we selected the server in the Publish View and selected **Edit**→**Properties** and the **Publish** tab. On the Publish page we clicked on Define Publishing Targets and set the targets as shown in Figure 39.

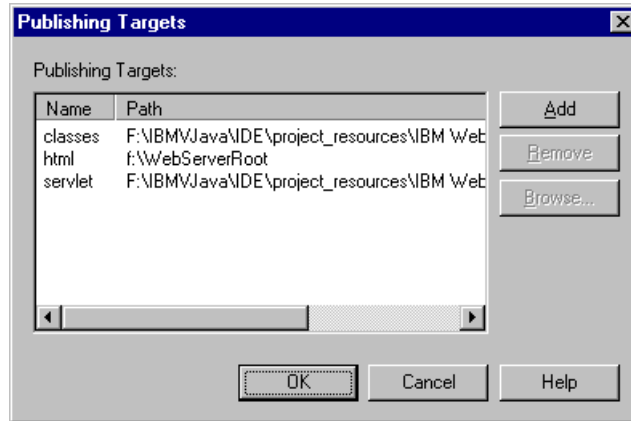


Figure 39. Defining Publishing Targets

After the Publishing Targets were set, we configured the project to publish to a local Windows NT configuration (Figure 40).

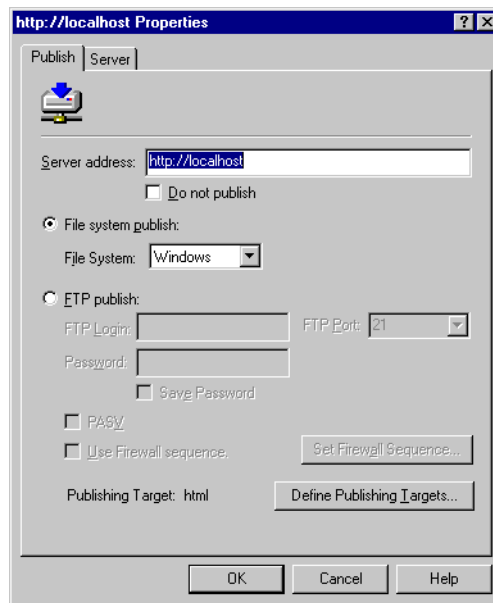


Figure 40. Publish Setup

Delete the Theme folder

We are not using style sheets for this project, so we deleted the Theme folder.

Figure 41 shows the Files view after restructuring.

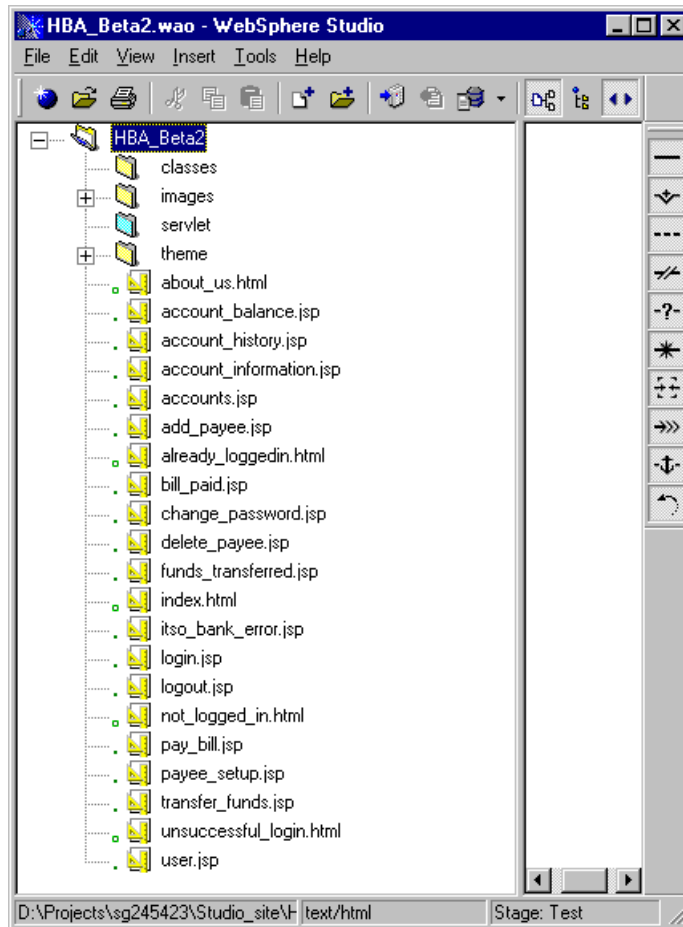


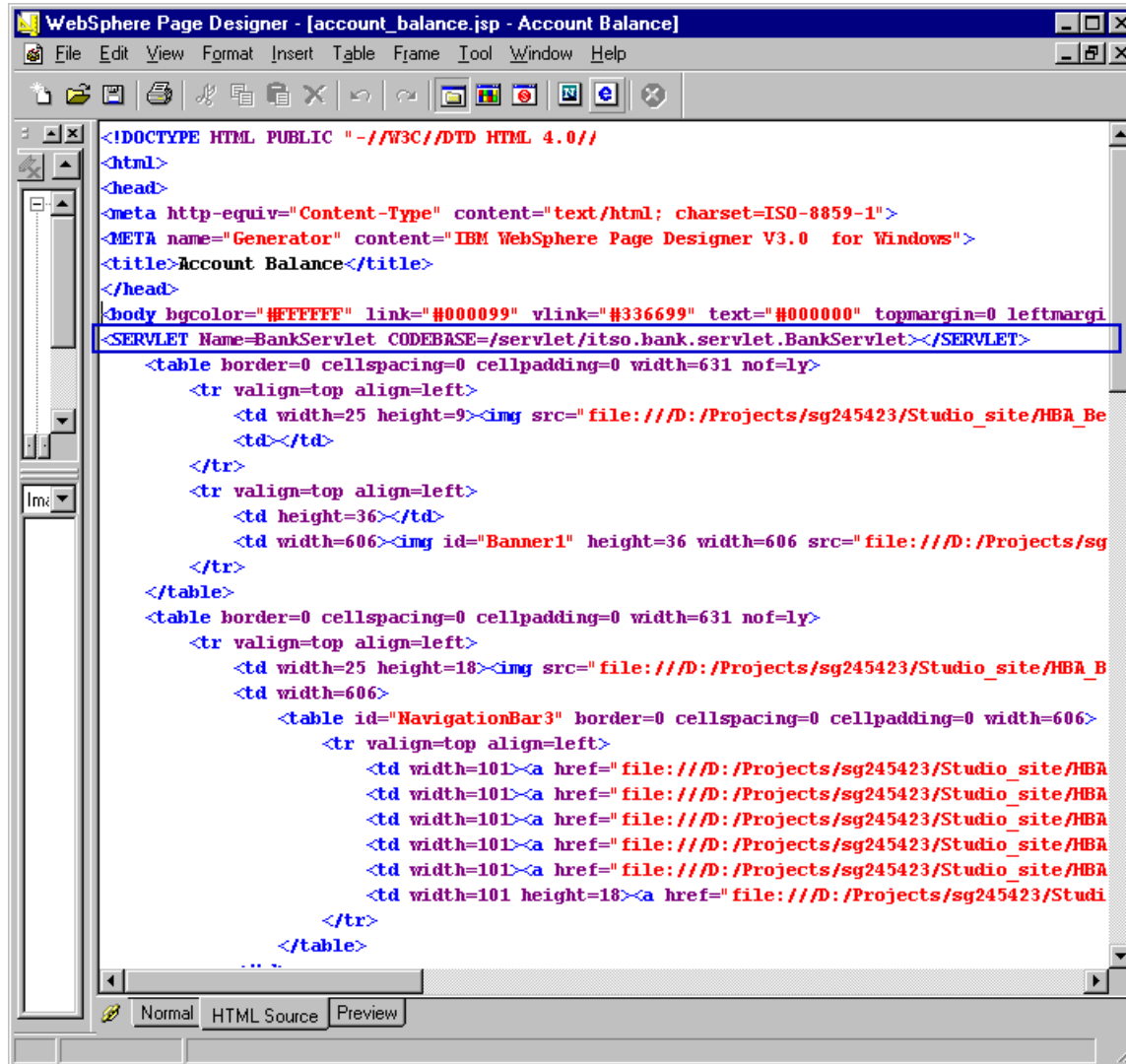
Figure 41. WebSphere Studio Files View after Site Restructure

4.4.4 Adding Dynamic Pages to the Site

2.4, “JavaServer Pages” on page 10 described the different types of JSP elements. In 3.8.3, “What Goes into a JavaServer Page?” on page 36, we discussed the reasons you might use the different types of JSP elements in a JavaServer Page. In this section we describe how JavaServer Pages were created for the HBA using the WebSphere Studio Page Designer.

Adding the SERVLET Tag

Our HBA architecture (3.8, “HBA Architecture and Design” on page 32) calls for a servlet to be invoked in each JSP to determine whether the user is authenticated or should be redirected to another area of the site. The SERVLET tag is added to each JavaServer Page using the Page Designer. The SERVLET tag is added as the first element of the HTML BODY, as shown in Figure 42.



```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//  
<html>  
<head>  
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">  
<META name="Generator" content="IBM WebSphere Page Designer V3.0 for Windows">  
<title>Account Balance</title>  
</head>  
<body bgcolor="#FFFFFF" link="#000099" vlink="#336699" text="#000000" topmargin=0 leftmargin=0>  
<SERVLET Name=BankServlet CODEBASE=/servlet/itso.bank.servlet.BankServlet></SERVLET>  
<table border=0 cellspacing=0 cellpadding=0 width=631 nofl=y>  
<tr valign=top align=left>  
<td width=25 height=9>  
</table>  
<table border=0 cellspacing=0 cellpadding=0 width=631 nofl=y>  
<tr valign=top align=left>  
<td width=25 height=18>  
<tr valign=top align=left>  
<td width=101><a href="file:///D:/Projects/sg245423/Studio_site/HBA  
<td width=101><a href="file:///D:/Projects/sg245423/Studio_site/HBA  
<td width=101><a href="file:///D:/Projects/sg245423/Studio_site/HBA  
<td width=101><a href="file:///D:/Projects/sg245423/Studio_site/HBA  
<td width=101><a href="file:///D:/Projects/sg245423/Studio_site/HBA  
<td width=101 height=18><a href="file:///D:/Projects/sg245423/Studi  
</tr>  
</table>  
</table>  
...
```

Figure 42. Adding the SERVLET Tag

We could also add the `SERVLET` tag using the **Insert**→**Servlet** menu item in the Normal View of the Page Designer.

Adding JSP Elements

We used JSP elements to display all the dynamic content for the site. The JSP elements were added to each file using the Page Designer.

In each JavaServer Page we replaced the dummy text we created using NetObjects Fusion with the JSP syntax required to provide our HBA functionality. The specific syntax for each page is discussed in Chapter 5, “Implementing the Home Banking Application” on page 99. The following is an example of adding JSP syntax using the Page Designer.

In the Page Designer we either select the table in Normal view and then switch to the HTML source view, or search for `TABLE 3` (the name that NetObjects Fusion gave the table) in HTML Source View and then insert the JSP syntax:

```
<BEAN NAME="account" TYPE="itso.bank.viewobjects.BankAccountView" INTROSPECT="no"
      CREATE="no" SCOPE="request"> </BEAN>

<table id="Table3" border=1 cellspacing=1 cellpadding=3 width=408 <TR>
  <TBODY><TR>
    <TD WIDTH=72><P ALIGN=LEFT><B><FONT COLOR="#0099CC" SIZE="+1">Date</FONT></B></TD>
    <TD WIDTH=90><P ALIGN=LEFT><B><FONT COLOR=#0099CC SIZE="+1">Type</FONT></B></TD>
    <TD WIDTH=99><P ALIGN=CENTER><B><FONT COLOR=#0099CC SIZE="+1">Amount</FONT></B></TD>
    <TD WIDTH=108><P ALIGN=CENTER><B>
      <FONT COLOR=#0099CC SIZE="+1">Balance</FONT></B></TD></TR>
  <tr>
    <repeat index=count>
      <% account.getTransactions( count); %>
      <td><insert bean=account
        property=transactions(count).transTimeStamp></insert></td>
      <td><insert bean=account property=transactions(count).transType></insert></td>
      <td><insert bean=account
        property=transactions(count).transAmount></insert></td>
      <td><insert bean=account
        property=transactions(count).transClosingBalance></insert></td>
    </tr>
  </repeat>
</TBODY>
</table>
```

We can immediately preview the page (Figure 43), or continue to work with the visual representation of the page (Figure 44).

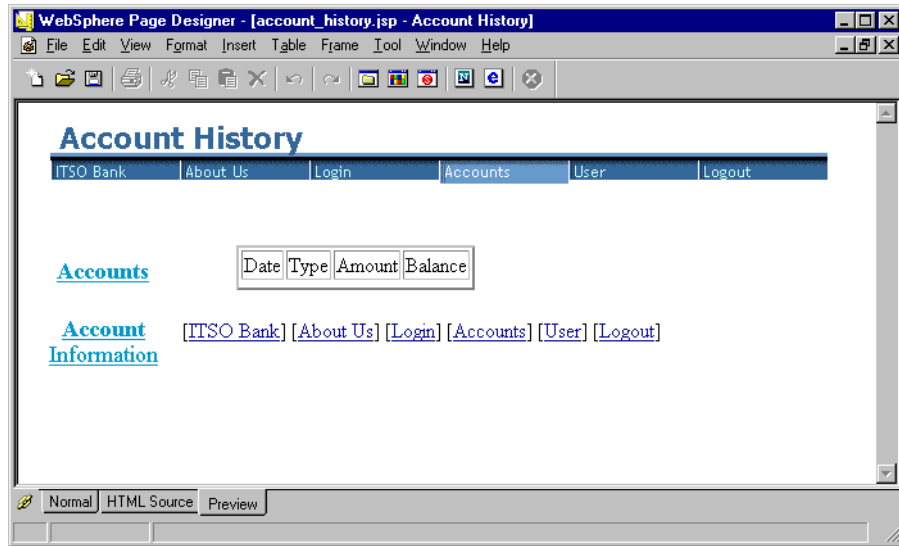


Figure 43. Previewing the Account History Page in the Page Designer

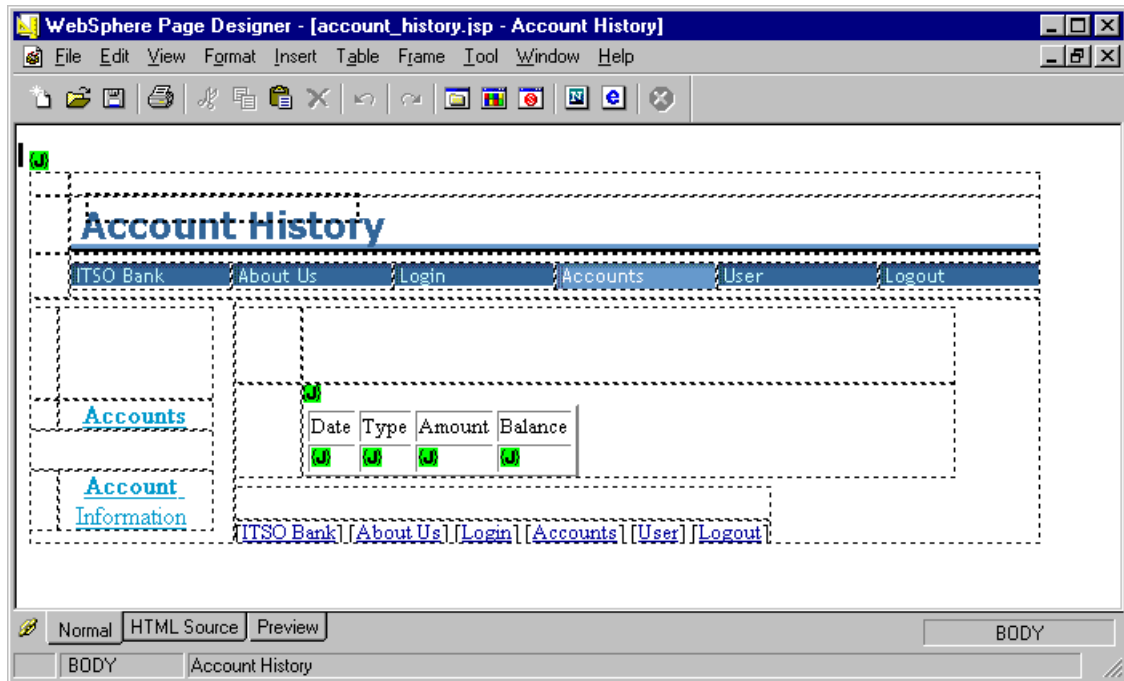


Figure 44. Editing the Account History Page in the Page Designer

Creating Links to HBA Java Components

As discussed in 4.1, “The Tool Suite” on page 47, the Java code for the HBA is created and unit tested in VisualAge for Java. Once the code is working, the servlets are exported as class files, while all the supporting Java classes are exported as one Jar file. The Jar file (bank.jar) is added to the classes folder, while the servlets directory (\itso\bank\servlet) is added to the servlets folder. The XML servlet configuration files are added to the WebSphere Studio servlet folder from the VisualAge for Java Project Resources directory. The links to the servlet configuration files and the servlets are automatically created by WebSphere Studio. We then created a custom link from index.html to the JAR file to ensure that the site was complete when published.

The complete site for the HBA can be found at:

<ftp://www.redbooks.ibm.com/redbooks/SG245423/>

4.4.5 Publishing the Site

The site is published to the Assembly stage by selecting the Test assembly stage in the Publish view and then selecting **File→Publish Whole Project**.

4.5 Java Development Environment: VisualAge for Java

VisualAge for Java is IBM’s Java development environment. It is an integrated, visual development environment with powerful support for JavaBeans, client/server development, visual programming and enterprise connectivity.

These are three VisualAge for Java editions: Entry, Professional, and Enterprise.

- VisualAge for Java Entry Edition is a free version with a 500 class limit. This makes it ideal for small projects or evaluation purposes.
- VisualAge for Java Professional Edition removes the 500 class limit from the Entry edition.
- VisualAge for Java Enterprise Edition adds enterprise access builders and a team programming environment to the Professional Edition.

Common to all editions is:

- Incremental compilation
- Visual Composition Editor—for visual programming

- Integrated Development Environment, including:
 - Debugger
 - Browsers—Project, Package, and Class
 - Source code editor
- Repository-based environment
- Advanced coding tools, including automatic formatting, automatic code completion, and fix-on-save
- Data Access Beans for simplified access to relational databases

For more information on VisualAge for Java see www.software.ibm.com/ad/vajava and Appendix D., “Related Publications” on page 197.

4.5.1 Developing Servlets with VisualAge for Java

VisualAge for Java is a powerful servlet development and testing environment supporting multiple JVM emulation as well as incremental compilation and linking. In particular, it has strong support for testing and debugging servlets, which is one of the more complex tasks of servlet development. To appreciate these strengths, we need to contrast servlet development using VisualAge for Java with traditional approaches.

In a typical servlet development life cycle, the servlet is developed and then deployed to an application server for testing. Debugging the servlet typically involves some well-placed print statements or writing to the log. This clutters the application code, and we have to remember to either remove these debugging statements or wrap them in an if statement and use a debug attribute to toggle them on or off. Checking the errors involves mining through the server’s error logs. Code modifications involve deploying the servlet back to the server during each iteration. In addition, if the server’s JVM does not support automatically reloading the updated servlet, the server must be restarted. This cycle continues until the servlet is ready for production and deployed to the production server.

Using VisualAge for Java, you can develop and test servlets using the Java Servlet Development Kit (JSDK) or the WebSphere Test Environment.

To use the JSDK to develop servlets you run the HTTPServer class (that comes with the JSDK) within the VisualAge for Java environment. The HTTPServer class is a minimal Java Web Server that handles HTTP requests for servlets. It does not serve HTML documents or JSP files. To test the

servlet you load the servlet's URL in a Web browser, `http://localhost:8080/servlet/HelloWorldServlet`, for example.

Debugging the servlet is simply a matter of placing a breakpoint in the code that handles the request and reloading the URL to generate another request. This causes the debugger to be activated, and at this point you can step through the code, inspect any variables, and make any desired code modifications. If you make any code modifications, the new code is invoked by the `HTTPServer` class. You can then resume execution to see the results of our code changes. Note: Be careful when modifying code in the debugger when working with VisualAge for Java Version 2. There is a bug which can cause the Workspace to become corrupt.

4.5.2 WebSphere Test Environment

While VisualAge for Java is a powerful servlet development environment, servlets only represent one architectural component of a Web application. A Web application also includes other resources such as HTML documents and JSP files. As stated earlier, the `HTTPServer` class only handles HTTP requests for servlets. It does not serve HTML documents or JSP files. Developing, testing, and debugging a Web application that incorporates all these components is a major challenge.

The WebSphere Test Environment is a version of the WebSphere Application Server that provides an execution environment for testing Web applications. In addition to supporting HTML requests for servlets, as is the case with the `HTTPServer` class, it serves both HTML documents and JSP files.

We used the WebSphere Test Environment to develop, test, and debug the components of the Home Banking Application within VisualAge for Java, including the HTML documents, JSP files, servlets, and business objects.

Using the WebSphere Test Environment

The WebSphere Test Environment environment is part of the VisualAge for Java Enterprise or Professional Update available from www.software.ibm.com/vadd, the VisualAge Developers Domain Web site.

Follow the installation instructions that come with the update and add the WebSphere Test Environment feature to VisualAge for Java. During the installation, you will be prompted for the document root directory, where your Web resources, including HTML documents and JSP files, reside. If your Web server is on a different machine, you need to copy or map the document root directory from your Web server to the machine on which you are installing the WebSphere Test Environment.

Once the WebSphere Test Environment is running, you can serve your HTML documents and JSP files from this document root. The document root can be changed after install by modifying the doc.properties file in the httpservice directory. Figure 45 shows how the WebSphere Test Environment works with the first request for a JSP page.

VisualAge for Java

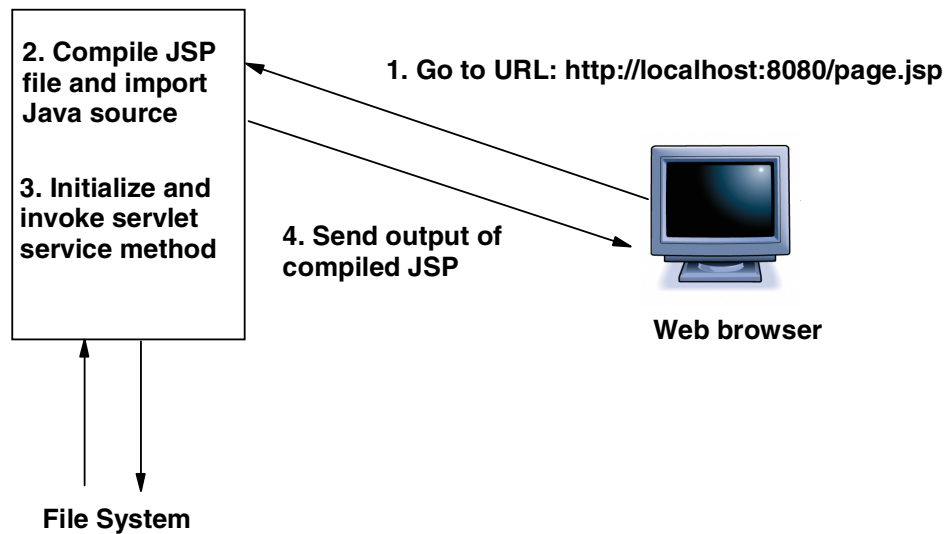


Figure 45. The WebSphere Test Environment

Prior to testing a Web application, the WebSphere Test Environment must be started. This is done by running the `com.ibm.servlet.SERunner` class (Figure 46) located in the WebSphere Test Environment project. You must also add your project to the SERunner's classpath. The SERunner listens on port 8080 by default.

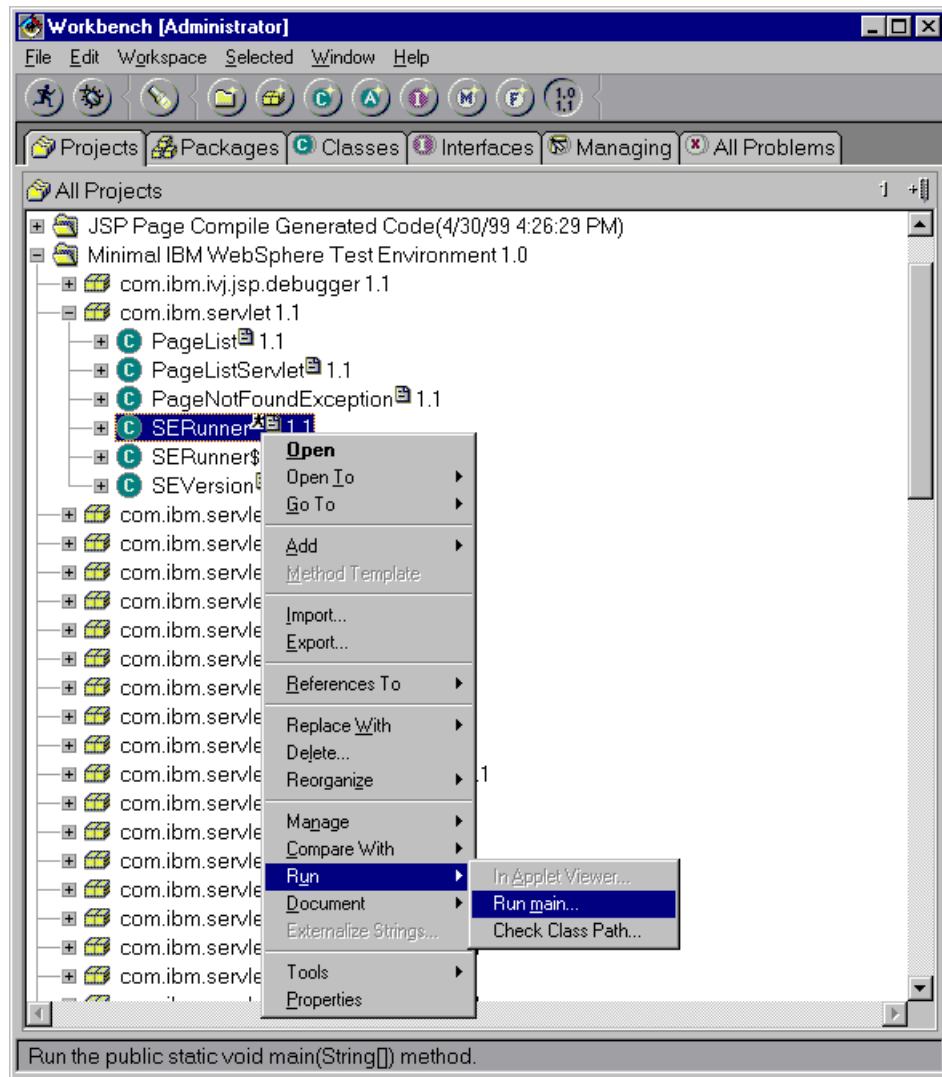


Figure 46. Launching the WebSphere Test Environment

If the WebSphere Test Environment has been successfully launched, you will see a WebSphere Test Environment window (Figure 47) and within the console window the message `endpoint.main.port=80` displayed twice (Figure 48).

You may get the following error message when loading JavaServer Pages in the WebSphere Test Environment:

Error getting compiled page.

Internal Error: Cant load page compiled class {0}: {1}.

If you get this error message, check that the JSP Page Compile Generated Code project is added to the SERunner's classpath; and if you are using the VisualAge for Java Enterprise Edition, make sure that you have authority to create packages in this project.



Figure 47. WebSphere Test Environment Window

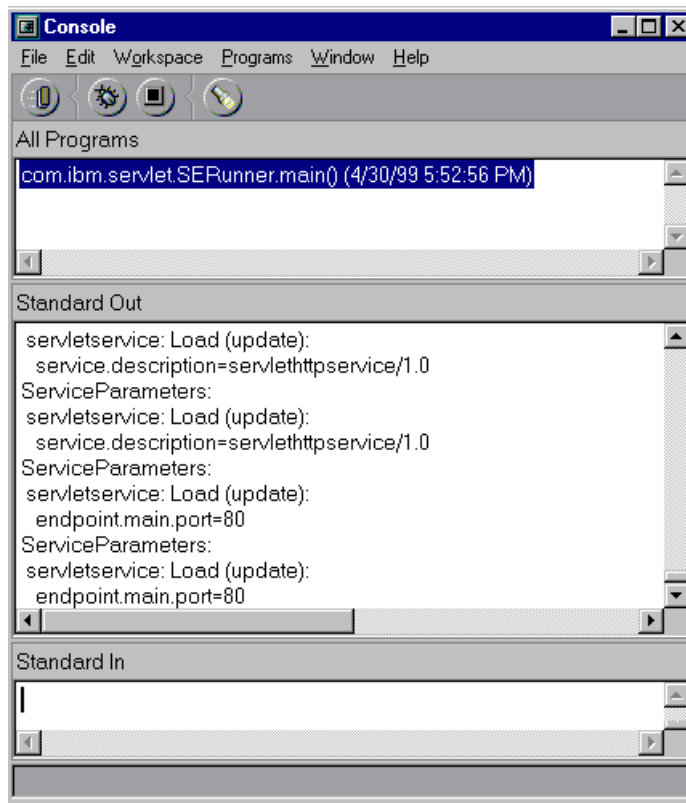


Figure 48. WebSphere Test Environment Output to Console Window

Running Internal and External Servlets

If you are using the WebSphere Test Environment, you probably want to run servlets that reside in the VisualAge for Java environment in order to use the VisualAge for Java debugger to step through problem code. You can also invoke external servlets, which reside outside the VisualAge for Java environment. To run an external servlet it must be located in the WebSphere Test Environment servlets directory: <install dir>\ide\project_resources\IBM WebSphere Test Environment\servlets. However, with external servlets you cannot debug and step through the code inside VisualAge for Java. In addition, any classes referenced by the servlet must be in a directory that is in the VisualAge for Java workspace classpath. Note that the WebSphere Test Environment does not automatically reload updated external servlets, so you will have to stop and restart it when making changes to your servlet.

If your servlet resides in both the VisualAge for Java environment and in the WebSphere Test Environment servlets directory, the external servlet will be invoked.

4.5.3 JSP Execution Monitor

While the WebSphere Test Environment enables you to test a Web application, including HTML documents and JSP files, the JSP Execution Monitor gives you finer control in monitoring, testing, and debugging your JSP source.

The JSP Execution Monitor simplifies testing and debugging of your JSP source by allowing you to detect run-time errors and syntax errors and step through your JSP code. You can dynamically make modifications and reload the JSP file.

Using the JSP Execution Monitor

Before monitoring the execution of your JSP files, you must have already started the WebSphere Test Environment (“Using the WebSphere Test Environment” on page 73).

To monitor the execution of your JSP files, you need to start the JSP Execution Monitor. Select **Workspace**→**Tools**→**JSP Execution Monitor** (Figure 49) to open the JSP Execution Monitor’s Option dialog (Figure 50). In this dialog, you specify the port number the JSP Execution Monitor uses, whether to monitor execution of the JSP source, and whether to report the types of syntax errors in the JSP source. By default, the port number is 8082, which can be changed if it is already in use.

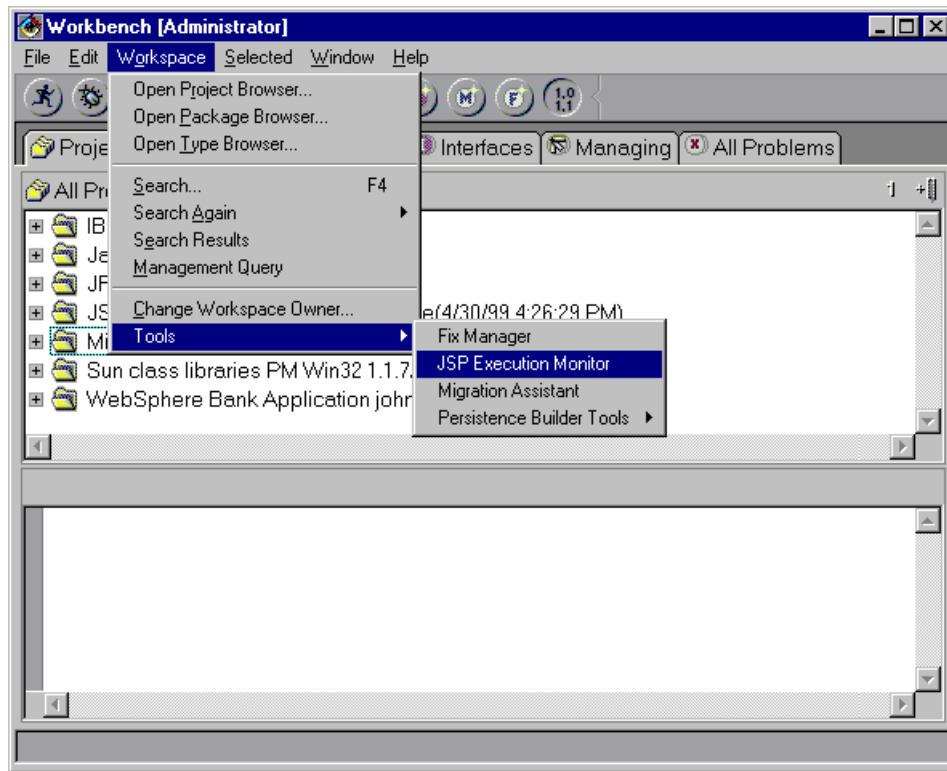


Figure 49. Launching the JSP Execution Monitor

Monitoring the Execution of Your JSP

To enable monitoring execution of our JSP source, you need to select the **Enable monitoring JSP execution** checkbox and click **OK** to begin monitoring (Figure 50).

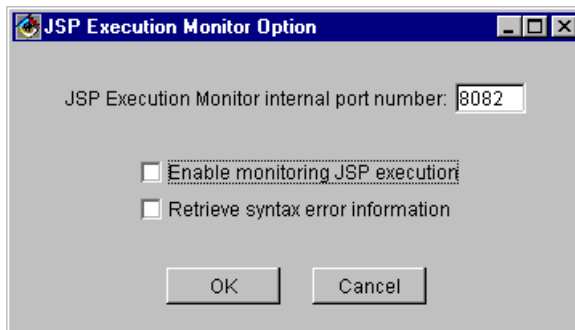


Figure 50. Options Dialog for JSP Execution Monitor

Now you can monitor the execution of your JSP source by pointing your browser at the JSP or navigating through your site until you reach the JSP page (Figure 51).

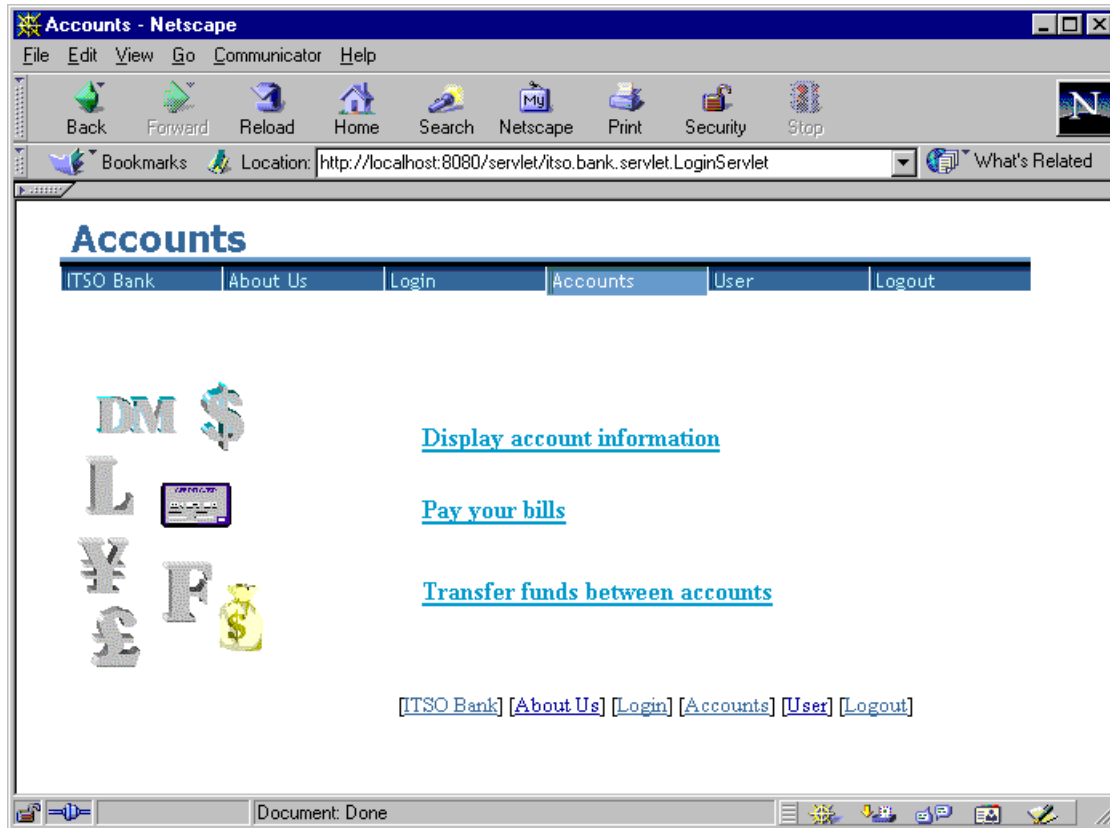


Figure 51. Loading a JSP for Monitoring

The JSP Execution Monitor window opens as the JSP file is loaded (Figure 52). You will see up to four panes displayed:

- JSP File List—JSP files that have been launched in the browser.
- JSP Source—The JSP source code for the running JavaServer Page.
- Java Source—Java code that is generated from the JSP source.
- Generated HTML Source—Generated HTML output.

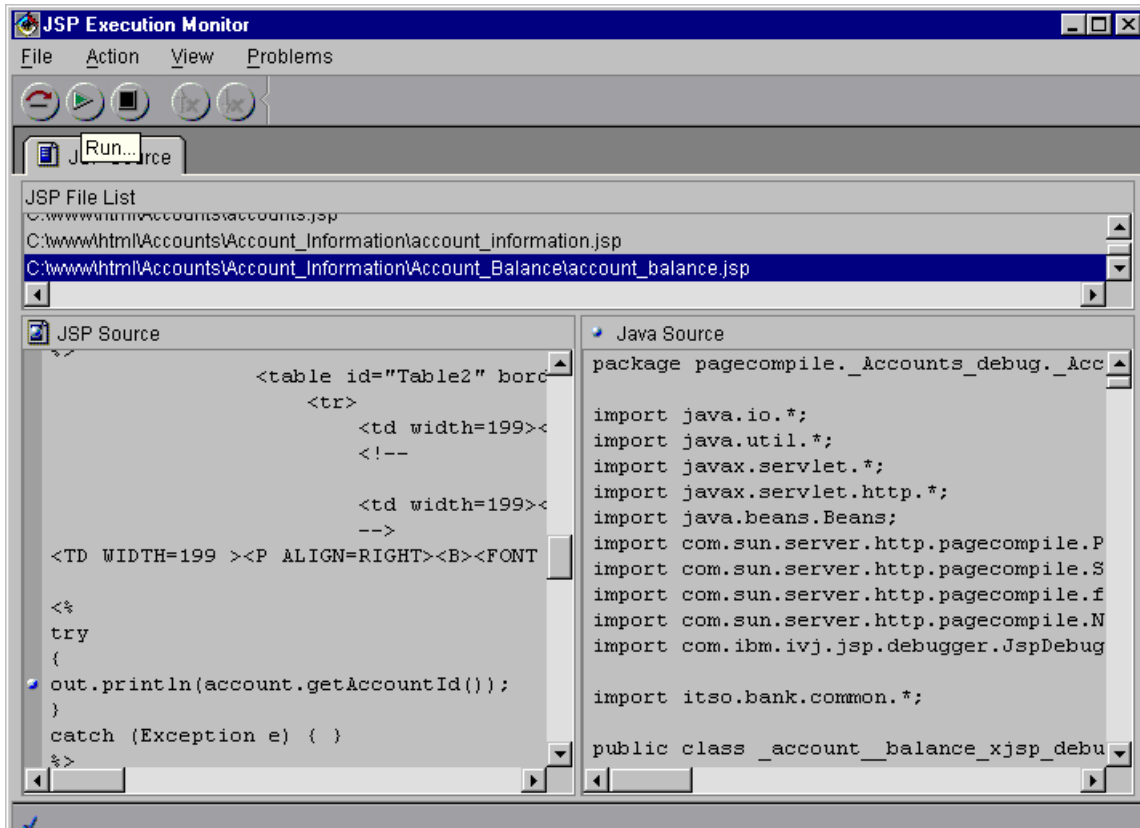


Figure 52. The JSP Execution Monitor

It is not required to have all four panes opened at any one time. You can toggle them on or off, depending on what information you want displayed in the JSP Execution Monitor. In our application, we did not display the Generated HTML source pane.

To step through your JSP source, select the JSP file in the JSP File List, then click the **Step** button from the toolbar, or alternatively, the **Action→Step** menu option. As you step through the code, you will see the JSP source highlighted in the JSP source pane. You will also see the equivalent generated Java source in the Java Source pane.

In addition, you can insert breakpoints in the Java Source pane, then press the **Play** button on the toolbar to resume execution of the JSP up to the next breakpoint. If you press the Play button without having inserted a breakpoint, you will step all the way to the end of the JSP file. If you want to finish executing the JSP source without stepping through each line or stopping at a

breakpoint, just press the **Terminate** button from the toolbar to resume execution.

The user interface of the JSP Execution Monitor changes if you install the Professional or Enterprise Update for VisualAge for Java Version 2. You also get a Fast Forward button in your JSP Execution Monitor, which will execute the JSP (without stepping) until the end of the page or the next breakpoint.

Retrieving Syntax Error Information

In addition to monitoring run-time errors (“Monitoring the Execution of Your JSP” on page 78), it is also useful to be able to monitor syntax errors in your JSP files. It is possible for the JSP Execution Monitor to detect syntax errors even when the Retrieve syntax information option is disabled. However, it will only tell you that a syntax error has occurred, not the type of syntax error. Select the **Retrieve syntax error information** checkbox when launching the JSP Execution Monitor (Figure 50 on page 78) to see details of the syntax errors.

In order for the Retrieve syntax error information to function, the servlets and referenced classes must exist within VisualAge for Java as well as outside the environment in the VisualAge for Java classpath.

After the Retrieve syntax error information has been selected and a JSP file is loaded that has syntax errors, the JSP Execution Monitor will display the syntax error in the status line (Figure 53). There are two types of syntax errors, JSP and Java syntax errors. JSP syntax errors, as the name suggests, are errors in the actual JSP syntax, while Java errors are errors in the generated Java source. The type of syntax error is displayed in the status line located at the bottom of the JSP Execution Monitor window.

For example, the `accounts_balance.jsp` has a Java syntax error in the form of a missing semi-colon at the end of a statement (Figure 53). If you have multiple syntax errors, you can step through them by selecting **Problems**→**Previous Problem** or the **Problems**→**Next Problem** menu option as shown in Figure 54.

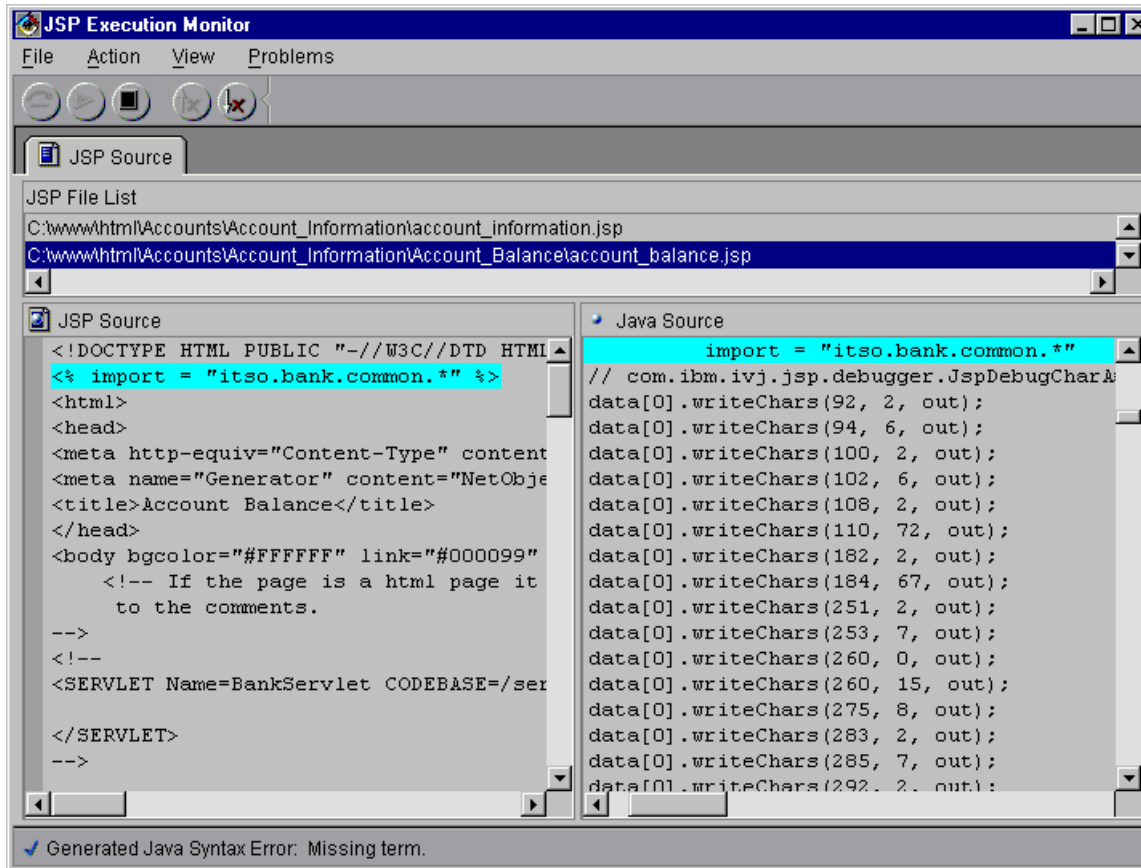


Figure 53. JSP Syntax Error in the JSP Execution Monitor

If you attempt to load a JSP file that has either a JSP or Java syntax error, and you have launched the JSP Execution Monitor without selecting Retrieve syntax error information, the JSP Execution Monitor will not launch.

In our application, we chose to deselect the Retrieve syntax error information option after the first couple of invocations of a JSP file. By this point, we were confident there were no syntax errors in the JSP file and did not want the extra overhead.

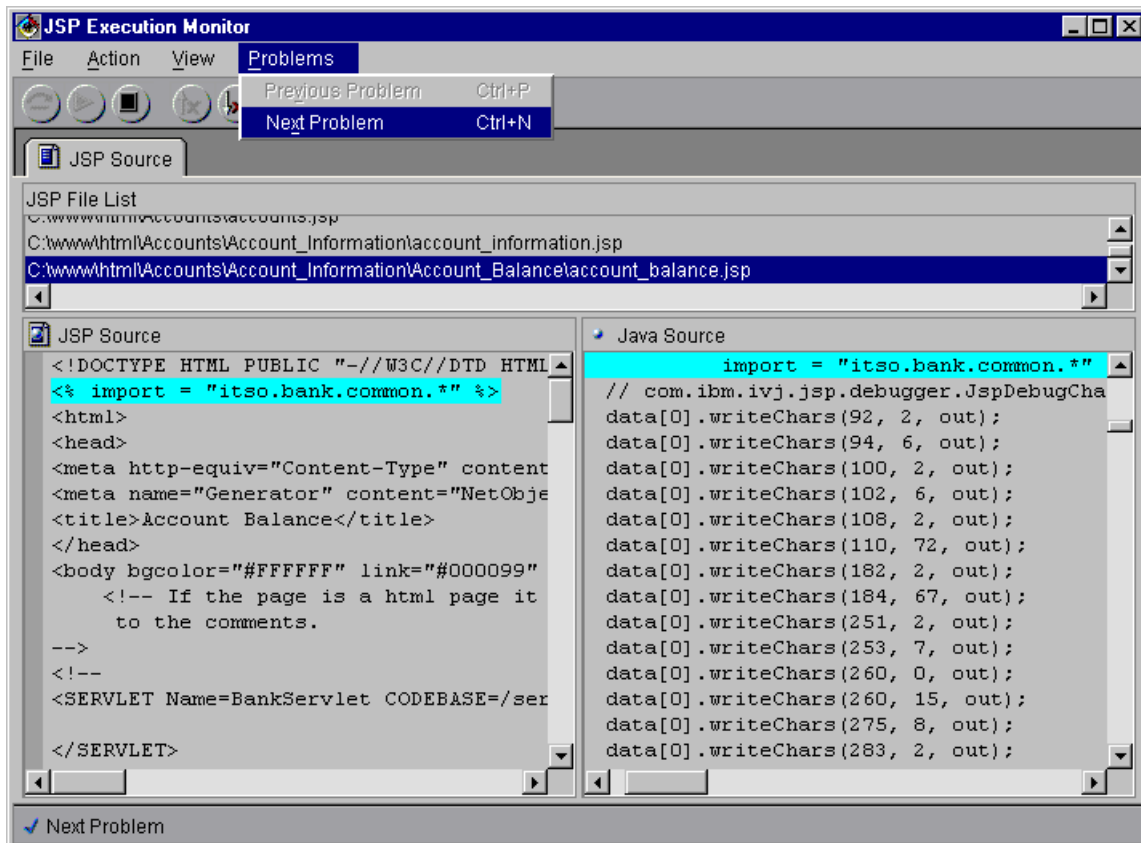


Figure 54. Stepping Through Syntax Errors in the JSP Execution Monitor

JSP Generated Code in VisualAge for Java

When a JSP file is loaded, it is compiled outside the IDE by a page compiler into a servlet then imported into the IDE. Generated servlets are placed into the JSP Page Compile Generated Code Project in the workspace. The name of the servlet and its package varies, depending on whether or not the JSP Execution Monitor was enabled when the JSP file was loaded.

If the JSP Execution Monitor was enabled, the generated code will be placed in the `pagecompile._<JSP directory name>_debug.<JSP file name>_debug` package. The generated servlets will have the name `<JSP file name>_jspx_debug`, and extend `JspDebugHttpServlet` because they contain extra debugging information. `JspDebugHttpServlet` comes with the WebSphere Test Environment (it is located in the `com.ibm.ivj.jsp.debugger` package) and extends `HttpServlet`.

If the JSP Execution Monitor was disabled when the JSP was loaded, the package name is <JSP directory name>.<JSP file name>. The generated servlet is named <JSP file name>_xjsp and extends HttpServlet.

Figure 55 shows the generated servlets for the JSP files, accounts.jsp, account_information.jsp, and account_balance.jsp with both the JSP Execution Monitor enabled and disabled. The generated servlets extending HttpServlet directly were loaded with the JSP Execution Monitor disabled, while conversely, those loaded with the JSP Execution Monitor enabled extend JspDebugHttpServlet.

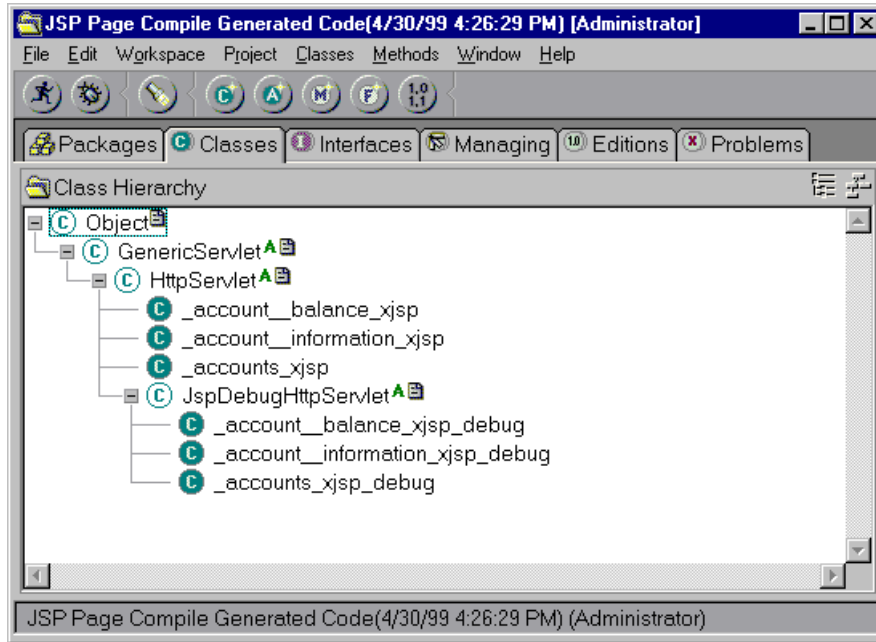


Figure 55. JSP Generated Servlets

Disabling the JSP Execution Monitor

Once you are satisfied with your JSP source, you probably want to disable the JSP Execution Monitor for faster compilation and better performance when testing your application. To do this, simply start the JSP Execution Monitor ("Using the JSP Execution Monitor" on page 77) and deselect the options. You can still debug servlets, including the servlets generated from JSP files, without the JSP Execution Monitor.

4.6 Application Server: WebSphere Application Server

The IBM WebSphere Application Server is a Java based application environment for building, deploying and managing Internet and intranet Web applications. This complete set of products expands to fit your Web application server needs, ranging from the simple to enterprise level applications. The WebSphere Application Server has three editions:

WebSphere Application Server Standard Edition

The Standard Edition includes the following features:

- Simple installation
- High performance
- A function-rich IBM HTTP server (based on technology from the Apache HTTP server) with additional SSL-based security and performance features
- Support for Lotus Domino Version 5.0
- Tivoli-ready modules
- Enhanced administration
- XML Document Structure Services
- Works with most popular Web servers

WebSphere Application Server Advanced Edition

The Advanced Edition contains all the features of the WebSphere Application Server Standard Edition, as well as:

- CORBA support, enhanced to provide both bean-managed and container-managed persistence
- Enterprise JavaBeans Server, providing relational database transaction management and monitoring based on Enterprise JavaBeans and CORBA components

WebSphere Application Server Enterprise Edition

The Enterprise Edition includes all features of the WebSphere Application Server Advanced Edition, as well as:

- TXSeries support, IBM's world-class transactional application environment
- Component Broker support, with its full distributed object and business process integration capabilities

For additional information, go to the web site:

<http://www.software.ibm.com/webservers>

For our project, we only needed the capabilities of the WebSphere Application Server Standard Edition. The rest of this book refers to the Standard Edition when WebSphere Application Server is mentioned.

4.6.1 WebSphere Application Server Architecture

The IBM WebSphere Application Server is built on the services of a Web server to provide additional services to support business applications and transactions on the Web. It provides support for serving static HTML as well as dynamic content for industrial-strength business applications. It can also use a set of connectors to act as a gateway to an existing legacy application. Figure 56 shows the WebSphere Application Server architecture. The server is built to work with the most common industry standards.

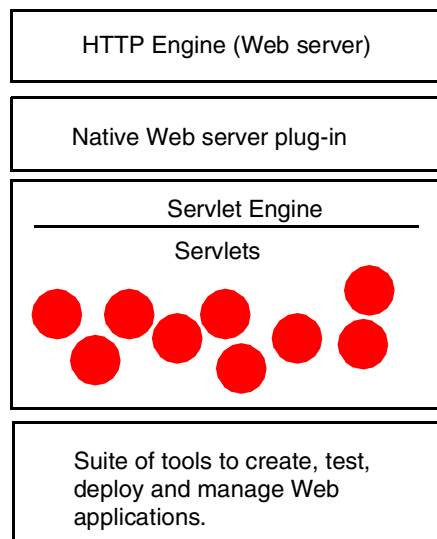


Figure 56. WebSphere Application Server Architecture

The WebSphere Application Server provides the following services:

- HTTP Engine

The HTTP engine is a Web server that handles HTTP Web requests: requests for static resources such as GIF files and HTML files, requests for CGI programs, and requests for plug-in applications. Servlet requests are passed on to the servlet engine after undergoing Web server authentication.

The Web servers we used for the HBA application were the Netscape Enterprise Server on Windows NT and the IBM HTTP Server on NT and AIX.

- Native Web Server Plugin

The native Web server plugin allows WebSphere to be connected to many Web servers using their proprietary protocol, such as ISAPI to connect to Microsoft IIS and NSAPI to connect to Netscape servers.

- Servlet Engine

The servlet engine is used to process dynamic content. It provides a facility for servlet management and supports JavaServer Pages. The server also comes with built in servlets for remote administration and page compilation.

4.6.2 WebSphere Implementation of JavaServer Pages

WebSphere Application Server 2.0 supports a modified version the JSP 0.91 specification. A 0.92 specification has been released, but the WebSphere Application Server will move directly to the JSP 1.0 specification sometime after it is finalized.

The JSP 1.0 specification will change the names of some of the tags, for example, the BEAN tag will be called `jsp:usebean`. The tags described in this book are from the WebSphere Application Server 2.0 implementation of the JSP 0.91 specification.

For more complete information on the JSP implementation in WebSphere, see the WebSphere documentation.

4.6.3 Managing Your WebSphere Environment

Once you have installed and started WebSphere Application Server, you will need to configure it. The WebSphere Application Server Administration Tool makes it easy to:

- Manage servlets
- Debug and monitor servlets
- Manage connections to databases
- Manage sessions

To go to the Administration Tool, direct your browser to `http://hostname:9527`, where `hostname` is the TCP/IP hostname of your WebSphere server. This takes you to the login screen of the WebSphere Application Server Manager (Figure 57).

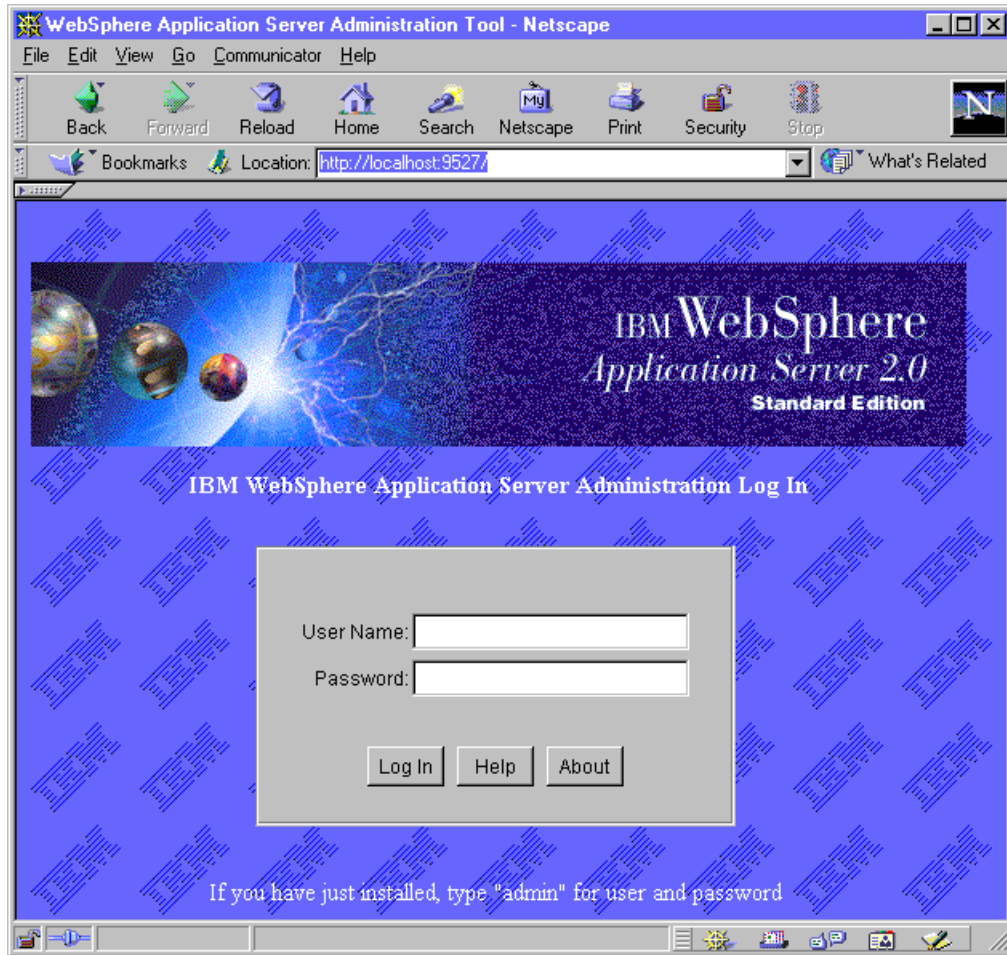


Figure 57. WebSphere Administration Console

Enter your User Name and Password (by default it is admin/admin) and click **Log In**, and the Introduction page of the WebSphere Application Server Manager (Figure 58) will be loaded into your browser.

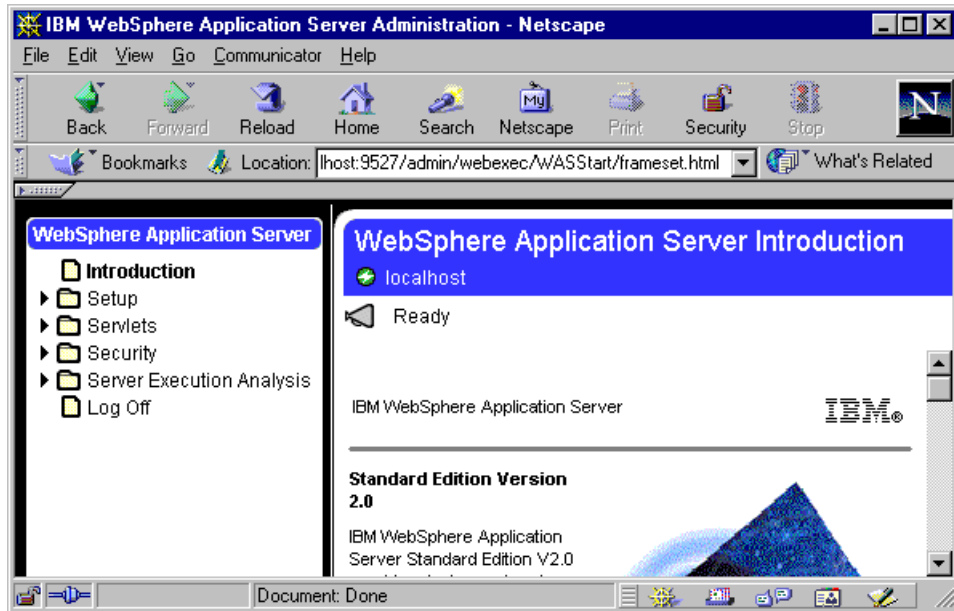


Figure 58. WebSphere Application Server Manager Introduction

Servlet Management

Once you have logged into the WebSphere Application Server Manager and are at the Introduction page (Figure 58), click on the arrow next to Servlets in the left pane and the Servlets list is expanded. Under Servlets there are three options: Configuration, Aliases and Filtering. When you click on Servlet Configuration, you go to the Servlet Configuration Section (Figure 59).

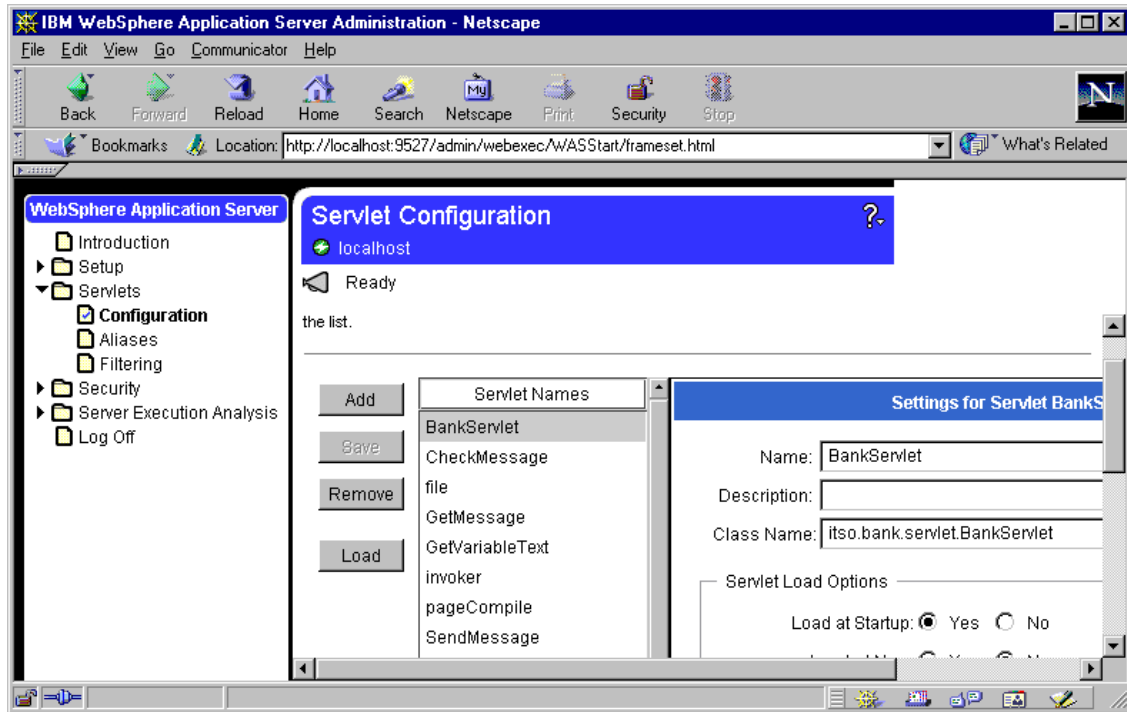


Figure 59. Servlet Configuration under WebSphere

The Servlet Configuration section is used to add, configure, and remove servlets from the WebSphere application environment. You can set up servlets so that they are loaded on server startup, and set initialization parameters for the servlet. The Servlet Aliases section (Figure 60) defines names for servlets and series (chains) of servlets.

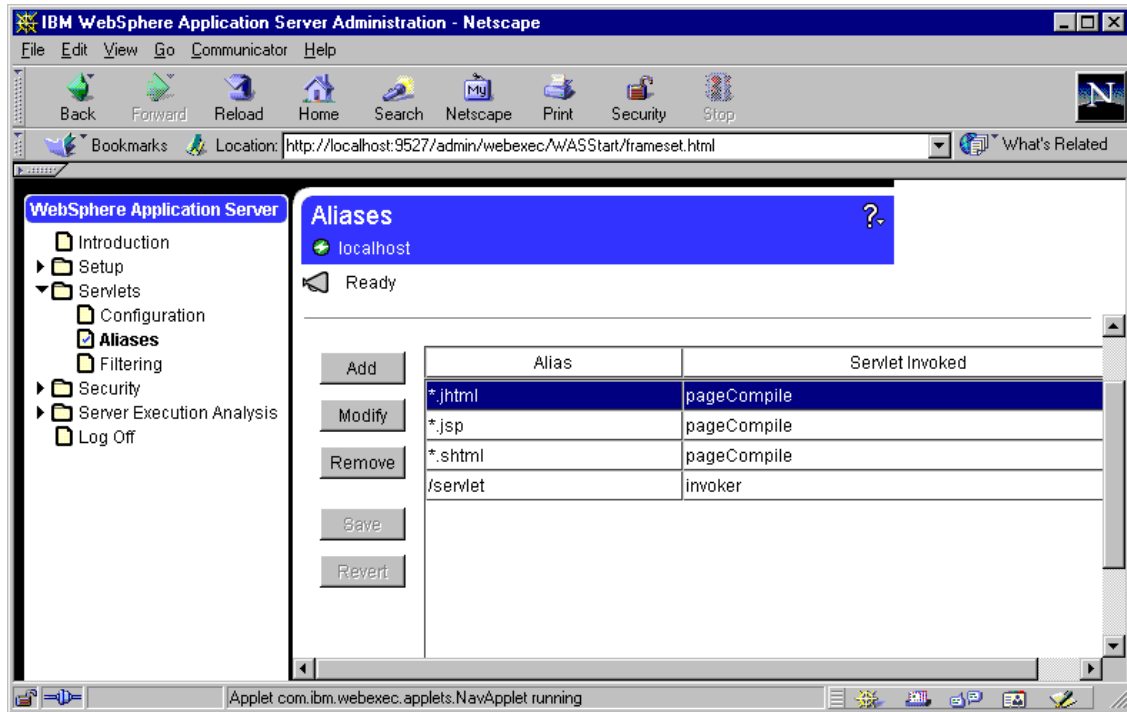


Figure 60. Servlet Aliases in WebSphere

The Servlet Filtering section allows for a specific mime type to be associated with a servlet. Once the mime-type is requested from the Web server, the page is filtered through the servlet that is mapped to the mime-type. For instance, the pageCompile filter is defined for java-internal/parsed-html. All parsed-html files must go through this servlet before being served out to the client. This capability can be used to filter content of a particular mime-type through certain servlets. This technique can be used to control the data that is presented to the client (Figure 61).

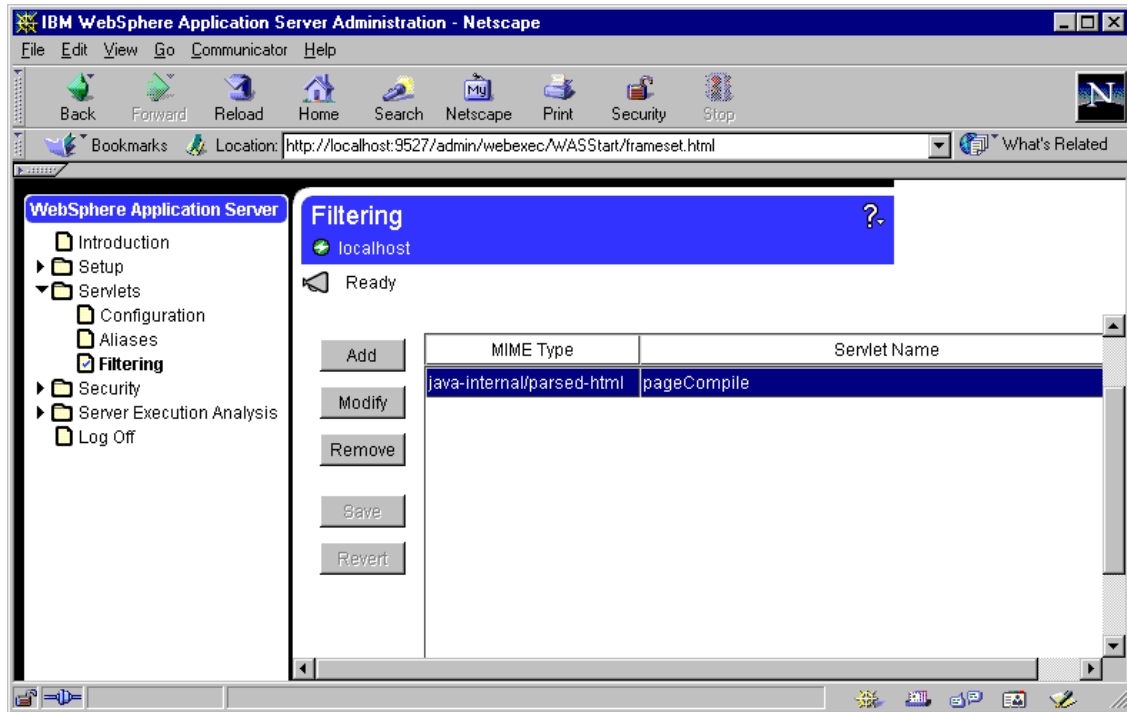


Figure 61. Servlet Filtering in WebSphere

Debugging and Monitoring

The WebSphere Application Server provides powerful debugging and monitoring capabilities to monitor details about specific areas of the server. It provides monitors for system administrators as well as for developers to track down any problems in their servlets. The Debugging and Monitoring Section can be found under Server Execution Analysis. It consists of: JVM Debug, Log Files, Monitors and Trace.

The JVM Debug (Figure 62) section allows the administrator to turn on the debugging of the JVM execution environment within WebSphere. This puts information into the log files about garbage collection and the classes being loaded, that information enables a developer to monitor the execution of instructions within the JVM in WebSphere. This is very useful when looking for memory leaks.

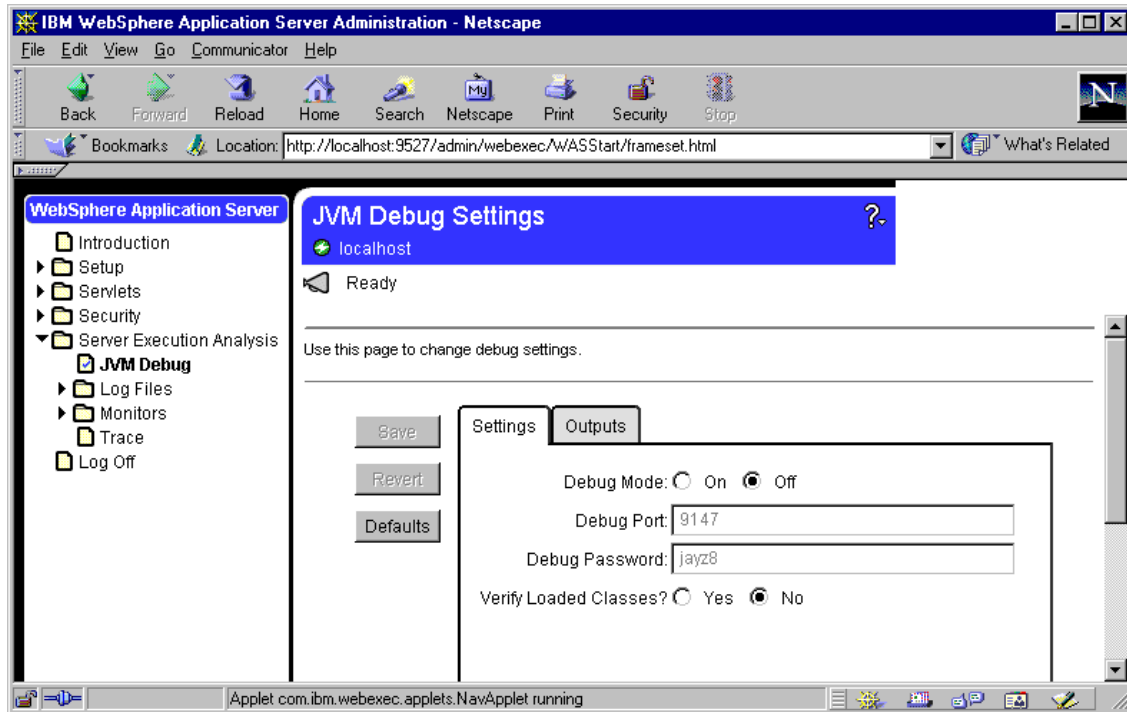


Figure 62. JVM Debug Settings in WebSphere

The Log Files section controls how the log files should be managed, what should be put into them, and whether to create a new file every day or to append to a master log file.

The Monitors section in WebSphere is the most important area for debugging. It provides monitors for sessions, database connections, exceptions, loaded servlets, and log files. The Active Sessions monitor shows all the active sessions connected to your Web server and the page of your site a user has currently loaded (Figure 63).

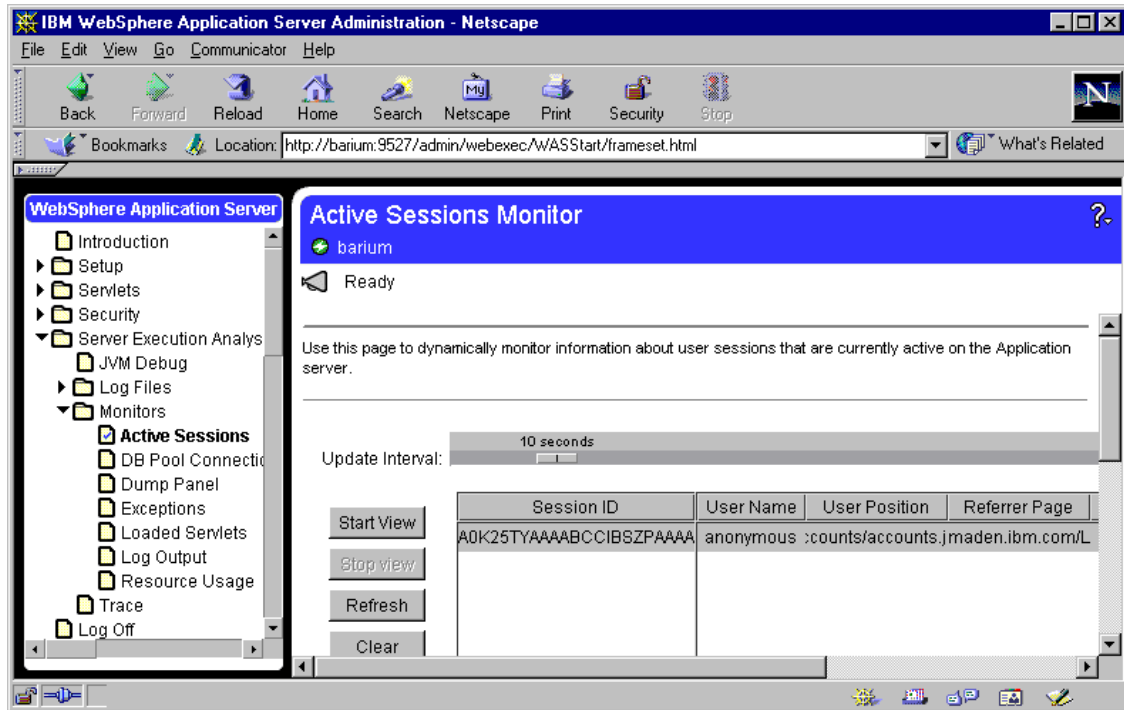


Figure 63. Active Session Monitor in WebSphere

The Resource Usage Monitor shows the resources being used by the server. This allows you to monitor the peak traffic hours of your Web site, and the associated performance (Figure 64).

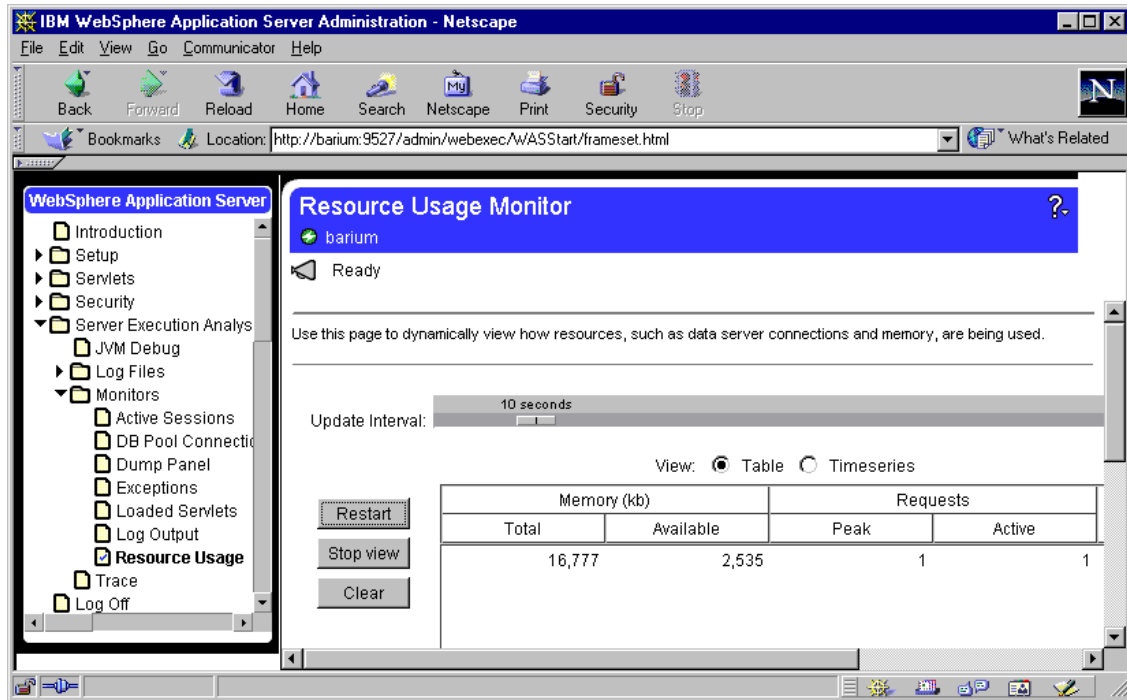


Figure 64. Resource Monitor in WebSphere

WebSphere provides the capability to pool database connections. The DataBase Connection Monitor allows you to monitor the status of database connections to databases (Figure 65).

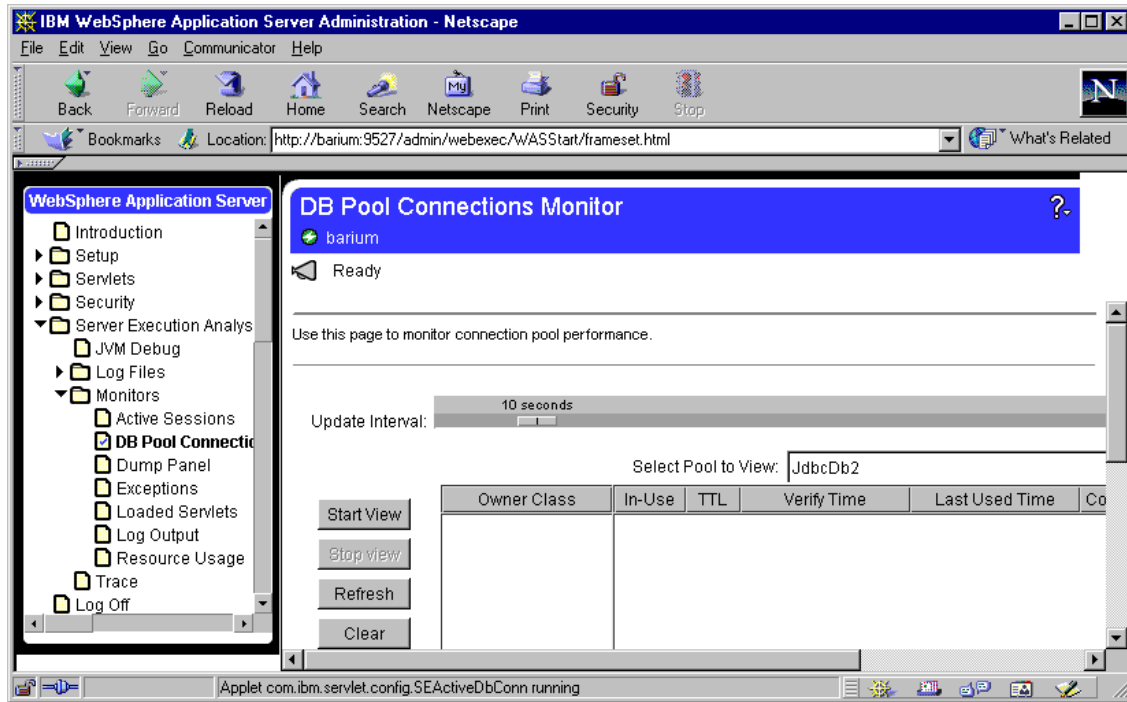


Figure 65. Database Connection Monitor in WebSphere

These are some of the more important monitors that WebSphere provides to monitor the status of your Web application. These tools along with the JSP Execution Monitor give you a complete environment for testing and debugging your application.

Connection Management to Databases

WebSphere Application Server provides connection management to databases and provides a central location to manage these connections and view the usage of those connections (Figure 66).

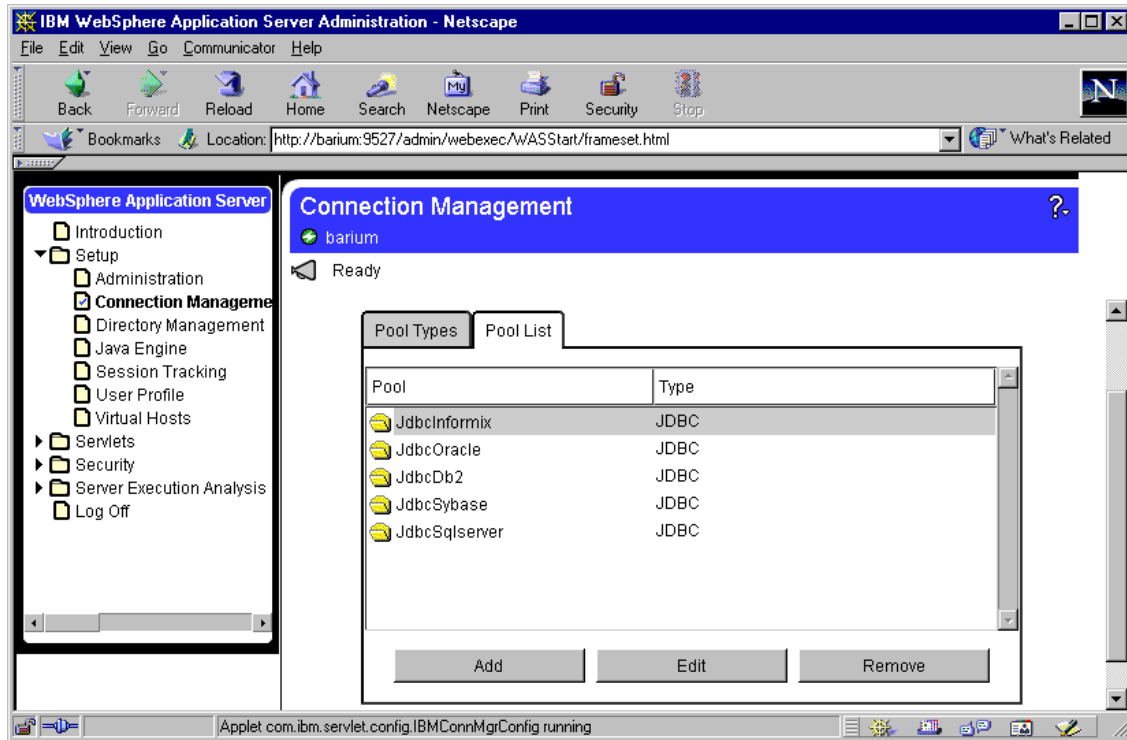


Figure 66. Connection Management in WebSphere

Session Tracking

The WebSphere Application Server provides session tracking: the ability to keep track of a user's state while they are browsing your Web site. This enables a Web application to act intelligently by responding based on a user's previous actions. You can store sessions in two ways in WebSphere: using cookies or URL rewriting, as discussed in 2.5, "Maintaining State in Web Applications" on page 16. You can also combine the approaches: cookies are used if the client supports them, otherwise URL encoding is used (Figure 67). The session tracking mechanism can also be configured to time out a users session if they have not been active for a specified amount of time.

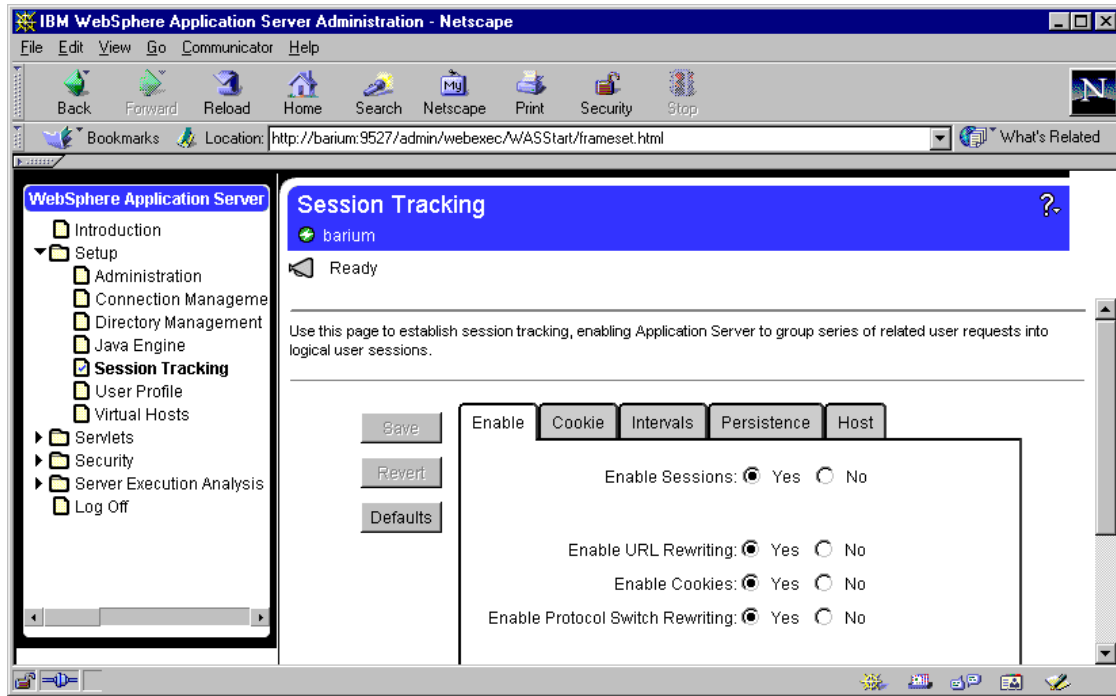


Figure 67. Session Management in WebSphere

Chapter 5. Implementing the Home Banking Application

In this chapter we discuss the implementation of the Home Banking Application (HBA).

The HBA implementation is composed of several subsystems:

- Domain Firewall
- Business Model
- Servlet/JSP Implementation (Web application)

The focus of this book is on the servlet/JSP implementation. The implementation of the business model and the domain firewall will be briefly explained so that the system as a whole is understood.

5.1 Implementing the Domain Firewall

In 3.6, “Subsystem Design” on page 29 we discussed the design of the domain firewall. Although in most cases the implementation would be created before or at least concurrently with the domain firewall, it may be easier to understand the system if the domain firewall is introduced first. The implementation classes (see 5.2, “Implementing the Business Model” on page 103) implement the Java interfaces defined in the firewall.

The implementation of the firewall consists of a set of Java interfaces and classes. These types provide access to all the functionality of the bank implementation as well as providing initial finder methods to locate and instantiate the bank implementation. This firewall should make it possible to substitute multiple implementations of the bank without changing any code in the firewall or Web application. In addition, the domain firewall made it possible to test the bank implementation using simple command line or Java clients without a Web application infrastructure.

The interfaces in the package are implemented by the appropriate type in the implementation package, providing a clear separation between the domain firewall (interface) and the implementation. All interaction between the Web application and the implementation goes through the interfaces.

The interface package is `itso.bank.common` and is composed of the following interfaces and selected methods:

Bank

- `getAccount`: Return the account associated with `accountID`.
- `getCustomerByUserId`: Return the customer by the `UserId`.
- `getPayeeAccounts`: Return all the payees registered with the bank.

BankAccount

- `getAccountType`: Return the account type associated with the account.
- `getBalance`: Return the balance of the account.
- `getHistory`: Return the transaction records which match the criteria specified by the parameters.
- `transfer`: Transfer the amount from this account to the `toAccount` parameter.

BankSystem

- `getBank`: Return the bank implementation.

CheckingAccount

- `getOverDraftLimit` - Get the allowed overdraft limit.

Customer

- `addPayee`: Add this account to the customer's bill payment profile.
- `checkLoginPassword`: Return true if this is the correct password for this customer.
- `checkTransactionPassword`: Return true if this is the correct password for this customer.
- `getAccountByID`: Return the account that has this account ID.
- `getAccounts`: Return all the non-payee (saving or checking) accounts owned by this customer.
- `getPayees`: Return all the payee accounts associated with this customer.
- `removePayee`: Remove this account from the customer's bill payment profile.
- `changeLoginPassword`: Change the login password.
- `changeTransactionPassword`: Change the transaction password.

PayeeAccount

- `getBillPaymentTitle`: Get the payee title (company name).

SavingsAccount

- `getMinimumBalance`: Get the minimum accepted balance for the account.

TransactionRecord

- `equals`: Compare two `TransactionRecords`.
- `getAccount`: Return the account number associated with the transaction.
- `getClosingBalance`: Return the balance following the transaction.

- `getOtherAccount`: Return the other account number associated with a transfer type transaction.
- `getTimeStamp`: Return the timestamp associated with the transaction.
- `getTransAmount`: Return the amount of the transaction.
- `getTransType`: Return the type of the transaction.

The package also contains the following classes which would be common to any implementation of the HBA:

BankCollection

An abstraction of `java.util.Vector` that provides a simple implementation independent means of passing vectors.

BankHome

Initial finder class. Provides a `create` method to create the bank implementation given an implementation key. In this implementation the key is simply the package name for the implementation.

- `create`: This static method builds the class name of the `BankSystem` class and creates an instance of it. The `BankSystem` in this implementation either creates a new bank with sample data or deserializes an existing bank.
- `getBank`: Return the bank. The bank is a singleton object created by the `BankSystem`.

The following exceptions are used in the HBA:

NotImplementedException

This exception is used during development to flag implementation areas not completed.

ITSOBankCommunicationException

Used if the implementation is distributed, for example, using an RMI based implementation.

ITSOBankException

- `BankTransactionException`—This exception is thrown, for example, if there are insufficient funds for a transfer.
- `UnauthorizedException`
 - `InvalidPasswordException`
 - `InvalidPinException`

The complete JavaDoc for the domain firewall and source for the HBA is on: <ftp://www.redbooks.ibm.com/redbooks/SG245423/>, the FTP site.

Although it is not part of this book, an administration interface was also created to make it easy to create new customers and accounts. The bank implementation supports this interface also. Figure 68 shows selected elements of the domain firewall as defined in Rational Rose.

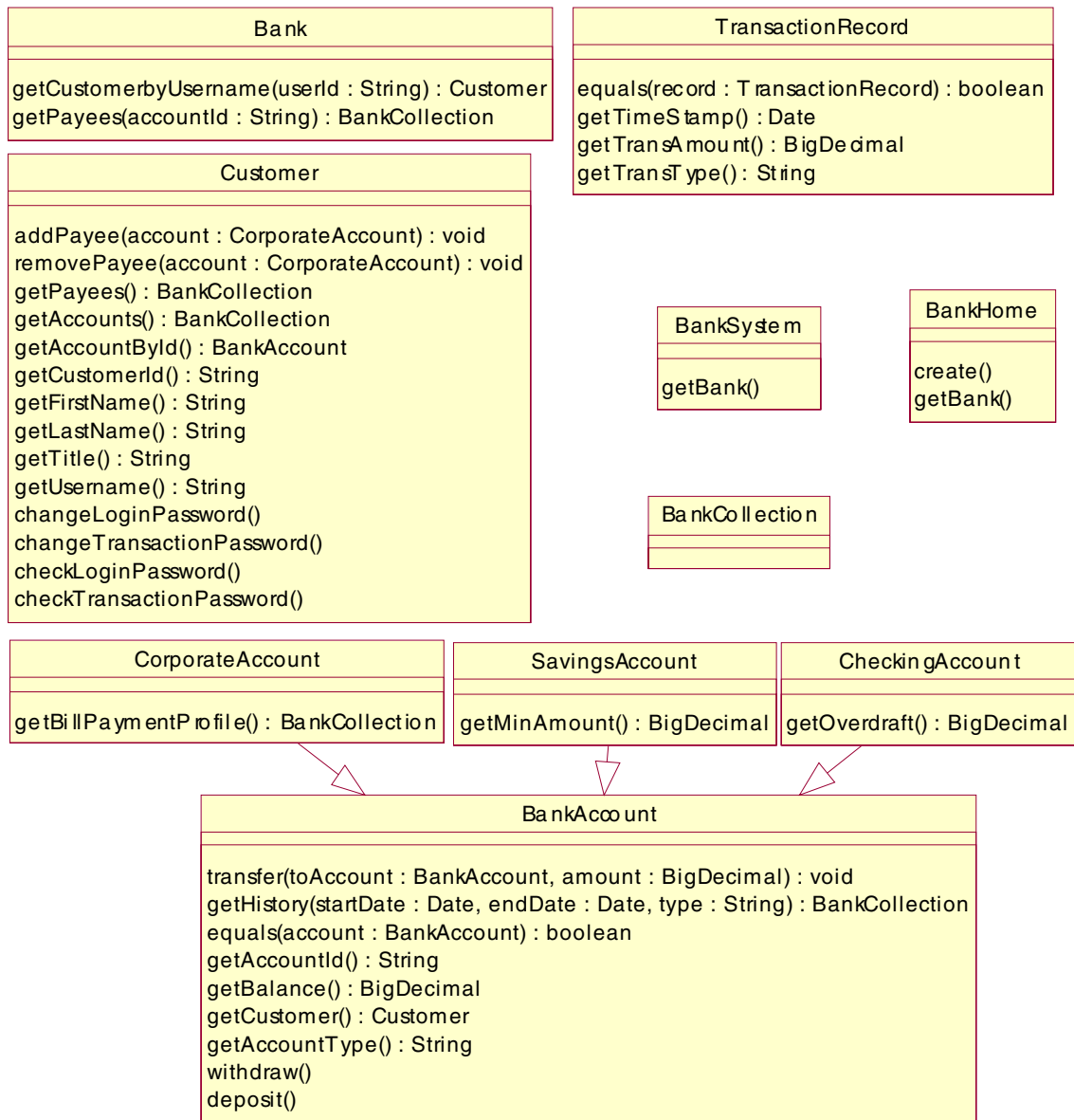


Figure 68. Selected Elements of the Bank Domain Firewall

5.2 Implementing the Business Model

As discussed in 3.5, “Analysis Object Model” on page 28, the business model for a banking application would probably be based on an existing legacy application. Since we did not have one, we built a simple Java bank that uses Java serialization for persistence. While serialization is an extremely brittle means of providing persistence and we do not recommend it for a production application, it was a simple and effective means for this application and allowed us to concentrate on the Web application itself.

The implementation package is `itso.bank.baseimpl` and it contains the following classes (shown in Figure 69):

- `BankAccountImpl`
- `BankImpl`
- `BankSystemImpl`
- `CheckingAccountImpl`
- `CustomerImpl`
- `PayeeAccountImpl`
- `SavingsAccountImpl`
- `TransactionRecordImpl`

Each class implements the corresponding interface from the `itso.bank.common` package.

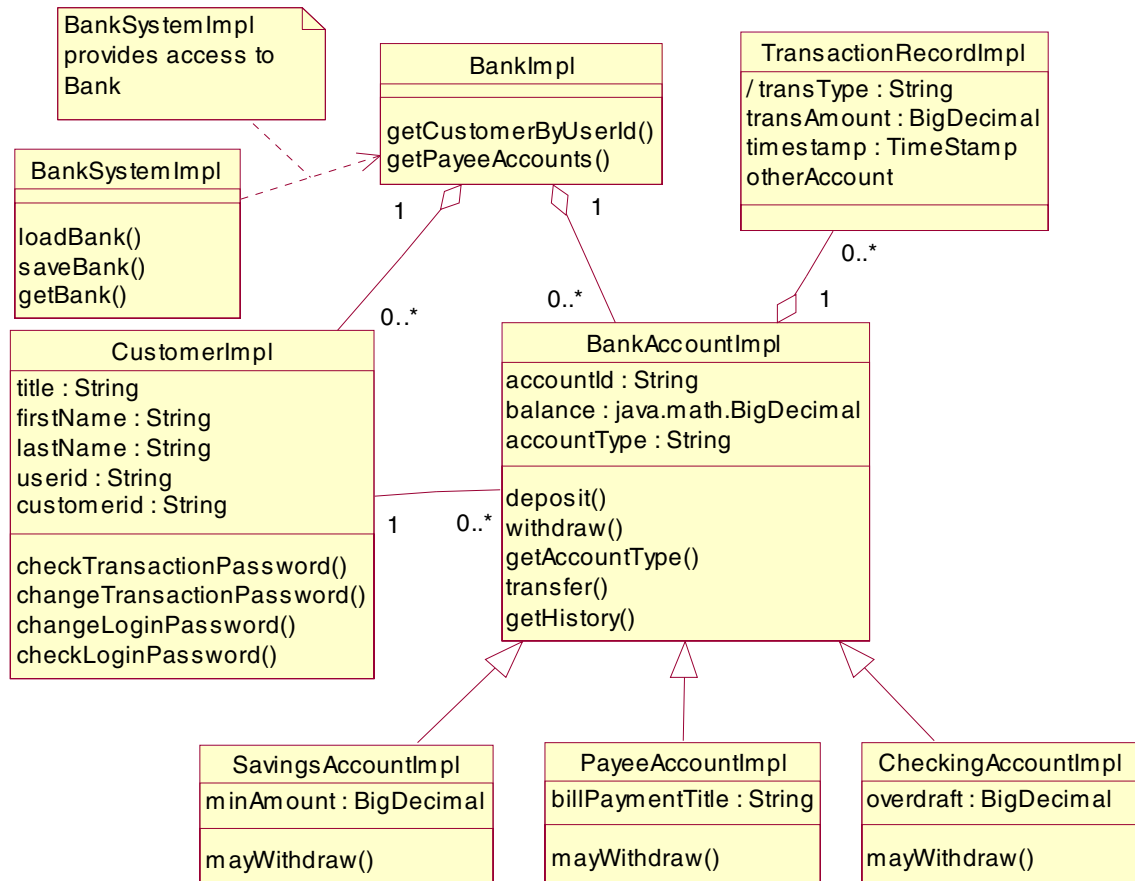


Figure 69. Selected Elements of the Rose Model of the Bank Implementation

5.3 Implementing the Web Application

In this section we discuss the general implementation issues for the Web application layer of the HBA and then describe each subsystem in detail. Figure 70 shows the complete implementation of the HBA.

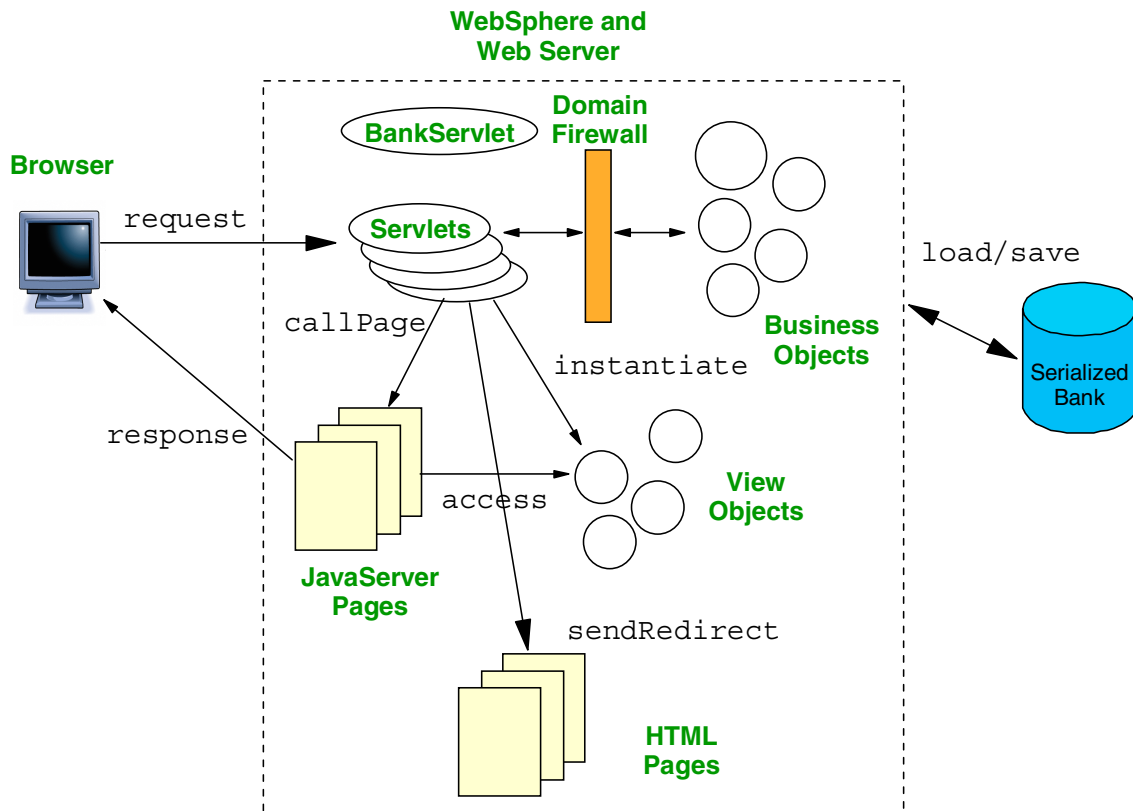


Figure 70. Complete HBA Implementation

5.3.1 General Implementation Issues

Servlets

The WebSphere Application Server provides a subclass of `HttpServlet` named `PageListServlet`. The `PageListServlet` adds some very useful function to the `HttpServlet`. The pathnames of `JavaServer Pages` accessed by the servlet can be specified in a separate XML configuration file. In this file you can specify a default `JavaServer Page` and an error page, and name any other pages called from the servlet. In this way you can change the names and locations of `JavaServer Pages` without changing code.

In the HBA we used the `PageListServlet` and added another utility that allowed us to use the XML configuration file to specify pages we accessed using the `sendRedirect` method. This utility is described in 5.12, "Utility Classes" on page 166.

The XML configuration file is named ServletName.servlet and is placed in the same directory as the servlet class file. An example configuration file is AccountServlet.servlet:

```
<?xml version="1.0"?>
<servlet>
<page-list>
  <error-page>
    <uri>/itso_bank_error.jsp</uri>
  </error-page>
  <default-page>
    <uri>/account_information.jsp</uri>
  </default-page>
  <page>
    <uri>/account_information.jsp</uri>
    <page-name>account_information</page-name>
  </page>
  <page>
    <uri>/account_history.jsp</uri>
    <page-name>account_history</page-name>
  </page>
  <page>
    <uri>/account_balance.jsp</uri>
    <page-name>account_balance</page-name>
  </page>
</page-list>
<code>itso.bank.servlet.AccountServlet</code>
</servlet>
```

For more information on servlet configuration files, see the WebSphere Application Server product documentation.

View Beans

In 3.8.3, “What Goes into a JavaServer Page?” on page 36 you learned that we decided to use view beans to encapsulate the data retrieved from the business objects. The view bean is a class that represents a data view of the real implementation object. The JSP will then extract whatever data it requires from the view using bean tags. Servlets interact with the implementation through the domain firewall and construct view beans from the implementation object for subsequent use by JSP’s.

There were two main approaches to creating the view beans:

1. The view class simply holds a reference to the implementation object and would delegate calls to the implementation object.

This approach presented a number of issues. Because each method in the interface specifies a throws clause, our view class has to specify the same throws clause for each method implemented, or enclose each method call in a try/catch block in the JavaServer Page. This meant using a combination of bean tags and scriptlets, where bean tags were used to get the properties from the view, with scriptlets used to place try/catch blocks around the bean tags. This approach also meant that an error was not detected until the bean was accessed in the JavaServer Page: probably too late to do anything about it.

2. Create a view class that does not implement the interface. What it has is a number of read-only properties that mirror those in the implementation object. In addition, it requires a default constructor to conform to JSP implementation, and a constructor that takes one argument, the actual implementation object. The second constructor sets the view bean's properties to those of the implementation using its get methods. Because the get methods can throw checked exceptions, the constructor also specifies the same throws clause. This meant that we could remove the throws clause from the view's get methods, eliminating the need to wrap the Bean tags in try/catch blocks and catching errors in the servlet when constructing the bean.

The use of view beans also meant that we could perform any necessary transforms between the model and the view when the view bean was constructed. In our case we made appropriate date and currency transforms and also converted BankCollections into arrays for access by the bean tags. Other applications could use these transforms to support multiple languages or personalization.

Using View Beans in the Accounts Subsystem

The BankAccountView class is used to encapsulate objects that implement the BankAccount interface: BankAccountImpl objects in the HBA. The class has four read-only properties, balance, accountId, accountType, and transactions that are accessed by their corresponding accessor methods:

```
public final String getAccountId() {
    return accountId;
}
public final String getAccountType() {
    return accountType;
}
public final String getBalance() {
    return balance.toString();
}
public final itso.bank.viewobjects.TransactionRecordView[]
getTransactions() {
```

```

        return fieldTransactions;
    }
    public final TransactionRecordView getTransactions(int index) {
        return getTransactions()[index];
    }
}

```

Constructors

The class has two constructors:

```

public BankAccountView( BankAccount account){}

public BankAccountView(BankAccount account) throws
ITSOBankCommunicationException,
ITSOBankException
{
    this.balance = Formatter.getAsCurrency( account.getBalance());
    this.accountId = account.getAccountId();
    this.accountType = account.getAccountType();
    BankCollection transactions = account.getTransactions();
    fieldTransactions = new TransactionRecordView[ transactions.size()];
    for( int i = 0; i < transactions.size(); i++){
        fieldTransactions[i] = new TransactionRecordView(
            (TransactionRecord)transactions.elementAt(i));
    }
}

```

The Formatter class is shown in 5.12.2, "Formatter" on page 167.

Using the View Objects

Here is an example using a view object in the AccountServlet (ignoring try/catch and error handling requirements). An account is retrieved from a customer and the account view is then inserted into the request object:

```

HttpSession session = req.getSession(false);
CustomerView customerView =
    (itso.bank.viewobjects.CustomerView) session.getValue("customer");
Customer customer = BankHome.getBank().getCustomerByUserId(
    customerView.getUserId());
BankAccount account = customer.getAccountByID(accountID);
((com.sun.server.http.HttpServiceRequest)req).
    setAttribute("account", new BankAccountView(account));

```

To access the balance property of the view object in a JSP page, the following code could be used:

```

<BEAN name="account" type="itso.bank.viewobjects.BankAccountView"
introspect="no" create="no" scope="request"></BEAN>
<INSERT bean="account" property="balance"></INSERT>

```

5.4 SubSystem Implementation

The subsystems in the HBA were introduced in 3.6, “Subsystem Design” on page 29. Each implementation is described in this section.

5.5 Application Manager

Once a customer has logged in to the HBA, other subsystems need to be able to verify that the customer has been authenticated. This is accomplished using sessions and the application manager. The Application Manager is implemented through the BankServlet, which also creates the bank implementation when it is preloaded by the WebSphere Application Server and handles the logout functionality (Figure 71).

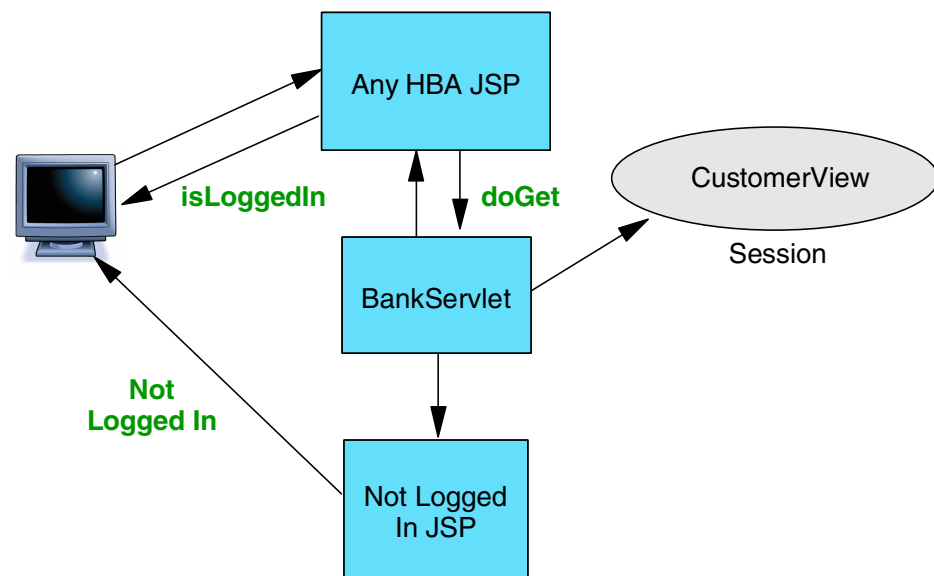


Figure 71. Application Manager - User Recognition

In WebSphere you can provide session management in Servlets and JSP pages quite easily by using the `HttpSession` class. When a user first visits your site, you create a new session by a call to the `HttpServletRequest` class:

```
HttpSession session = request.getSession(true);
```

This creates a new session that a user uses to navigate the Web site. As the user moves between pages, you use this session to maintain state information about the user. You can store information in a session by putting a key-value pair in the session:

```
session.putValue("Entry", "WebSphere")
```

This puts a key with a name of "Entry" in the session that stores a String object with the value of "WebSphere". You can retrieve this value from the session by requesting the value by its key.

```
String entry= (java.lang.String)session.getValue("Entry");
```

This takes the value for the key "Entry" and stores it in a String variable called entry. This mechanism provides a way to store information about the user. This technique is used in the Home Banking Application to store the CustomerView object. In most other cases, we used the HttpRequest object to store information, which is then accessed by a JavaServer Page. In this way, information was only accessed once when it was still valid. It also kept the information in the session to a minimum. Using the session to store many objects with indefinite lifetimes can use up large amounts of memory and should be avoided.

The only time we used the HttpSession object to store additional objects was when we used the sendRedirect method to redirect the response to a completely new page. This was required for responses to actions which modified the state of objects and which could be replayed by reloading the page. For example, after transferring money to another account pressing the Reload button on the browser could perform the transfer again unless the sendRedirect method was used.

Values are set in the HttpServletRequest object through the setAttribute method. For example:

```
Customer customer = BankHome.getBank().getCustomerByUserId(
    customerView.getUserId());
BankAccountViewList accountList =
    new BankAccountViewList( customer.getAccounts());
((com.sun.server.http.HttpServletRequest)req).
    setAttribute( "accountlist", accountList);
```

Access to the HttpSession objects should be synchronized so that several users are not accessing information simultaneously. In the HBA, we synchronized access in our business model when the state of an object was being modified. All other access is simply read access and did not need synchronization.

5.5.1 Application Manager Interaction

The BankServlet is loaded by the WebSphere Application Server on startup. At that time the servlet instantiates the bank implementation using the BankHome object (Figure 72).

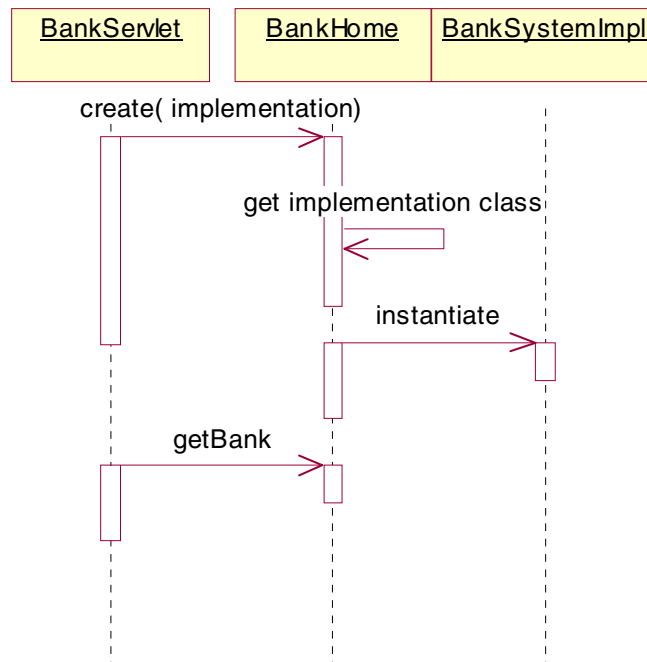


Figure 72. BankServlet init Method Sequence

The BankServlet is placed in each JavaServer Page of the HBA using the SERVLET tag:

```
<SERVLET Name=BankServlet  
CODEBASE=/servlet/itso.bank.servlet.BankServlet></SERVLET>
```

When the user requests a page that contains this line in it, the doGet method of the BankServlet is called (see “doGet” on page 114). This method checks to see if the user has logged in. If the user has not logged in, the Not Logged In page is sent to the user’s browser. This interaction is shown in Figure 73.

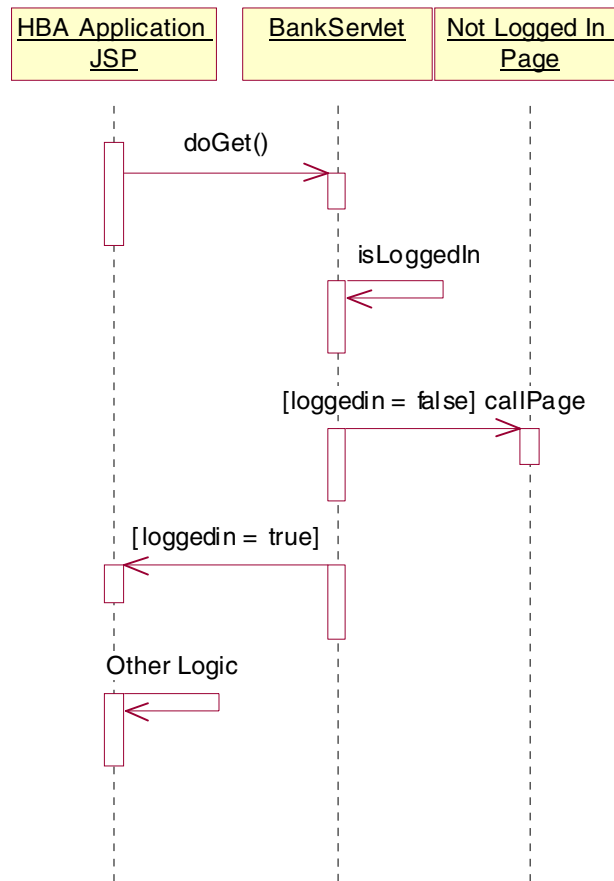


Figure 73. Session Management JSP/BankServlet Interaction Diagram

5.5.2 Application Manager Servlets

The servlet responsible for enforcing session management in the Home Banking Application is the BankServlet.

BankServlet

Due to problems with using preloaded servlets of type PageListServlet, the BankServlet extends HttpServlet and calls other pages using the URL rather than a value from the servlet configuration file.

The BankServlet checks if the user is logged in before allowing the user to access a page in the HBA. It does this each time the user accesses a page of the Web site. If the user is not logged in, the Not Logged In page is displayed.

If the request is for Logout.jsp, the BankServlet invalidates the session before passing control back to Logout.jsp. The BankServlet also initializes the Bank object during Web server startup, and on server shutdown the BankServlet is used to save information about the bank. Currently all information is saved as output to a serialized file.

Table 2 shows the BankServlet methods and Table 3 shows the collaborating objects.

Table 2. BankServlet Methods

Method	Description
init	Initializes the BankServlet and the bank implementation.
destroy	Sets the bank to null so that it will be finalized and serialized.
isLoggedIn	Checks to see if the user has a valid session.
doGet	Checks to see whether the customer needs to be logged in and whether they have a valid session. Invalidates the session if the request is for the Logout page.
doPost	Calls doGet.

Table 3. BankServlet Collaborators

Class	Description
BankHome	Used to initialize the Bank object. It returns the current Bank object if one is already created.
Bank	Provide bank services.
HttpSession	Manage user sessions.
HttpRequest	Access to pathInfo parameter to supply the target URL.
HttpResponse	Used to invoke sendRedirect.

BankServlet Error Handling

The BankServlet sends the user to the Not Logged In Page if they are not logged in and they are accessing a page which requires authentication. If the user is accessing the Login page itself and they are already logged in, the BankServlet sends the user to the Already Logged In page.

BankServlet Methods

doGet

The doGet method is invoked when the BankServlet is invoked from a JSP (Figure 73 on page 112). The doGet method checks whether the user is logged in and whether they are accessing the Login page. If the user is not logged in and not accessing the Login page, they are sent the Not Logged In page. If the user is logged in:

- If the request is for Login.jsp, the user is redirected to the Already Logged In page.
- If the request is for Logout.jsp, the user's session is invalidated and control is returned to the requesting page, which will send the rest of Logout.jsp page to the user.
- If the request is for any other page, control is returned to the requesting page, which will continue sending the page to the user.

The method body is:

```
public final void doGet(HttpServletRequest req, HttpServletResponse res)
    throws javax.servlet.ServletException, java.io.IOException
{
    String destination = req.getPathInfo();
    String pageName = destination.substring(destination.lastIndexOf("/"));
    if (isLoggedIn(req)){
        if (pageName.equals("/login.jsp")){
            res.sendRedirect(
                "/already_loggedin.html");
        }
        else if (pageName.equals("/logout.jsp")){
            HttpSession session = req.getSession(false);
            if( session != null){
                session.invalidate();
            }
        }
    }
    else{
        if ( !pageName.equals("/login.jsp")){
            res.sendRedirect("/not_logged_in.html");
        }
    }
}
```

isLoggedIn

The isLoggedIn method checks to see if the user is logged into the Home Banking Application. If the user is logged in it returns true, otherwise it returns false:

```
public boolean isLoggedIn(HttpServletRequest req)
{
    boolean status = false;
    HttpSession session = req.getSession(false);
    if (session != null){
        status = true;
    }
    return status;
}
```

5.6 Login

The Login subsystem is the entry point into the Home Banking Application application (Figure 74). The design of this subsystem is explained in “Login” on page 39.

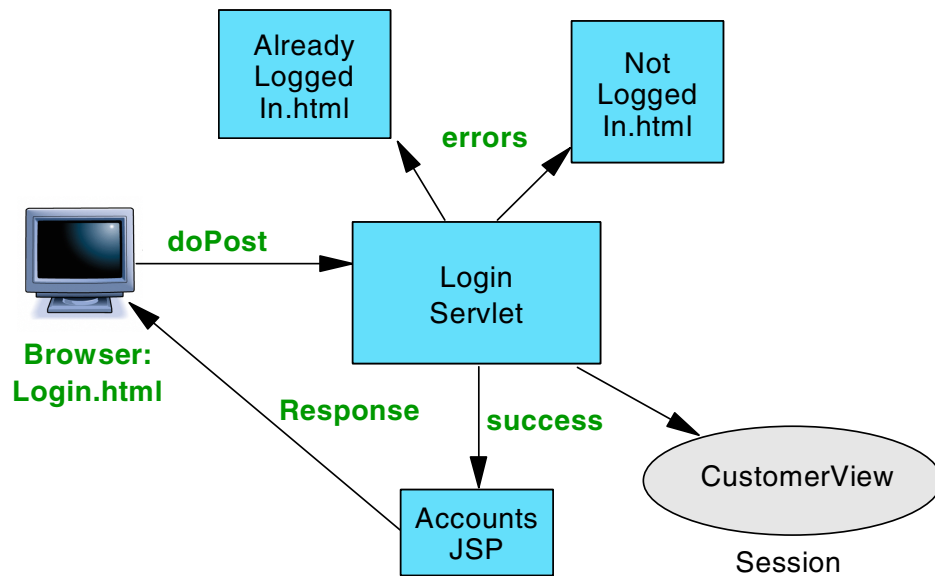


Figure 74. Login Subsystem

When the user goes to the Login page they need to enter their user ID and login password (Figure 75).

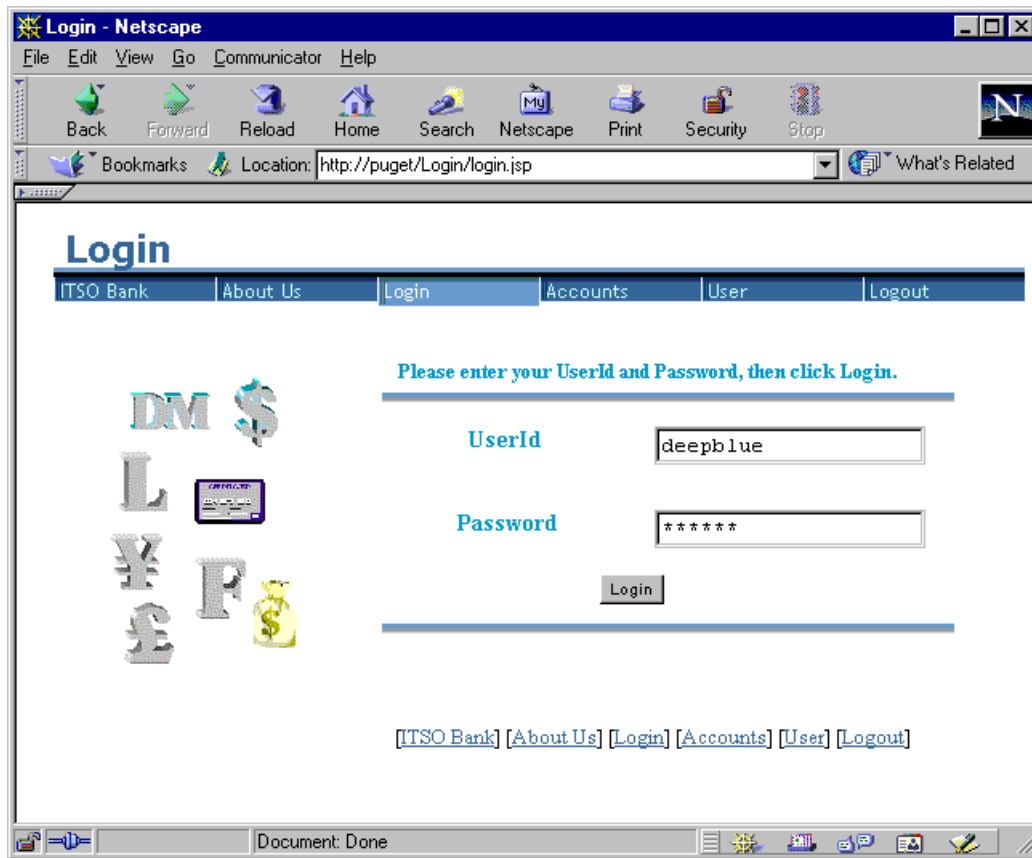


Figure 75. Login Screen

Once they enter these values, they click **Login** to submit their request. If the login is successful, they are sent to the Accounts page (Figure 76); otherwise they are sent to the Unsuccessful Login page (Figure 77).

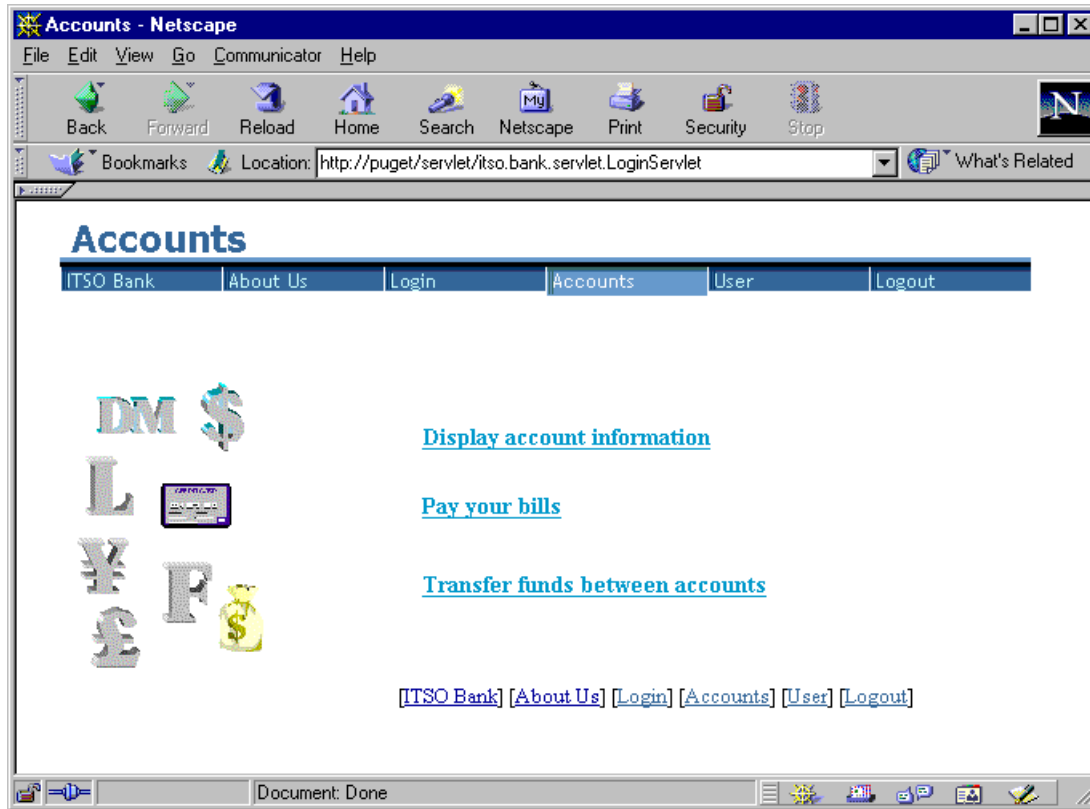


Figure 76. Accounts Page

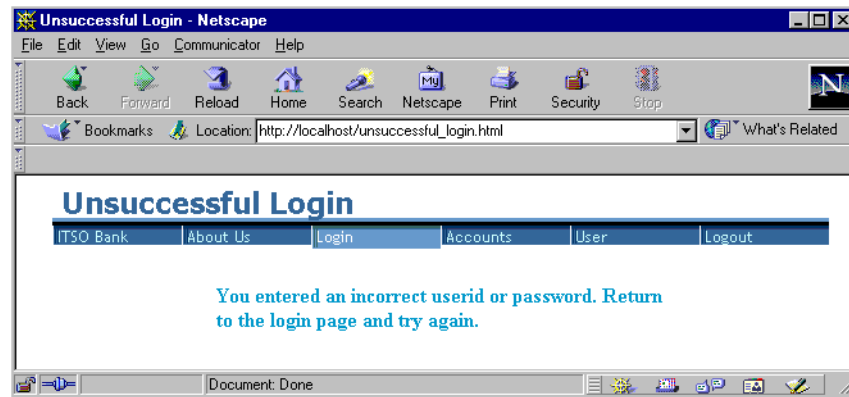


Figure 77. Unsuccessful Login Page

5.6.1 Login Interaction

When the user submits the login form, the request is sent to the doPost method of the LoginServlet (Figure 78). The doPost method retrieves the user's user ID and password and delegates the authentication to the login method. This method retrieves the Customer object for this userid from the bank, if one exists. The Customer object checks the password of the user. If the password is correct a session is created, a new CustomerView object is added to the session and the user is sent to the Accounts JSP. If an error occurs, the user is sent to the Unsuccessful Login page.

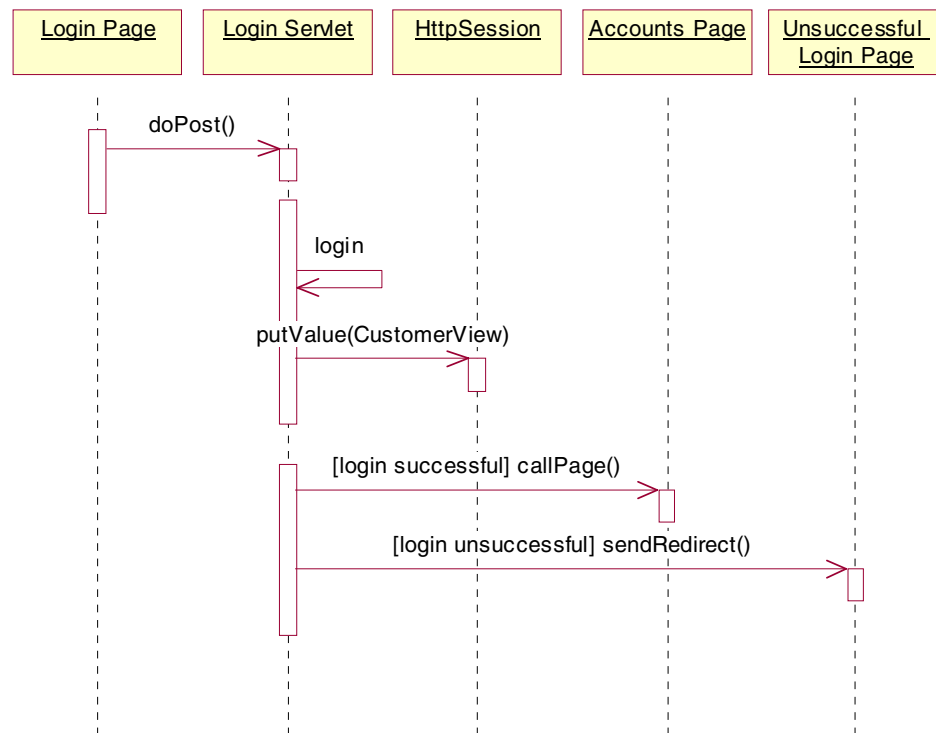


Figure 78. LoginServlet Interaction Diagram

5.6.2 Login Servlets

The LoginServlet is the gatekeeper of the HBA application. All users are granted entry to the application by this servlet.

LoginServlet

The LoginServlet is responsible for authenticating the user's login attempt. If the login attempt is successful, it allows entry into the Home Banking Application. It is also responsible for taking any actions if the login attempt was unsuccessful.

Table 4 shows the LoginServlet methods and Table 5 shows the collaborating objects.

Table 4. LoginServlet Methods

Method	Description
doPost	Process the Login request. Called when a user submits the form on the Login page ("doPost" on page 120).
init	Access the XML configuration file.
login	Performs the login validation. Sends the user to the appropriate page based on the login status: Accounts if successful, Unsuccessful Login if unsuccessful.

Table 5. LoginServlet Collaborators

Class	Description
BankHome	Provide a reference to the bank.
Bank	Provide a reference to a customer object.
Customer	Check the login password.
CustomerView	Store customer information in session.
HttpSession	Add CustomerView to session.
HttpRequest	Provide UserId and Password.
HttpResponse	Invoke sendRedirect.

LoginServlet Error Handling

Send the user to Unsuccessful Login page if the login is unsuccessful. If any other error is found, the user is sent to the ITSO Bank Error page with a description of the error using the callErrorPage method.

LoginServlet Methods

doPost

As shown in the interaction diagram (Figure 78 on page 118), the method called when the user submits the Login Form is `doPost`:

```
public final void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
{
    try {
        String userId = req.getParameter("txtUserId");
        String password = req.getParameter("txtPassword");
        login(req, res, userId, password);
    }
    catch (Exception e)
    {
        callErrorPage( req, res, e);
        return;
    }
}
```

init

```
public final void init(ServletConfig config) throws
    javax.servlet.ServletException {
    super.init( config);
    xmlconfig = new XMLConfigUtil( this);
}
```

login

The `login` method is used to validate the login attempt of the user. If the login attempt is successful, the user is sent to the Accounts page, otherwise they are sent to the Unsuccessful Login Page. The Bank provides the `getCustomerById` method, which is used to retrieve a Customer object from the bank. The Customer object is used throughout the HBA to get and put information about the customer to and from the bank.

The Customer object is retrieved from the Bank by the call to `getCustomerById`.

```
public final void login(HttpServletRequest req, HttpServletResponse res,
    String userId, String password)
    throws ServletException, IOException,
    ITSOBankCommunicationException, ITSOBankException
{
    Customer customer = null;
    boolean status = false;
    if (BankHome.getBank() != null) {
        customer = BankHome.getBank().getCustomerById(userId);
    }
}
```

The login password that was entered is validated against that of the Customer object. If the login attempt by the user was successful, a status flag is set to true. If there is an error of another type, it is redirected to the callErrorPage method. If the customer is authenticated (status is true), then a new session is created.

```
if (customer != null) {
    if (customer.checkLoginPassword(password)) {
        HttpSession session = req.getSession(true);
        if (session != null) {
            status = true;
        }
    }
}
```

If the customer is authenticated (status is true) then a CustomerView object is created and stored in the user's session. The purpose of storing the object in the session is to make it accessible to the other subsystems.

```
        status = true;
        session.putValue("customer",
            new itso.bank.viewobjects.CustomerView( customer));
    }
    else{
        callErrorPage( req, res, new Exception
            ("Error creating session"));
        return;
    }
}
```

If the customer is authenticated (status is true) then the user is sent to the Accounts page, otherwise they are redirected to the Unsuccessful Login page.

```
if (status){
    callPage("accounts", req, res);
}
else{
    res.sendRedirect(xmlconfig.getPageURI("unsuccessful_login"));
}
else{
    callErrorPage( req, res, new Exception("Bank servlet null"));
    return;
}
}
```

5.6.3 Login JavaServer Pages and HTML Pages

The Login subsystem uses several JSP and HTML pages:

- Login—Provides the Login form. If the user is already logged in when they access this page, the BankServlet sends them to the Already Logged In page.
- Accounts—Lists the functions that users can perform on their accounts.
- Unsuccessful Login—This HTML page simply tells the user they entered an incorrect userid or password.
- Already Logged In—This HTML page tells the user that they are already logged in.
- ITSO Bank Error—This page is used throughout the HBA if an unexpected error is encountered. It displays a message using the following code:

```
<p>The HBA application has encountered the following error:  
<BEAN NAME="error" TYPE="java.lang.Exception" INTROSPECT="no"  
CREATE="no" SCOPE="request"> </BEAN>  
<br>  
<INSERT BEAN="exception"> </INSERT>
```

5.7 Account Information

Users access the Account Information subsystem to request a current balance for a specified account, or a transaction history for that account. When the user clicks on Account Information from the Accounts page they are presented with a screen that has a drop-down list of their available accounts and the actions they can perform on these accounts (Figure 79).

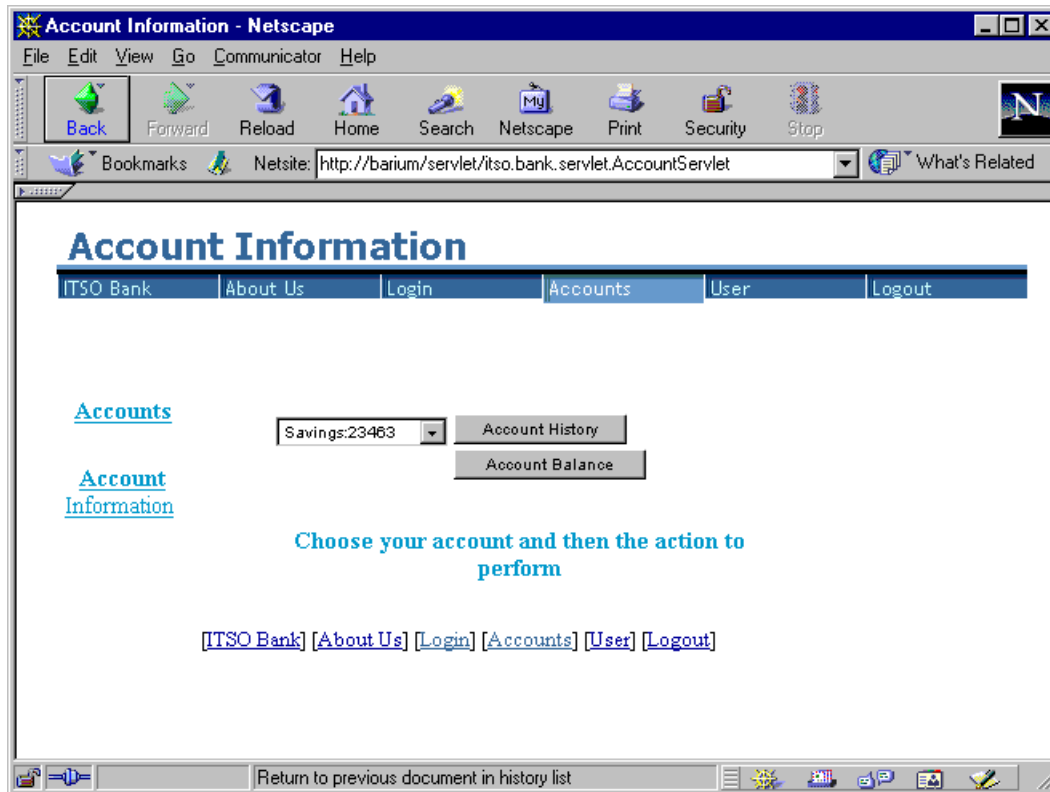


Figure 79. Account Information Page

The user selects the account from the drop down list, and an action to perform. If they choose the Account Balance button, the current balance for that account will be displayed (Figure 80). If they choose the Account History button, they get a transaction history for that account (Figure 81).

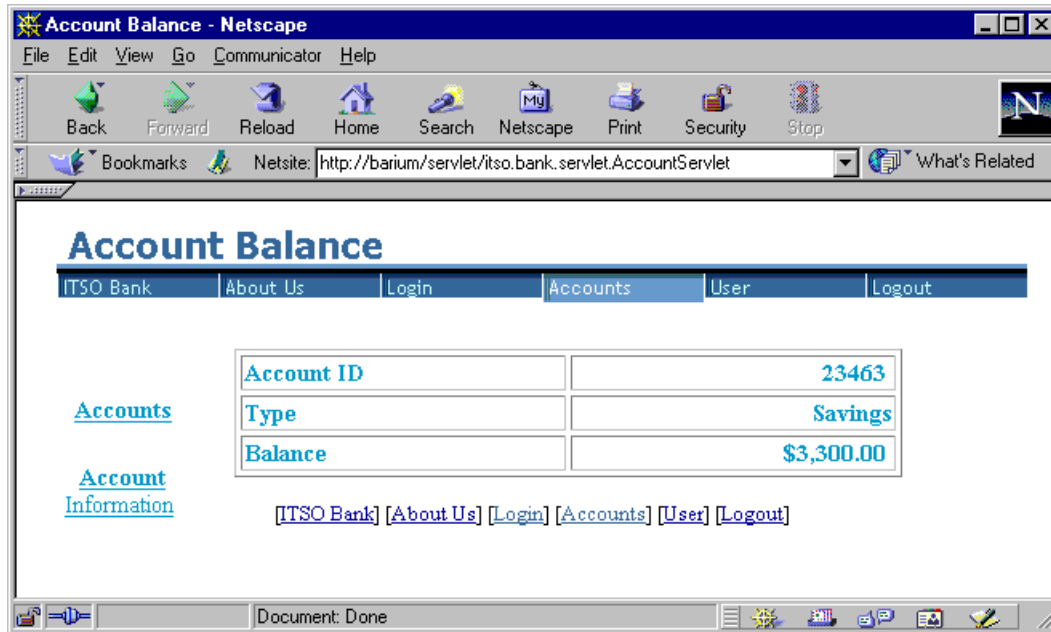


Figure 80. Account Balance Page

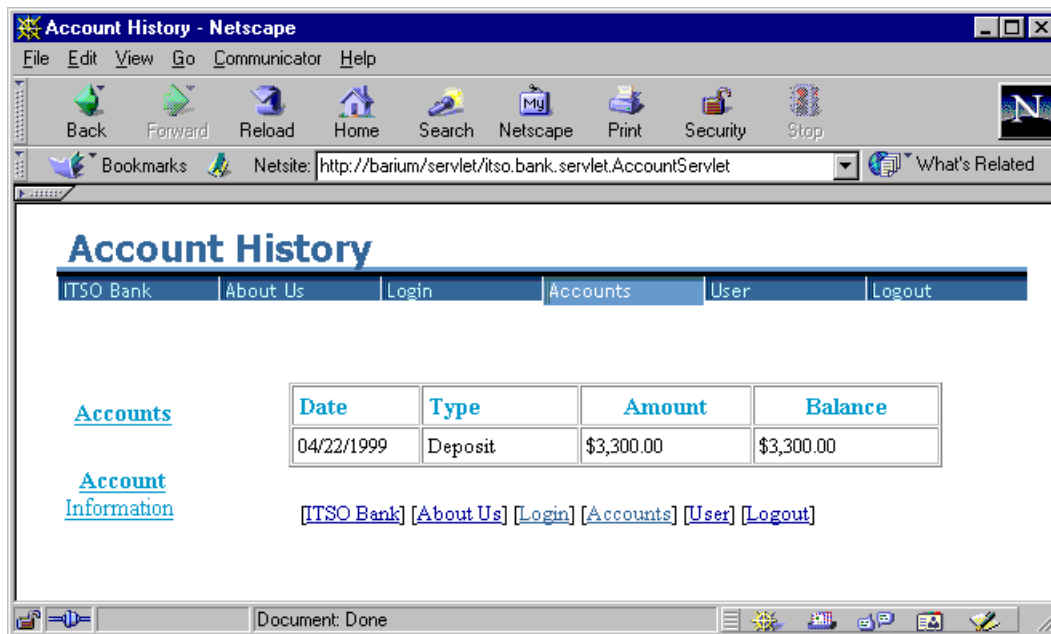


Figure 81. Account History Page

5.7.1 Account Information Interaction

The AccountServlet is invoked by the user selecting **Account Information** on the Accounts page (invoking doGet) or pressing the **Account History** or **Account Balance** button in the Account Information page to invoke the doPost method (Figure 82).

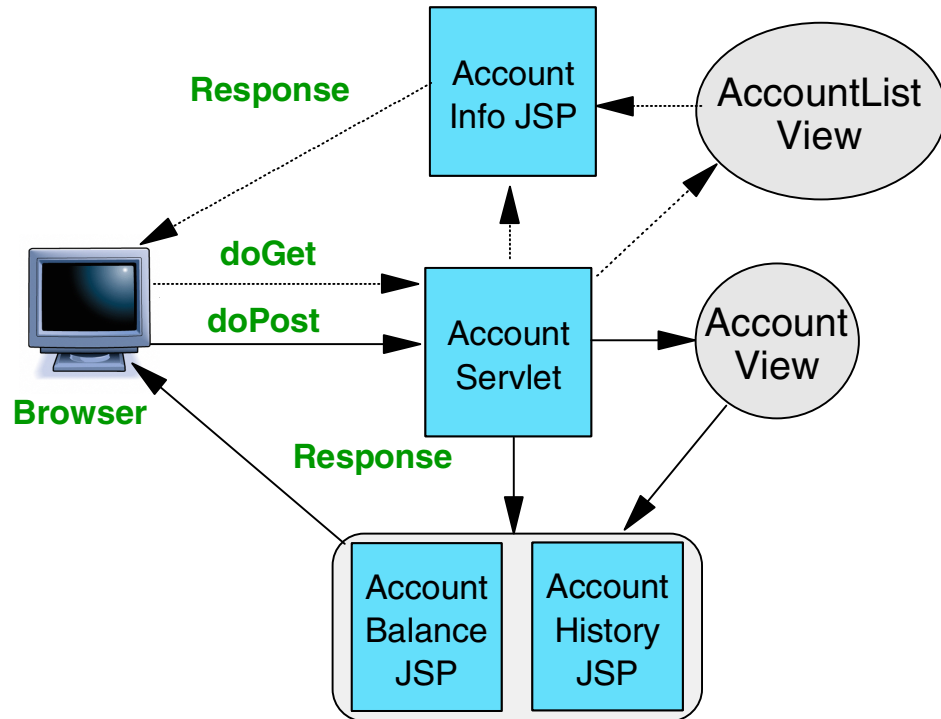


Figure 82. Account Information Architecture

The AccountServlet determines which account inquiry the user requested and calls the appropriate JavaServer Page. If account information was selected (doGet was invoked), it calls the Account Information JSP. If an account balance was requested (doPost was invoked), it calls the Account Balance JSP (“AccountBalance” on page 130); otherwise it calls the Account History JSP (“Account History” on page 130). The interactions involved are shown in Figure 83 and Figure 84.

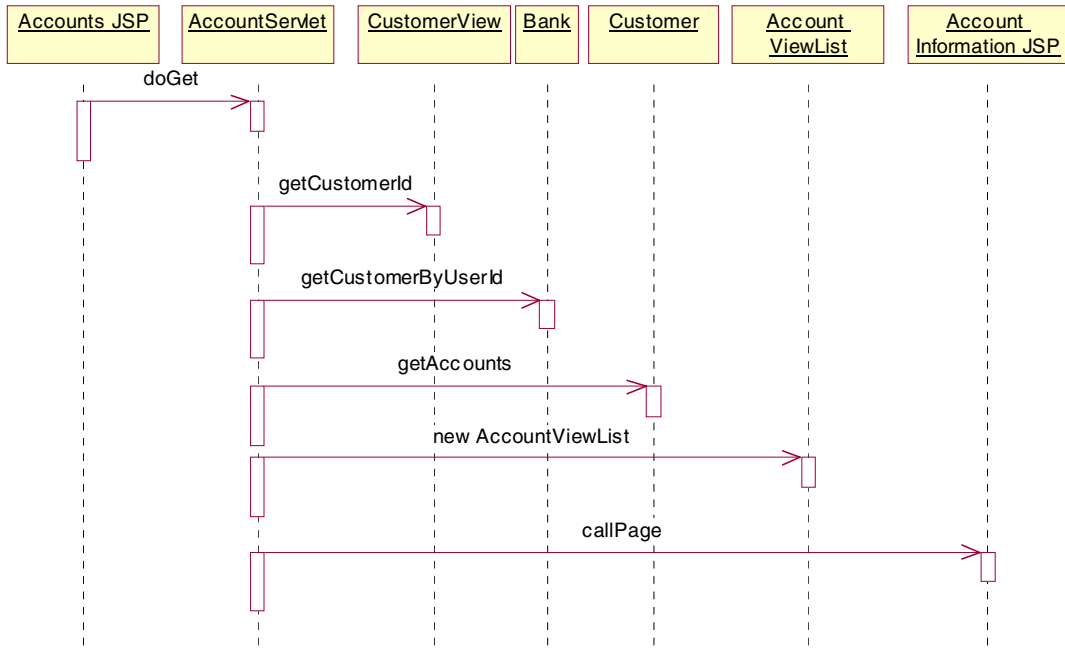


Figure 83. Account Information Interaction

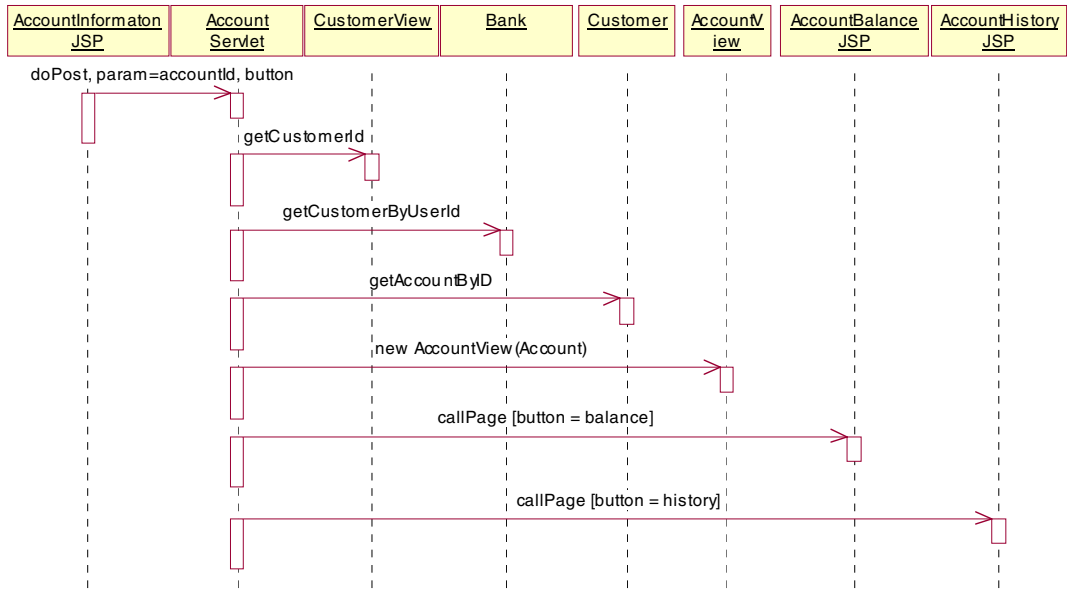


Figure 84. Account Balance and History Interaction

5.7.2 Account Information Servlets

Account Servlet

There is only one servlet required in this subsystem, the AccountServlet. The AccountServlet is responsible for coordinating user requests for account information and delegating that request to the appropriate JavaServer Page.

Table 6 shows the AccountServlet methods and Table 7 shows the collaborating objects.

Table 6. AccountServlet Methods

Method	Description
doGet	Creates the Account Information page.
doPost	Calls either the Account History JSP or the Account Balance JSP with the account information.

Table 7. AccountServlet Collaborators

Class	Description
BankHome	Provide a reference to the bank.
Bank	Used to get a reference to a customer object.
Customer	Provides a list of accounts or a single account object.
CustomerView	Used to store customer information in the session.
BankAccount	Provides account information.
BankAccountView	Used to store account information in request object.
BankAccountViewList	Used to store information for a set of accounts in the request object.
HttpSession	Get CustomerView from session.
HttpRequest	Provides account IDs and button values.

AccountServlet Error Handling

Send the user to the ITSO Bank Error page with a description of the error.

AccountServlet Methods

doGet

The doGet method stores the user's accounts in the request and calls the Account Information page:

```
public final void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
{
    itso.bank.util.CacheControl.setCache( res, true);
    HttpSession session = req.getSession(false);
    if( session == null){
        callErrorPage( req, res, new Exception("No session"));
        return;
    }
    CustomerView customerView =
        (itso.bank.viewobjects.CustomerView)
        session.getValue("customer");
    try {
        Customer customer = BankHome.getBank().getCustomerById(
            customerView.getUserId());
```

Get the customer's accounts and store them in a view object: BankAccountViewList. The BankAccountViewList object is placed into the request object for subsequent retrieval by the Account Information JavaServer Page. It uses the setAttribute method of the HttpServletRequest class, which is a subclass of HttpServletRequest, necessitating the cast.

```
        BankAccountViewList accountList =
            new BankAccountViewList( customer.getAccounts());
        ((com.sun.server.http.HttpServletRequest)req).
            setAttribute( "accountlist", accountList);
    }
    catch (Exception e) {
        callErrorPage( req, res, e);
        return;
    }
    callPage( AccountInfoJSP, req, res);
}
```

Page caching (introduced in 2.7, "Caching" on page 21) is controlled by a call to itso.bank.util.CacheControl.setCache(res, true), which instructs the browser not to cache the page. The setCache methods are shown in 5.12.1, "CacheControl" on page 166.

doPost

This method is invoked from a POST request from the Account Information JSP. The selected account view object is placed into the request object.

```

public final void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    String accountID = req.getParameter("lstAccountName");
    itso.bank.util.CacheControl.setCache( res, true);
    HttpSession session = req.getSession(false);
    if( session == null){
        callErrorPage( req, res,
            new Exception("Null session in Account servlet"));
        return;
    }
    CustomerView customerView = (itso.bank.viewobjects.CustomerView)
        session.getValue("customer");
    try {
        Customer customer = BankHome.getBank().
            getCustomerByUserId(customerView.getUserId());
        BankAccount account = customer.getAccountByID(accountID);
        ((com.sun.server.http.HttpServiceRequest)req).
            setAttribute( "account", new BankAccountView(account));
    }
    catch (Exception e) {
        callErrorPage( req, res, e);
        return;
    }
    if (req.getParameter("btnSubmit").equals("Account Balance"))
        callPage( AccountBalanceJSP, req, res);
    else
        callPage( AccountHistoryJSP, req, res);
}

```

5.7.3 Account Information JavaServer Pages

There are three JavaServer Pages used in the Account Information subsystem:

- Account Information—Provides a list of accounts and buttons to choose account history or balance inquiries.
- Account Balance—Retrieves the current balance of the selected account.
- Account History—Retrieves a transaction history for the selected account.

Account Information

The Account Information JSP builds the user interface that allows a user to make account inquiries (Figure 79 on page 123). The dynamic portion of the page (the ComboBox or pulldown list of accounts) is created using the following code:

```

<BEAN NAME="accountlist" TYPE="itso.bank.viewobjects.BankAccountViewList"
INTROSPECT="no" CREATE="no" SCOPE="request"> </BEAN>
<select id="FormsComboBox1" name="lstAccountName" >
<repeat index=count>
<% accountlist.getAccounts(count); %>
<option value=<insert bean = accountlist
    property=accounts(count).accountId></insert>>
<insert bean = accountlist property=accounts(count).accountType>
</insert>;
<insert bean = accountlist property=accounts(count).accountId></insert>
</option>
</repeat>
</select>

```

The REPEAT tag loops until an IndexOutOfBoundsException is thrown. The line `<% accountlist.getAccounts(count); %>` is used to stop the loop before the next option tag is started. The processing of the user request is sent to the AccountServlet:

```

<FORM NAME="LayoutRegion2FORM"
ACTION="/servlet/itso.bank.servlet.AccountServlet" METHOD=POST>

```

AccountBalance

This JSP (Figure 80 on page 124) is responsible for displaying the current account balance by extracting the AccountView object from the request and inserting the fields into a table. The table code (without formatting tags) is:

```

<BEAN NAME="account" TYPE="itso.bank.viewobjects.BankAccountView"
INTROSPECT="no" CREATE="no" SCOPE="request"></BEAN>
<table>
<tr>
<td>Account ID</td>
<TD><INSERT BEAN="account" PROPERTY="accountId"></INSERT></TD>
</tr><tr>
<td>Type</td>
<TD><INSERT BEAN="account" PROPERTY="accountType"></INSERT></TD>
</tr><tr>
<td>Balance</td>
<TD><INSERT BEAN="account" PROPERTY="balance"></INSERT>
</TD></tr></table>

```

Account History

The Account History JSP (Figure 81 on page 124) produces a statement of all transactions that have taken place in an account. Like the Account Balance JSP, it extracts the AccountView object from the request. In this case, we are interested in the accounts' transactions. The code is as follows:

```

BEAN NAME="account" TYPE="itso.bank.viewobjects.BankAccountView"
INTROSPECT="no" CREATE="no" SCOPE="request"> </BEAN>
<table><TR>
<TD>Date</TD><TD>Type</TD><TD>Amount</TD><TD>Balance</TD>
<repeat index=count>
<% account.getTransactions( count); %>
<tr><td>
<insert bean=account property=transactions(count).transTimeStamp>
</insert></td>
<td><insert bean=account property=transactions(count).transType>
</insert></td>
<td><insert bean=account property=transactions(count).transAmount>
</insert></td>
<td><insert bean=account property=transactions(count).transClosingBalance>
</insert></td>
</tr></repeat></table>

```

5.8 Bill Payment

When a user wants to pay a bill, they go to the Pay Bill JSP page of the HBA (Figure 85). The user selects the account to pay the bill from, the payee, an amount, and a transaction password and clicks **Pay Bill**.

Figure 85. Pay Bill Page

The system responds with the Bill Paid JSP (Figure 86), which shows that the request has been processed, or redisplay the Pay Bill JSP with an error message.

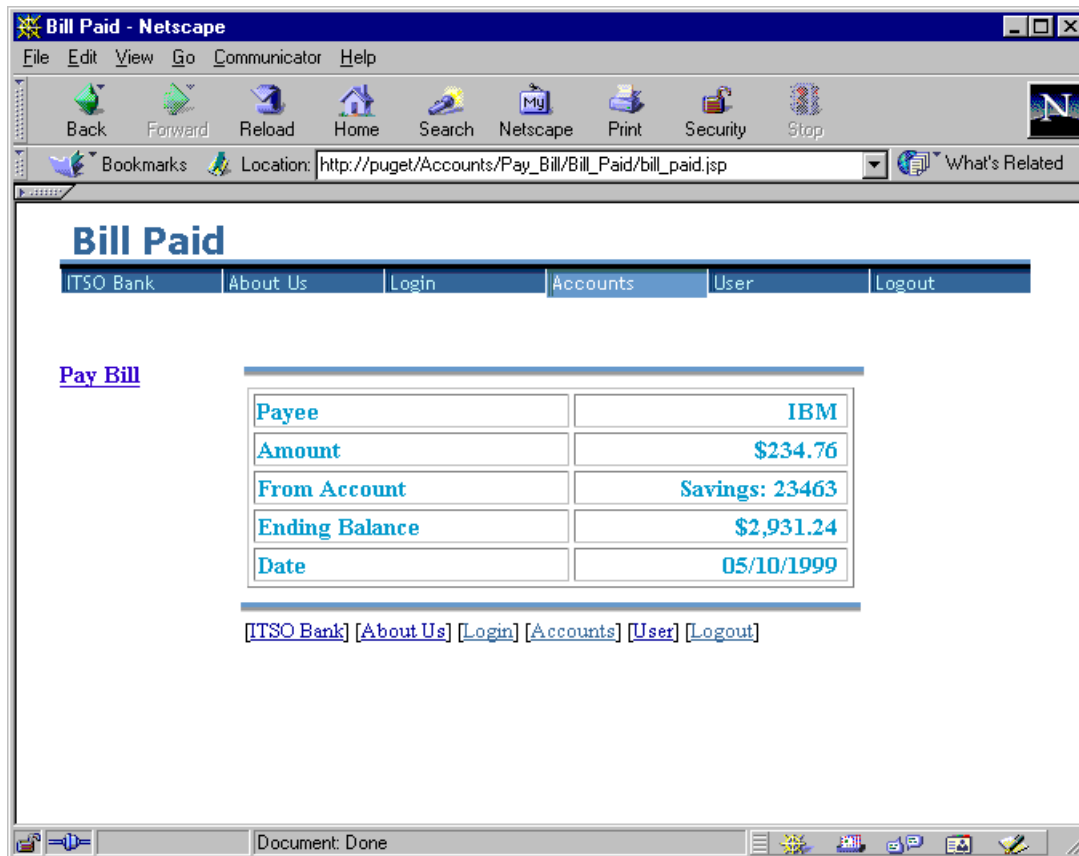


Figure 86. Bill Paid Page

5.8.1 Bill Payment Interaction

The Bill Payment JavaServer Page is displayed through a GET request to the MoneyTransferServlet with a parameter of PayBill (Figure 87). The actual transaction is performed by the BillPaymentServlet (Figure 88).

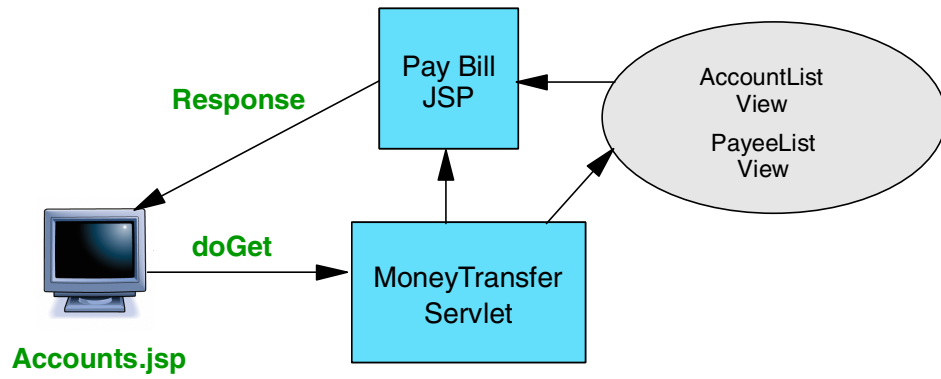


Figure 87. Bill Payment Architecture: Choose Bill Payment

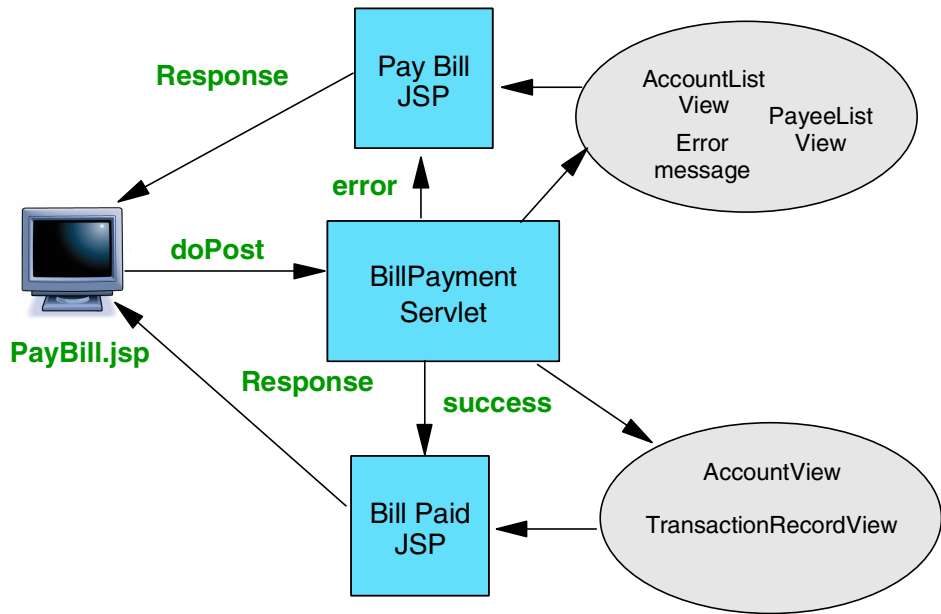


Figure 88. Bill Payment Architecture: Pay Bill

As shown in Figure 89, the doGet request from the Accounts JavaServer Page invokes the MoneyTransferServlet, which displays the appropriate page.

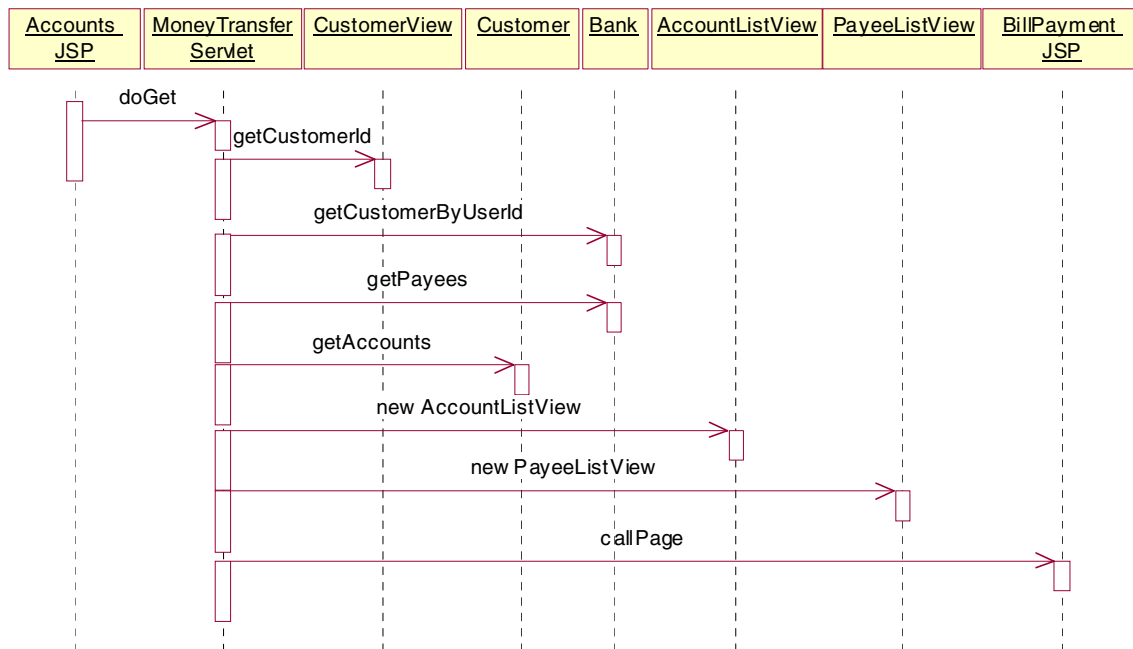


Figure 89. Displaying the Pay Bill or Transfer Funds JavaServer Page

When you enter your data and submit the form to pay a bill, a POST request is sent to the BillPaymentServlet. The doPost method in turn calls the MoneyTransferServlet's processRequest method, which validates the data that was submitted in the Pay Bill Form. If the validation fails, the user is sent back to the Pay Bill page with an error message. If the validation succeeds, the MoneyTransferServlet's transferFunds method is called. Once the transfer has occurred the user is sent to the Bill Paid JSP page (Figure 86 on page 132).

Before the transferFunds method is called, the servlet must retrieve the source and destination accounts from the Customer object and pass these objects along with the amount to the transferFunds method. The transferFunds method invokes the transfer method of the source BankAccount passing the destination BankAccount and the amount to transfer as parameters. If the transfer fails, the transfer method throws a BankTransactionException and the complete transaction is aborted. Figure 90 shows the interaction.

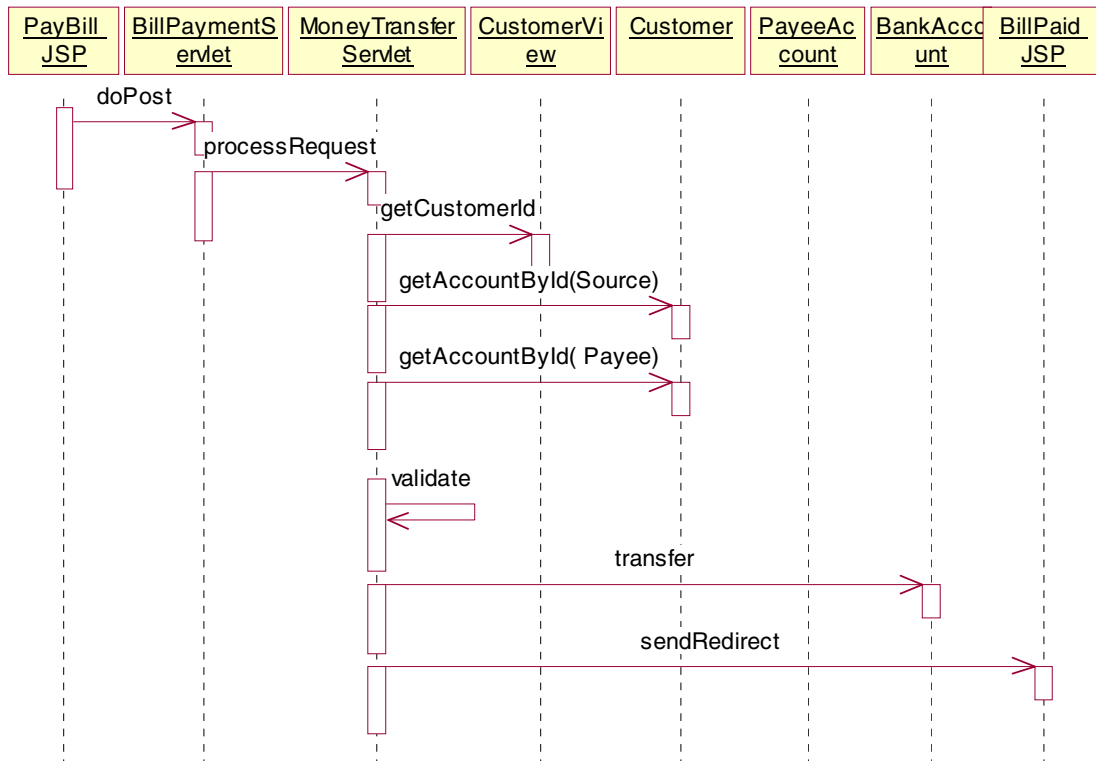


Figure 90. Bill Payment Interaction Diagram

5.8.2 Bill Payment Servlets

The BillPayment servlet transfers funds between a Checking or Savings Account and a Payee Account. The BillPaymentServlet extends MoneyTransfer servlet. The purpose of the MoneyTransfer servlet is to transfer money between two bank accounts.

MoneyTransferServlet

This servlet is responsible for displaying the initial Pay Bill or Transfer Funds page and validating that the source account has sufficient funds for the transfer and that the amount and password are valid. The servlet then performs the transaction if the validation is successful. Once the transaction is complete the user is sent to the appropriate page.

In this implementation of the HBA, the MoneyTransferServlet could have handled all the function of paying bills and transferring funds. However, we recognized that our implementation was very simplistic, and that providing the subclasses would provide more flexibility for another implementation.

Table 8 shows the MoneyTransferServlet methods and Table 9 shows the collaborating objects.

Table 8. MoneyTransferServlet Methods

Method	Description
doGet	Display the BillPayment JSP or TransferFunds JSP based on the passed parameter.
processRequest	Performs the validation. If the validation succeeds performs the transfer and sends the user to a result page, otherwise sends the user to the source page with an error message.
transferFunds	Performs the transfer between the two bank accounts.
validate	Validates the request to transfer the money. Checks for correct syntax, and any other requirements.

Table 9. MoneyTransferServlet Collaborators

Class	Description
Customer	Used to get the source and target bank accounts.
BankAccount	Used to transfer money between accounts.
BankHome	Provide a reference to the bank.
Bank	Used to get a reference to a customer object.
CustomerView	Used to store customer information in the session.
BankAccount	Provides account information.
BankAccountViewList	Used to store information for a set of accounts in the request object.
PayeeAccount	Provides account information.
PayeeAccountViewList	Used to store information for a set of accounts in the request object.
HttpSession	Add view objects to the request object.
HttpRequest	Provide request information.
HttpResponse	Invoke sendRedirect.

MoneyTransferServlet Error Handling

A `BankTransactionException` is thrown if an error occurs in the money transfer. If other errors are found, the `callErrorPage` method is called.

MoneyTransferServlet Methods

init

Access the XML servlet configuration.

```
public final void init(ServletConfig config) throws
    javax.servlet.ServletException {
    super.init( config);
    xmlconfig = new XMLConfigUtil( this);
}
```

doGet

Display the Pay Bill or Transfer Funds page.

```
public final void doGet(javax.servlet.http.HttpServletRequest req,
    javax.servlet.http.HttpServletResponse res) throws
    javax.servlet.ServletException, java.io.IOException
{
    BankAccountViewList fromAccounts = null;
    BankCollection toList = null;
    itso.bank.util.CacheControl.setCache( res, true);
```

Get the parameter: either `PayBill` or `TransferFunds`.

```
String dest = req.getQueryString();
HttpSession session = req.getSession(false);
if( session == null){
    callErrorPage( req, res,
        new Exception("Null session in Account servlet"));
    return;
}
CustomerView customerView =
    (itso.bank.viewobjects.CustomerView)
    session.getValue("customer");

try {
    Customer customer = BankHome.getBank().getCustomerByUserId(
        customerView.getUserId());
```

Build the list of accounts to transfer the money from.

```
fromAccounts = new BankAccountViewList( customer.getAccounts());
if( dest.equals("PayBill")){
    destination = "pay_bill";
```

If this is a bill payment, create a list of Payee accounts.

```
PayeeAccountViewList toAccounts =
    new PayeeAccountViewList(customer.getPayees());
((com.sun.server.http.HttpServiceRequest)req).
    setAttribute("toaccounts", toAccounts);
}
else{
```

If the same list is used for source and destination (it is a transfer, not a bill payment), just refer to the first list.

```
destination = "transfer_funds";
BankAccountViewList toAccounts = fromAccounts;
((com.sun.server.http.HttpServiceRequest)req).
    setAttribute("toaccounts", toAccounts);
}
```

Use a view object to hold each account list.

```
((com.sun.server.http.HttpServiceRequest)req).
    setAttribute("fromaccounts", fromAccounts);
((com.sun.server.http.HttpServiceRequest)req).
    setAttribute("message", message);
}
catch (Exception e) {
    callErrorPage(req, res, e);
    return;
}
callPage(destination, req, res);
}
```

processRequest

The processRequest method is invoked during a bill payment or a funds transfer to transfer the funds between accounts.

```
public final void processRequest(HttpServletRequest req,
    HttpServletResponse res, String srcAccount, String dstAccount,
    String amount, String passCode) throws java.io.IOException
{
    BankAccount sourceAccount = null;
    BankAccount destinationAccount = null;
    Customer customer = null;
    String message = "";
    HttpSession session = req.getSession(false);
    if( session == null){
        callErrorPage( req, res, new Exception("Null session in servlet"));
        return;
    }
}
```

```

try
{
    CustomerView customerView = (itso.bank.viewobjects.CustomerView)
        session.getValue("customer");
    customer = BankHome.getBank().getCustomerByUserId(
        customerView.getUserId());

```

Get the source and destination bank accounts from the customer.

```

    sourceAccount = customer.getAccountById(srcAccount);
    destinationAccount = customer.getAccountById(dstAccount);

```

Validate the user input.

```

    if ((message = validate( customer, amount, passCode)) == null)
    {

```

If validated transfer the funds.

```

        TransactionRecord rec = transferFunds( sourceAccount,
            destinationAccount, amount);

```

Put the following objects in the session to be used by the Bill Paid and Funds Transferred pages. We need to use the session here because you cannot access the objects in the request object when you use sendRedirect. We do not use callPage here because the transaction would be replayed if the page was reloaded.

```

        session.putValue( "transrec", new TransactionRecordView( rec));
        session.putValue( "srcaccount",
            new BankAccountView(sourceAccount));
        if( this instanceof TransferFundsServlet){
            session.putValue( "destaccount",
                new BankAccountView( destinationAccount));
        }
        else{
            session.putValue( "destAccount",
                new PayeeAccountView( (PayeeAccount)destinationAccount));
        }

```

Send the user to the Destination JSP page using redirect instead of callPage so the transaction is not replayed.

```

        res.sendRedirect( this.destination);
    }
    else
    {
        throw new BankTransactionException( message);
    }

```

```

    }
    catch (itso.bank.common.BankTransactionException e)
    {

```

If we find errors, the original Pay Bill or Transfer Funds page is displayed with an error message.

```

        ((com.sun.server.http.HttpServiceRequest)req).
            setAttribute( "message", e.getMessage());
    try{
        BankAccountViewList fromAccounts =
            new BankAccountViewList( customer.getAccounts());
        if( destinationAccount instanceof PayeeAccount){
            PayeeAccountViewList toAccounts =
                new PayeeAccountViewList(customer.getPayees());
            ((com.sun.server.http.HttpServiceRequest)req).
                setAttribute( "toaccounts", toAccounts);
        }
        else{
            BankAccountViewList toAccounts =
                new BankAccountViewList(customer.getAccounts());
            ((com.sun.server.http.HttpServiceRequest)req).
                setAttribute( "toaccounts", toAccounts);
        }
        ((com.sun.server.http.HttpServiceRequest)req).
            setAttribute( "fromaccounts", fromAccounts);
        callPage(this.source, req, res);
    }
    catch( Exception f){
        callErrorPage( req, res, e);
        return;
    }
}
catch( Exception e){
    callErrorPage( req, res, e);
    return;
}
}

```

transferFunds

The transferFunds method performs the transfer on the business objects.

```

public final TransactionRecord transferFunds( BankAccount srcAccount,
    BankAccount dstAccount, String amount)
    throws ITSOBankCommunicationException,
    BankTransactionException, ITSOBankException
{
    if (srcAccount.getAccountId() == dstAccount.getAccountId())

```

```

        throw new itso.bank.common.BankTransactionException(
            "Accounts cannot be the same");
    return srcAccount.transfer(dstAccount,
        new java.math.BigDecimal(amount));
}

```

validate

The validate method checks to see that the values entered for the transfer are correct.

```

public final String validate( Customer customer,String amount,
    String passCode) throws ITSOBankCommunicationException,
    ITSOBankException
{
    String message = null;
    try {
        java.math.BigDecimal objAmount = new java.math.BigDecimal(amount);
        if (objAmount.compareTo( new java.math.BigDecimal(0)) < 0){
            message = "Cannot specify a negative amount";
        }
        else if (amount.equals("")) {
            message = "Invalid format for currency";
        }
        else if (passCode.equals("")) {
            message = "Please specify password";
        }
        else if (!customer.checkTransactionPassword(passCode)) {
            message = "Password incorrect: Authorization Denied!";
        }
    }
    catch (NumberFormatException e) {
        message = "Invalid format for currency";
    }
    return message;
}

```

BillPaymentServlet

The BillPayment servlet's task is to perform the bill payment transaction. When the customer submits the form to pay a bill to a payee (Figure 85) the BillPayment servlet fetches the data submitted in the form. It then transfers money from a customer bank account to the payee account using the MoneyTransferServlet's processRequest method.

Table 10 shows the BillPaymentServlet methods and Table 11 shows the collaborating objects.

Table 10. BillPaymentServlet Methods

Method	Description
doPost	Fetches the user data and delegates the transfer to the processRequest method of the MoneyTransfer servlet.

Table 11. BillPaymentServlet Collaborators

Class	Description
MoneyTransferServlet	The parent class of BillPayment servlet Provides this servlet with basic money transfer capability.
HttpRequest	Provide request information.
HttpResponse	Passed to processRequest.

BillPaymentServlet Error Handling

The user is sent to the Pay Bill JSP if there is an error in the parameters. For other errors the callErrorPage method is invoked.

BillPaymentServlet Methods

constructor

```
public BillPaymentServlet() {
    source = "pay_bill";
    destination = "bill_paid";
}
```

doPost

Retrieve the request parameters and invoke processRequest.

```
public final void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
{
    itso.bank.util.CacheControl.setCache( res, true);
    String accountId = req.getParameter("lstAccountName");
    String payee = req.getParameter("lstPayeeName");
    String amount = req.getParameter("txtAmount");
    String passCode = req.getParameter("txtPasscode");
    processRequest( req, res, accountId, payee, amount, passCode);
}
```


5.8.3 Bill Payment JavaServer Pages

The two JavaServer Pages involved in the BillPayment subsystem are the Pay Bill JSP (Figure 85 on page 131) used to enter the Bill Payment Information and the Bill Paid JSP (Figure 86 on page 132) to display the results.

Pay Bill

This JSP is responsible for displaying a screen from which to pay bills. It presents the user with lists of the user's bank accounts and payee accounts and fields to enter the amount and transaction password. After the user has filled in and submitted the form, the request is sent to the BillPayment servlet. This page has three dynamic components:

- Account ChoiceBox—This is populated using the same syntax as in 5.7, “Account Information” on page 122.
- Payee ChoiceBox—This is populated using the same syntax as in 5.7, “Account Information” on page 122, using the payee accounts of the customer.
- Error message—The error message is displayed on the BillPayment page if there was an error in a previous submission. The error message is displayed using the following code:

```
<BEAN NAME="message" TYPE="java.lang.String" INTROSPECT="no" CREATE="no"
SCOPE="request"> </BEAN>
<insert bean=message></insert>
```

The following HTML shows the ACTION attribute of the Pay Bill JSP Form.

```
<FORM NAME="LayoutRegion1FORM"
ACTION="/servlet/itso.bank.servlet.BillPaymentServlet" METHOD=POST>
```

Bill Paid

The user is sent to this page after a successful bill payment. The first thing this page does is get the information from the view beans in the request. It displays the payee name, source bank account name, the amount transferred, source account ending balance, and the transaction date. The code in the JSP (without the formatting attributes) is:

```
<BEAN NAME="srcaccount" TYPE="itso.bank.viewobjects.BankAccountView"
INTROSPECT="no" CREATE="no" SCOPE="session"> </BEAN>
<BEAN NAME="destaccount" TYPE="itso.bank.viewobjects.PayeeAccountView"
INTROSPECT="no" CREATE="no" SCOPE="session"> </BEAN>
<BEAN NAME="transrec" TYPE="itso.bank.viewobjects.TransactionRecordView"
INTROSPECT="no" CREATE="no" SCOPE="session"> </BEAN>
<table><tr><td>Payee</td>
<TD><insert bean=destaccount property=billPaymentTitle></insert>
```

```

</td></tr><tr><td>Amount</td>
<TD><insert bean=transrec property=transAmount></insert>
</td></tr><tr>
<td>From Account</td>
<TD><insert bean=srcaccount property=accountType></insert>:
<insert bean=srcaccount property=accountId></insert>
</td></tr><tr><td>Ending Balance</td><TD>
<insert bean=srcaccount property=balance></insert>
</td></tr><tr><td>Date</td><TD>
<insert bean=transrec property=transTimeStamp></insert>
</td></tr></table>

```

Note that the scope of the beans was session as discussed in “BillPaymentServlet Methods” on page 142.

5.9 Transfer Funds

Users access the Transfer Funds subsystem to transfer money between their bank accounts. A user selects the account to transfer money from, the account to transfer money to, an amount, and a transaction password and clicks the **Transfer** button (Figure 91).

The Transfer Funds subsystem is very similar to the Pay Bill subsystem, so it will be explained in less detail.

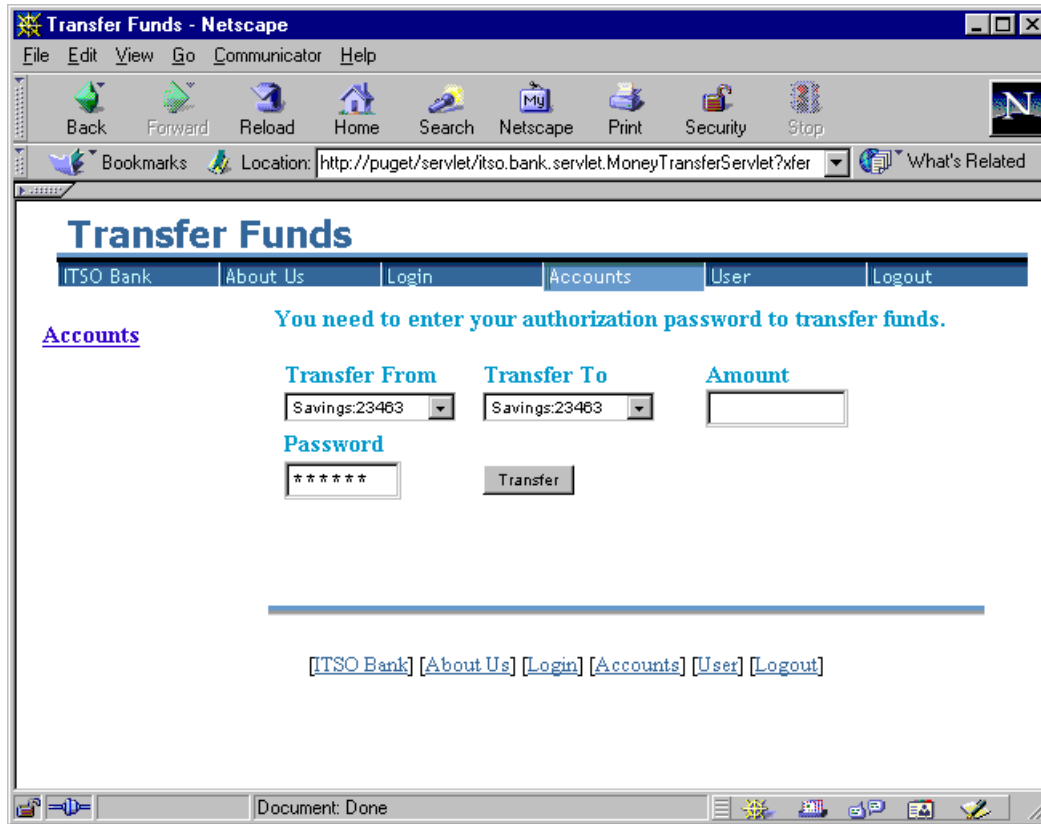


Figure 91. Transfer Funds Page

The user is then redirected to either the Funds Transferred page (Figure 92), which shows that the request has been processed, or back to the Transfer Funds page to display the error.

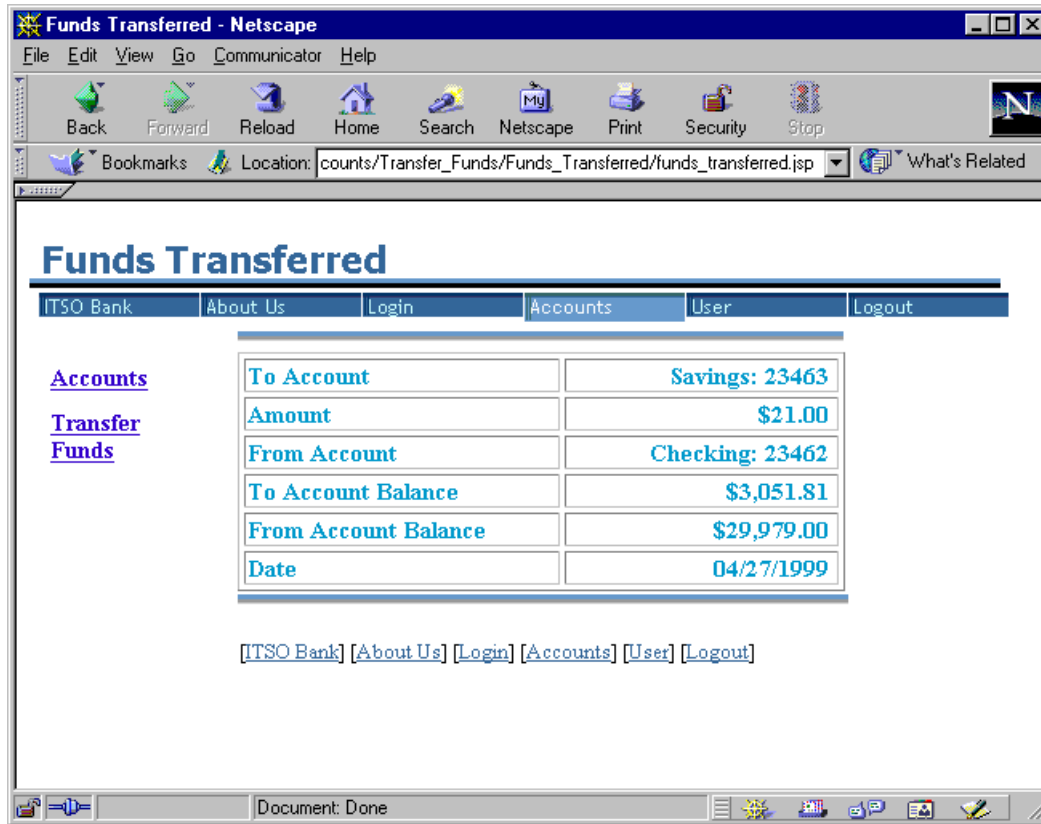


Figure 92. Funds Transferred Page

5.9.1 Funds Transfer Interaction

The Transfer Funds JavaServer Page is displayed through a GET request to the MoneyTransferServlet with a parameter of transfer (Figure 93). The actual transaction is performed by the TransferFundsServlet and MoneyTransferServlet (Figure 94).

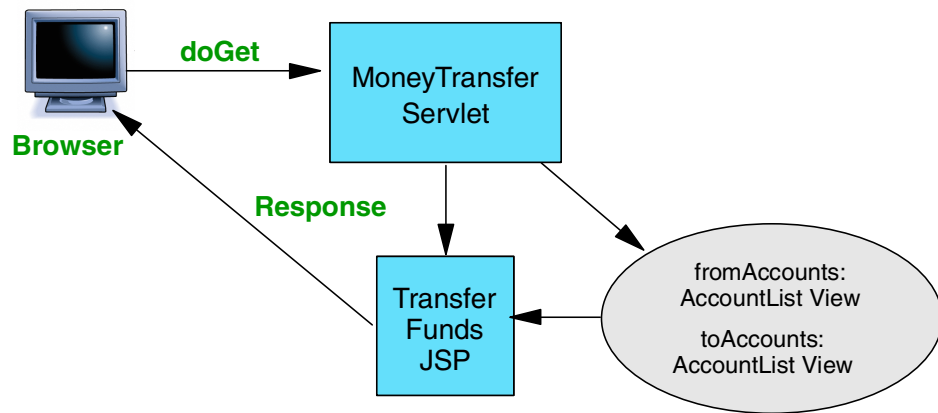


Figure 93. Transfer Funds Architecture: Choose Transfer Funds

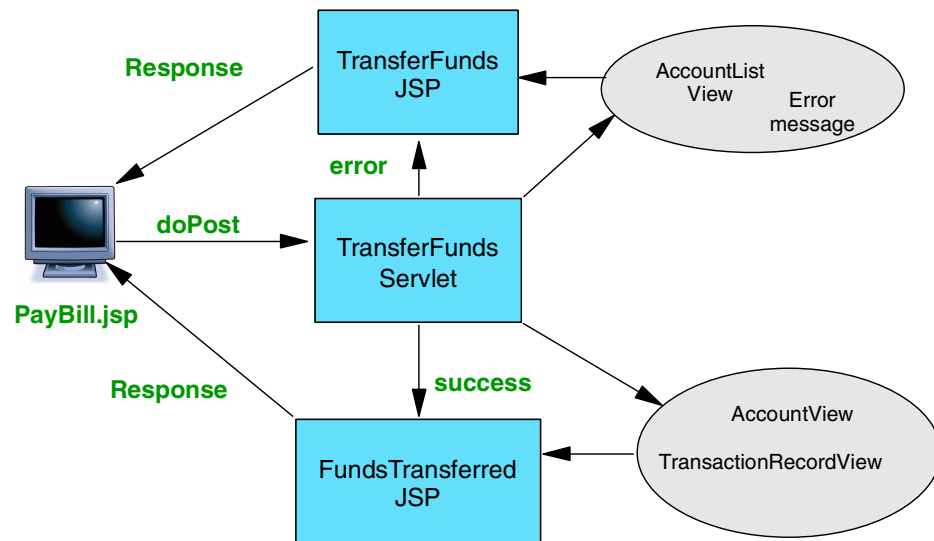


Figure 94. Transfer Funds Architecture: Transfer Funds

When you enter your data and submit the form for a funds transfer, the doPost method is invoked on the TransferFundsServlet. The doPost method calls the MoneyTransferServlet's processRequest method in the same way as the BillPaymentServlet. Once the money transfer has occurred, the callPage method is invoked, which sends the user to the Funds Transferred JSP page.

5.9.2 Transfer Funds Servlets

The Transfer Funds subsystem is made up of the MoneyTransferServlet and the TransferFundsServlet. The subsystem transfers funds between two of a user's checking or savings accounts. This functionality is common to Bill Payment as explained in the BillPayment subsystem. For this reason, the TransferFunds servlet also extends MoneyTransferServlet.

TransferFundsServlet

This servlet fetches the data that the user has submitted for transferring funds and calls processRequest to take care of the money transfer. This servlet extends MoneyTransferServlet.

Table 12 shows the TransferFundsServlet methods and Table 13 shows the collaborating objects.

Table 12. TransferFundsServlet Methods

Method	Description
doPost	This method performs the funds transfer. It fetches the data that the user has submitted for the funds transfer and delegates the transfer to the processRequest method of its parent class MoneyTransfer servlet.

Table 13. TransferFundsServlet Collaborators

Class	Description
MoneyTransferServlet	The parent class of TransferFunds servlet. Provides this servlet with basic money transfer capabilities.
HttpRequest	Provide request information.
HttpResponse	Passed to processRequest.

TransferFundsServlet Error Handling

The user is sent to the Pay Bill JSP if there is an error in the parameters. For other errors, the callErrorPage method is invoked.

Method Implementations

Constructor

```
public TransferFundsServlet ()
{
    source = "transfer_funds";
    destination = "funds_transferred";
}
```

doPost

```
public final void doPost(javax.servlet.http.HttpServletRequest req,
    javax.servlet.http.HttpServletResponse res) throws
    javax.servlet.ServletException, java.io.IOException
{
    String sourceAccountID = req.getParameter("srcAccountName");
    String targetAccountID = req.getParameter("destAccountName");
    String amount = req.getParameter("txtAmount");
    String passCode = req.getParameter("txtPasscode");
    itso.bank.util.CacheControl.setCache( res, true);
    processRequest(req, res, sourceAccountID, targetAccountID,
        amount, passCode);
}
```

5.9.3 Transfer Funds JavaServer Pages

The two JavaServer Pages involved in the Transfer Funds subsystem are the Transfer Funds JSP used to enter the Funds Transfer Information and the Funds Transferred JSP to display the results.

Transfer Funds JSP

This JSP is responsible for listing the users source and target bank accounts. It is almost exactly the same as the Pay Bill JSP in 5.8, "Bill Payment" on page 131.

Funds Transferred JSP

This JSP is responsible for showing the results of a successful transfer. It is almost exactly the same as the Bill Paid JSP in 5.8, "Bill Payment" on page 131.

5.10 Payee

The Payee subsystem is used to add and delete payees from the customer's payee list. The payees are added from the payee list of the bank.

The three pages of the Payee subsystem are shown in Figure 95, Figure 96, and Figure 97.

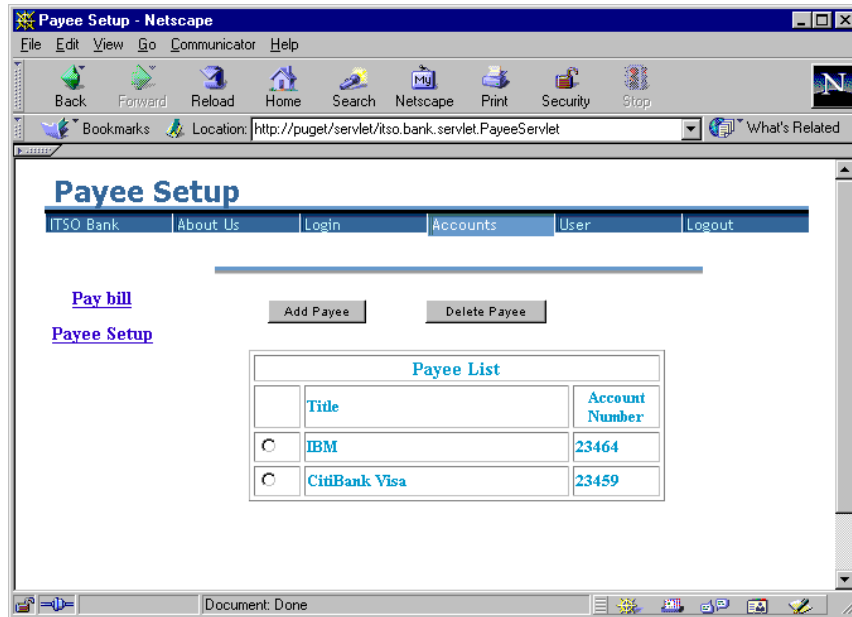


Figure 95. Payee Setup Page

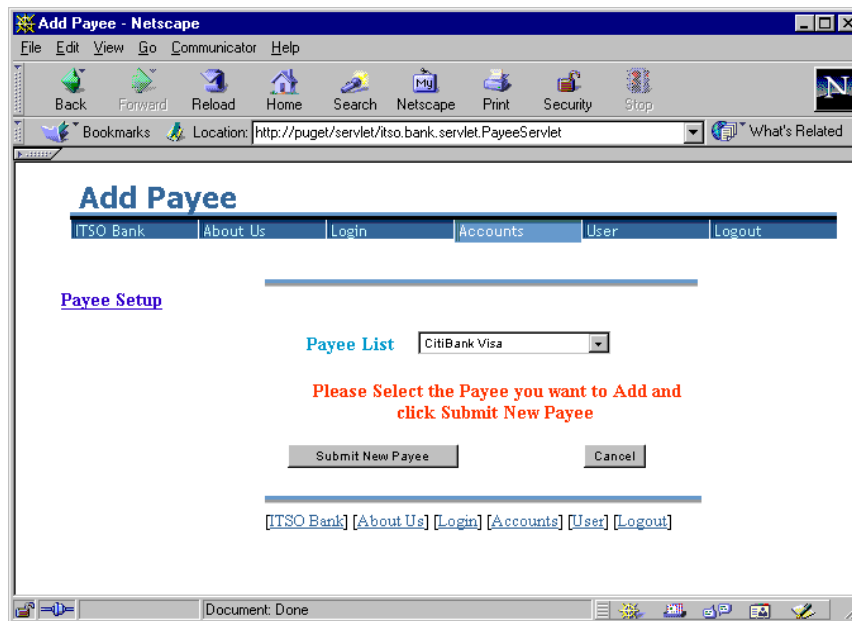


Figure 96. Add Payee Page

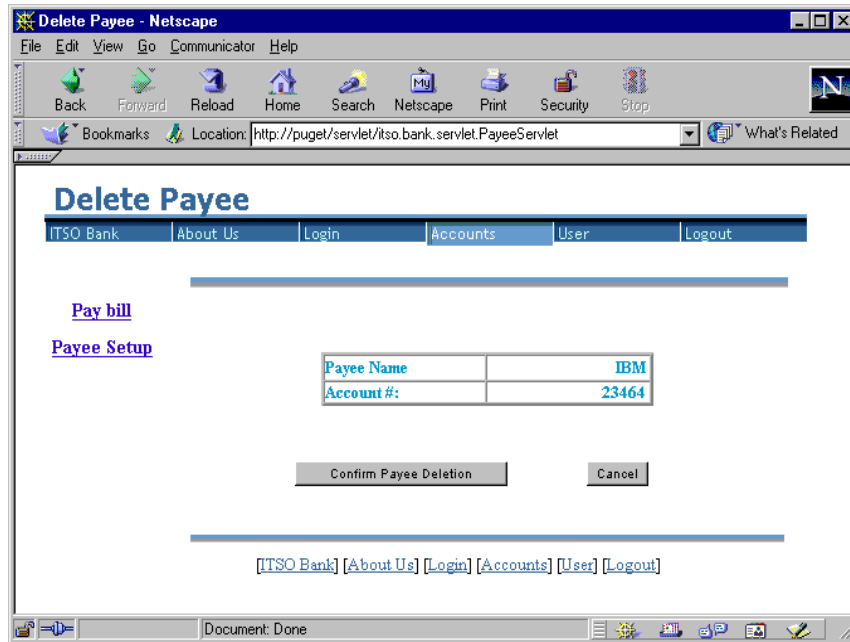


Figure 97. Delete Payee Page

5.10.1 Payee Interaction

The Payee subsystem is composed of one servlet and three JavaServer Pages, as shown in Figure 98.

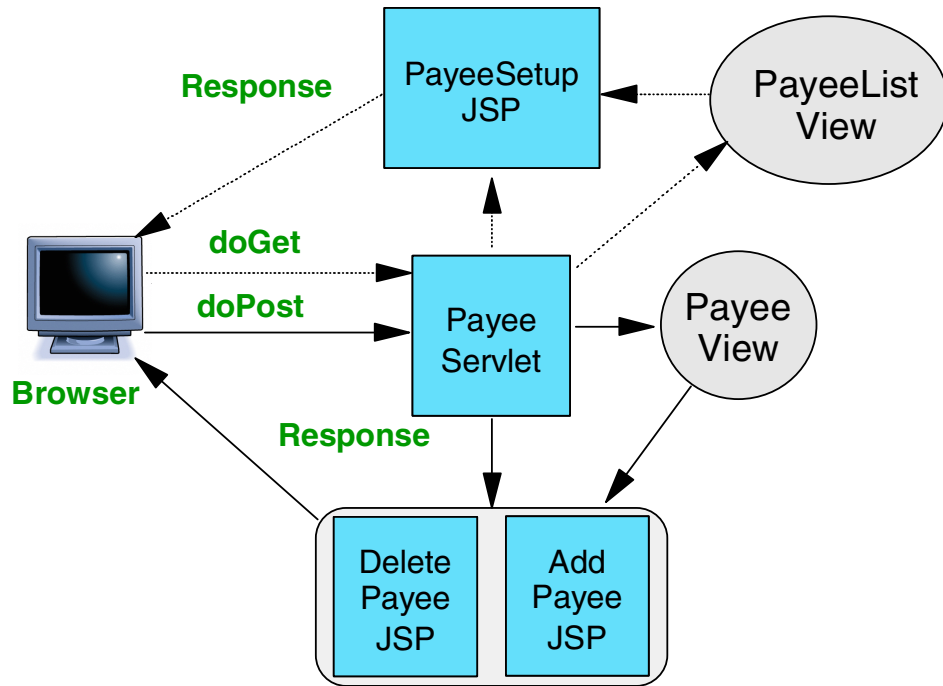


Figure 98. Add/Delete Payee Servlet Architecture

Figure 99 shows the sequence of calls when the doGet method of the PayeeServlet is invoked.

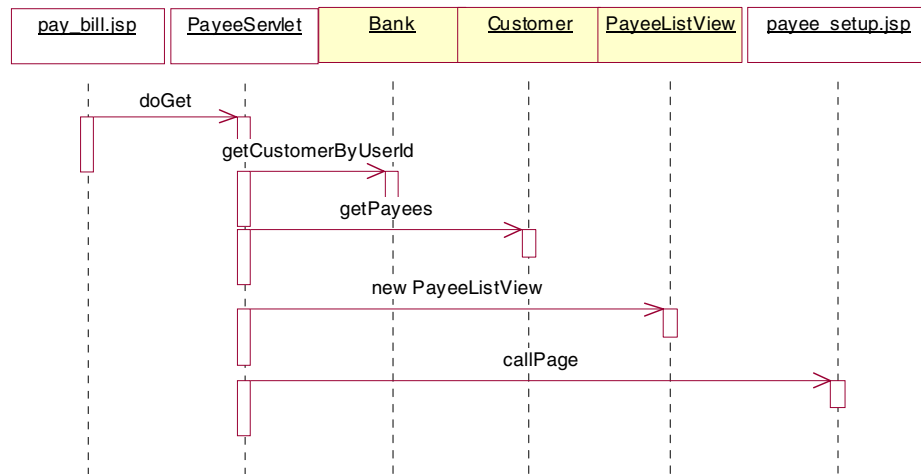


Figure 99. Payee Servlet doGet Interaction

When the user clicks the "Add Payee" or "Delete Payee" button in the Payee Setup page, the doPost method of the PayeeServlet is invoked (Figure 100) with the appropriate command. Based on the command (add or delete) the user is redirected to either the AddPayee page or the DeletePayee page.

When Add Payee is pressed, the PayeeServlet calls the Bank's getPayeeAccounts method in order to obtain all the potential new payees for the customer. Once the user chooses the new account, the PayeeServlet invokes the Customer's addPayee method, passing the new Payee object in order to add it to the Customer Payee List.

In the case of a delete payee action, the PayeeServlet invokes the Customer's getAccountByID method, passing it the AccountID obtained from the user in the PayeeSetup JavaServer Page. After having retrieved the payee, the PayeeServlet invokes the Customer's removePayee method, passing the payee reference.

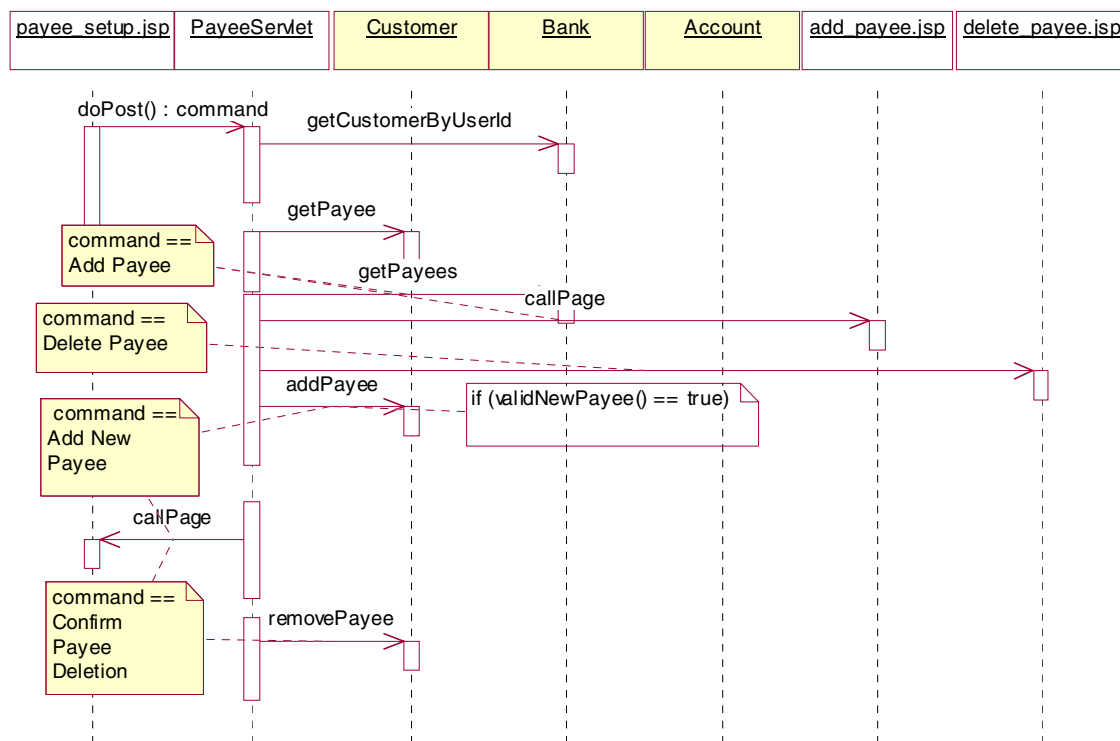


Figure 100. PayeeServlet doPost Sequence

5.10.2 Payee Servlets

The Payee subsystem is composed of the PayeeServlet.

PayeeServlet

The PayeeServlet manages all requests coming from the JavaServer Pages in the Payee subsystem. It decides which step to perform after the user submits his request. The PayeeServlet manages the following actions:

- Add Payee
- Delete Payee
- Confirm Payee Deletion
- Submit New Payee
- Cancel

Table 14 shows the PayeeServlet methods and Table 15 shows the collaborating objects.

Table 14. PayeeServlet Methods

Method	Description
doGet	Displays PayeeSetup.jsp
doPost	Performs the payee addition or deletion.

Table 15. PayeeServlet Collaborators

Class	Description
Customer	Used to get the source and target bank accounts.
BankHome	Provides a reference to the bank.
Bank	Used to get a reference to a customer object.
CustomerView	Used to store customer information in the session.
PayeeAccount	Payee account information.
PayeeAccountViewList	Used to store information for a set of accounts in the request object.
HttpRequest	Provide request information.

PayeeServlet Error Handling

If the user enters incorrect parameters, the Payee Setup page is displayed with the appropriate error. For any other errors, the `callErrorPage` method is called.

PayeeServlet Methods

doGet

The `doGet` method displays the current payee list and allows the user to select an account to delete or choose Add Payee.

```
public final void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
{
    String message = " ";
    itso.bank.util.CacheControl.setCache( res, true);
    HttpSession session = req.getSession(false);
    if( session == null){
        callErrorPage( req, res,
            new Exception("Null session in Account servlet"));
        return;
    }
    CustomerView customerView = (itso.bank.viewobjects.CustomerView)
        session.getValue("customer");
    try {
        Customer customer = BankHome.getBank().getCustomerById(
            customerView.getUserId());
```

Use a view object to hold the account list.

```
        PayeeAccountViewList payeeList =
            new PayeeAccountViewList( customer.getPayees());
        ((com.sun.server.http.HttpServiceRequest)req).
            setAttribute( "accounts", payeeList);
    }
    catch (Exception e) {
        callErrorPage( req, res, e);
        return;
    }
    ((com.sun.server.http.HttpServiceRequest)req).
        setAttribute("message", message);
    callPage( source, req, res);
}
```

doPost

The `doPost` method provides either the Delete Confirmation page and deletes the payee or the Add Payee page and adds the payee.

```

public final void doPost(HttpServletRequest req, HttpServletResponse res)
    throws javax.servlet.ServletException, java.io.IOException
{
    String message = "";
    Customer customer = null;
    PayeeAccount payee = null;
    String destination = source;
    BankServlet servlet = null;
    String command = null;
    String payeeID = null;
    if (req.getParameter("btnSubmit") != null){
        command = req.getParameter("btnSubmit").trim();
    }
}

```

Make sure a Payee was selected.

```

    if (req.getParameter("payee") != null){
        payeeID = req.getParameter("payee").trim();
    }
    else{
        message = "Please Select the Payee you want to Delete";
        destination = source;
    }
    itso.bank.util.CacheControl.setCache( res, true);
    HttpSession session = req.getSession(false);
    if (session == null){
        callErrorPage( req, res,
            new Exception("Null session in Account servlet"));
        return;
    }
    try
    {

```

Get the Customer object.

```

        CustomerView customerView = (itso.bank.viewobjects.CustomerView)
            session.getValue("customer");
        customer = BankHome.getBank().getCustomerByUserId(
            customerView.getUserId());
        if (command.equals("Add Payee"))
        {

```

Add the potential payees to the request object.

```

            PayeeAccountViewList payeeList =
                new PayeeAccountViewList(
                    BankHome.getBank().getPayeeAccounts());
            ((com.sun.server.http.HttpServiceRequest) req).
                setAttribute( "payees", payeeList);

```

```

        destination = "add_payee";
        message =
"Please Select the Payee you want to Add and click Submit New Payee";
    }
    else if(command.equals("Delete Payee") && payeeID != null)
    {

```

Send the Confirm Payee Deletion page.

```

        payee = (PayeeAccount) customer.getAccountByID(payeeID);
        ((HttpServletRequest)req) .
            setAttribute( "payee", new PayeeAccountView(payee));
        destination = "delete_payee";
        message = "Please Select the Payee you want to Delete";
    }
    else if(command.equals("Submit New Payee"))
    {
        if (customer.getAccountByID( payeeID) != null)
        {
            message = "Payee with this accountID already exists";
        }
        else if ((payeeID == null) || (payeeID.startsWith(" ")) ||
            (payeeID.equals("")))
        {
            message = "Payee payeeID cannot be blank or start with blank";
        }
        else{
            payee = (PayeeAccount)BankHome.getBank().getAccount ( payeeID);
            customer.addPayee( payee);
        }
    }
    else if(command.equals("Confirm Payee Deletion"))
    {

```

Remove the payee.

```

        payee = customer.getAccountByID( payeeID);
        customer.removePayee( payee);
    }
    if(destination.equals( source)){
        PayeeAccountViewList payeeList =
            new PayeeAccountViewList( customer.getPayees());
        ((com.sun.server.http.HttpServletRequest)req) .
            setAttribute( "accounts", payeeList);
    }
    ((com.sun.server.http.HttpServletRequest)req) .
        setAttribute("message", message);
    callPage(destination, req, res);

```

```

    }
    catch (Exception e)
    {
        callErrorPage( req, res, e);
        return;
    }
}

```

In the class declaration of the PayeeServlet source is defined as:

```
source = "payee_setup";
```

5.10.3 Payee JavaServer Pages

PayeeSetup JavaServer Page

In the PayeeSetup JavaServer Page, the dynamic content does the following:

- Populates the customer's Payee list—The Payee list is built using the following code (without formatting attributes):

```

<table><tr>
<td colspan=3>Payee List</td></tr><tr><td>&nbsp;&nbsp;&nbsp;</td>
<td>Title</td><td>Account Number</td></tr>
<BEAN NAME="accounts" TYPE="itso.bank.viewobjects.PayeeAccountViewList"
INTROSPECT="no" CREATE="no" SCOPE="request"> </BEAN>
<repeat index=count>
<% accounts.getAccounts( count); %>
<tr><td><input type=radio name="payee" value="<insert bean = accounts
property=accounts(count).accountId></insert>">&nbsp;&nbsp;&nbsp;</td>
<td><insert bean = accounts
property=accounts(count).billPaymentTitle></insert></td>
<td>
<insert bean = accounts property=accounts(count).accountId></insert>
</td></tr></repeat></table>

```

- Manages the message area—This is similar to the message function in 5.9, “Transfer Funds” on page 144.

AddPayee JavaServer Page

In the AddPayee JavaServer Page, the dynamic content does the following:

- Populates the potential new payee list—The code for the list is as follows:

```

<BEAN NAME="payees" TYPE="itso.bank.viewobjects.PayeeAccountViewList"
INTROSPECT="no" CREATE="no" SCOPE="request"> </BEAN>
<select id="FormsComboBox" name="payee">
<repeat index=count>
<% payees.getAccounts( count); %>

```



```

<option value=<insert bean = payees
property=accounts(count).accountId></insert>>
<insert bean = payees property=accounts(count).billPaymentTitle>
</insert>
</option>
</repeat>
</select>

```

- Manages the message area—This is similar to the message function in 5.9, “Transfer Funds” on page 144. DeletePayee JavaServer Page

DeletePayee JavaServer Page

In the DeletePayee JavaServer Page, the dynamic content is as follows:

```

<BEAN NAME="payee" TYPE="itso.bank.viewobjects.PayeeAccountView"
INTROSPECT="no" CREATE="no" SCOPE="request"> </BEAN>
<INPUT TYPE="hidden" NAME="payee" VALUE="<insert bean=payee
property=accountId></insert>">
<table><tr><td>Payee Name</td>
<TD><insert bean=payee property=billPaymentTitle></insert>
</TD></tr><tr><td>Account #:</td>
<TD><insert bean=payee property=accountId></insert>
</TD></tr></table>

```

5.11 User

The User subsystem of the HBA is used to maintain passwords through the Change Password page (Figure 101). When users go to the Change Password page, they are presented with a choice box and three fields. The choice box is used to control whether to change the login or transaction password. The first text field is used to enter the current password. The next two fields are the new password and the password confirmation.

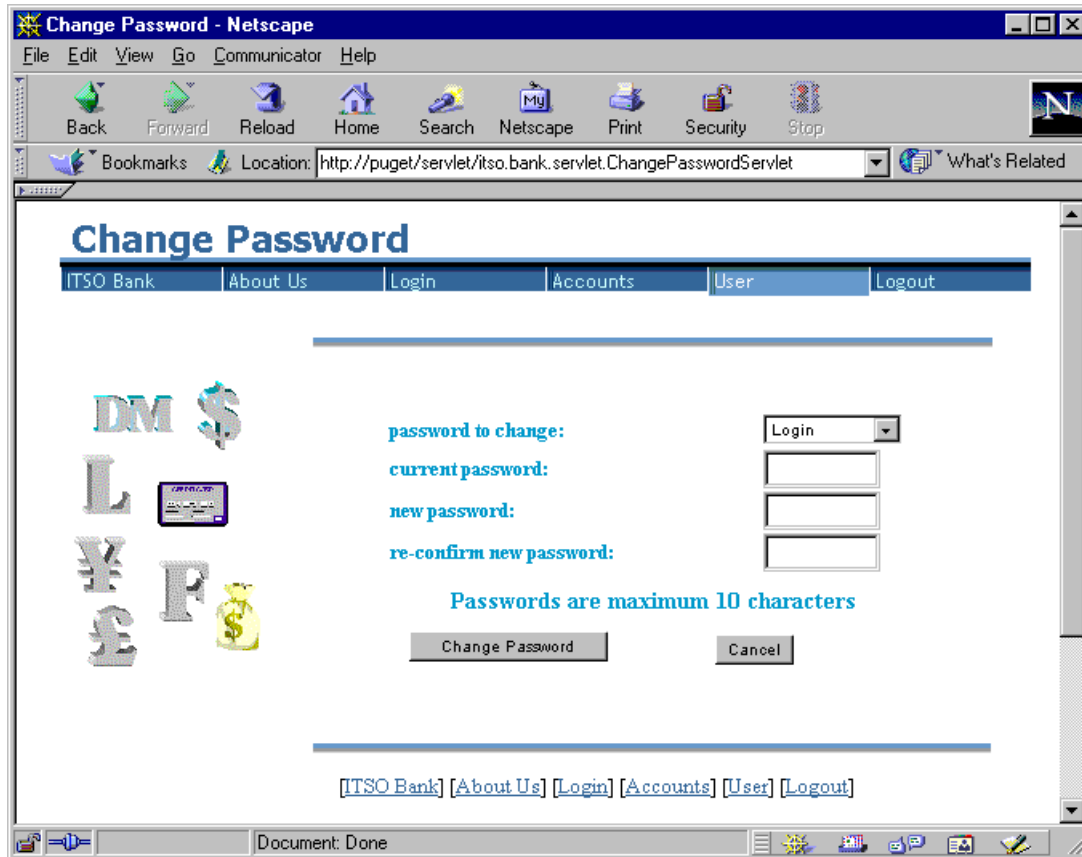


Figure 101. Change Password Page

5.11.1 User Interaction

The User subsystem is composed of one servlet and two JavaServer Pages, as shown in Figure 102.

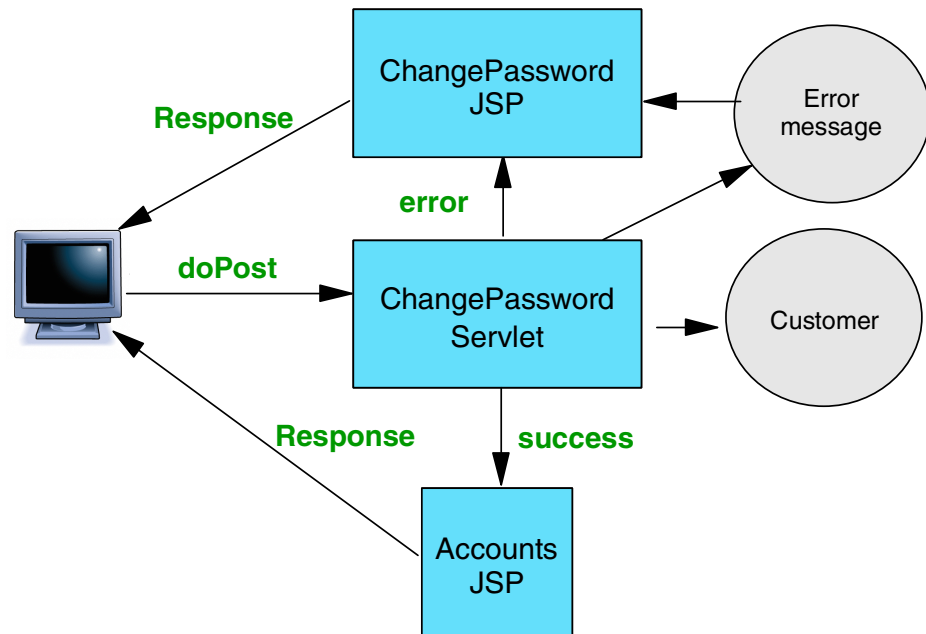


Figure 102. Change Password Architecture

Once the user selects the password type, enters the passwords and, submits the form, the ChangePasswordServlet's doPost method is invoked. The doPost method retrieves the parameters and validates the passwords. If validation succeeds, the user's password is changed, and the user is sent to the Accounts JSP. If the attempt is invalid, the user is sent back to the Change Password Page. This interaction is shown in Figure 103.

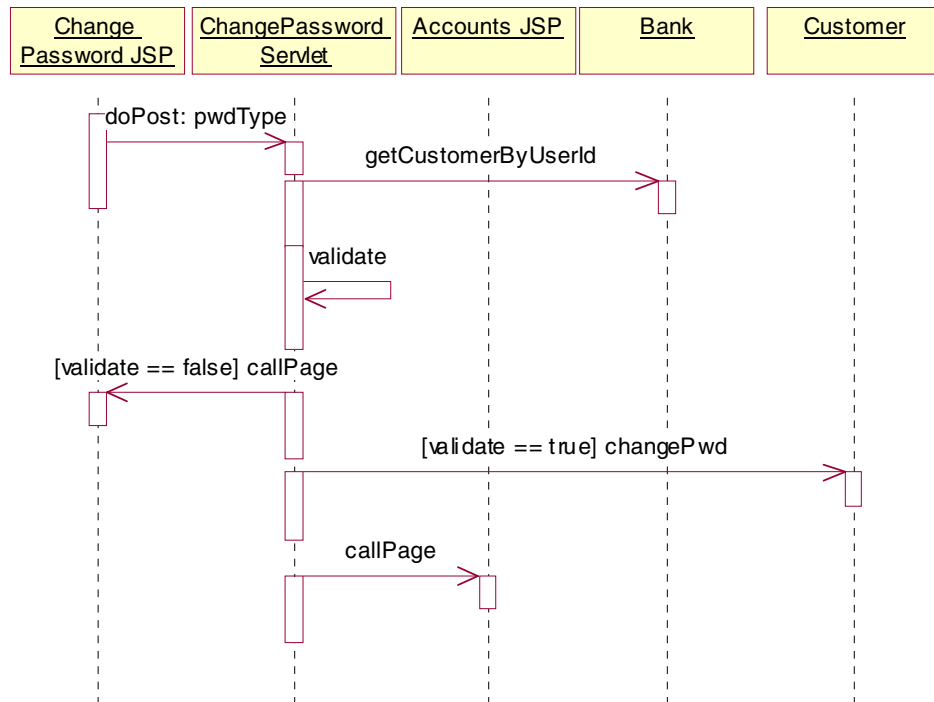


Figure 103. Change Password Interaction

5.11.2 User Servlets

The User subsystem is made up of the ChangePasswordServlet.

ChangePasswordServlet

The ChangePasswordServlet authenticates the user's password, validates the data the user enters, and changes the passwords.

Table 16 shows the ChangePasswordServlet methods and Table 17 shows the collaborating objects.

Table 16. ChangePasswordServlet Methods

Method	Description
doGet	Displays Change Password page.
doPost	Performs the password change.
changePwd	Calls the appropriate changePassword method on the Customer object.
validate	Validates the password values entered by the user.

Table 17. ChangePasswordServlet Collaborators

Class	Description
Customer	Used to change the passwords.
CustomerView	Stored in session to hold customer ID.
HttpSession	Used to store the CustomerView.
HttpRequest	Provides the request information.

ChangePasswordServlet Error Handling

If the validation fails, the Change Password JSP is displayed with an appropriate message; otherwise the callErrorPage method is invoked.

ChangePasswordServlet Methods

doPost

The doPost method calls the validate and changePassword methods.

```
public final void doPost(HttpServletRequest req, HttpServletResponse res)
    throws javax.servlet.ServletException, java.io.IOException {
    String passwordType = req.getParameter("lstPasswordType").trim();
    String currentPassword = req.getParameter("txtCurrentPassword").trim();
    String newPassword = req.getParameter("txtNewPassword").trim();
    String reNewPassword = req.getParameter("txtReNewPassword").trim();
    String message = null;
    itso.bank.util.CacheControl.setCache( res, true);
    HttpSession session = req.getSession(false);
    if( session == null){
        callErrorPage( req, res, new Exception(
            "Null session in Change Password servlet"));
        return;
    }
    CustomerView customerView = (itso.bank.viewobjects.CustomerView)
        session.getValue("customer");
    try {
```

```
Customer customer = BankHome.getBank().getCustomerByUserId(
    customerView.getUserId());
```

Validate the data.

```
if ((message = validate( customer, passwordType,
    currentPassword, newPassword, reNewPassword)) == null) {
```

Change the password.

```
    changePwd( customer, passwordType,
        currentPassword, newPassword);
    ((com.sun.server.http.HttpServletRequest)req).
        setAttribute( "message", "");
```

Send the user to the Accounts JSP page.

```
        callPage("accounts", req, res);
    } else {
    ((com.sun.server.http.HttpServletRequest)req).
        setAttribute( "message", message);
```

Send the user to the Change Password JSP.

```
        callPage("change_password", req, res);
    }
    } catch (Exception e) {
        callErrorPage( req, res, e);
        return;
    }
}
```

doGet

The doGet method displays the Change Password page.

```
public final void doGet(HttpServletRequest req, HttpServletResponse res)
throws
    javax.servlet.ServletException, java.io.IOException {
    String message = "";
    itso.bank.util.CacheControl.setCache( res, true);
    try{
        ((com.sun.server.http.HttpServletRequest)req).
            setAttribute("message", message);
        callPage("change_password", req, res);
    } catch (Exception e) {
        callErrorPage( req, res, e);
        return;
    }
}
```

validate

The validate method validates the current and new passwords.

```
public final String validate(itso.bank.common.Customer customer,
    String passwordType, String currentPwd, String newPwd,
    String confirmedNewPwd)
    throws ITSOBankCommunicationException, ITSOBankException {
    String message = null;
    if (customer != null)
    {
        if (currentPwd.equals(""))
        {
            message = "Current Password cannot be blank";
        }
        if (passwordType.equalsIgnoreCase("application"))
        {
            if (!customer.checkLoginPassword(currentPwd))
            {
                message = "Current login password is not correct";
            }
        }
        if (passwordType.equalsIgnoreCase("authorization"))
        {
            if (!customer.checkTransactionPassword(currentPwd))
            {
                message = "Current transaction password is not correct";
            }
        }
        if (newPwd.equals(""))
        {
            message = "New Password can not be blank";
        }
        if (confirmedNewPwd.equals(""))
        {
            message = "Confirmed New Password can not be blank";
        }
        if (currentPwd.equals(newPwd))
        {
            message = "Current Password cannot equal New Password";
        }
        if (!newPwd.equals(confirmedNewPwd))
        {
            message = "New Password and Confirmed New Password must be same.";
        }
    }
    return message;
}
```

changePwd

The changePwd method performs the password change.

```
public final void changePwd( Customer customer, String type,
    String oldPwd, String newPwd)
    throws ITSOBankCommunicationException, ITSOBankException
{
    if (type.equalsIgnoreCase("application")) {
        customer.changeLoginPassword( oldPwd, newPwd, newPwd);
    } else {
        customer.changeTransactionPassword(oldPwd, newPwd, newPwd);
    }
}
```

5.11.3 User JavaServer Pages

User JavaServer Page

The User page simply allows the user to select the Change Password function.

Change Password JavaServer Page

The Change Password page allows the user to select which password to change and enter the new password. The JSP code is similar to preceding pages.

5.12 Utility Classes

There are several utility classes used in the HBA and defined in the itso.bank.util package. They are:

- CacheControl—Handle the caching of the JavaServer Pages
- Formatter—Format dates and currencies for JavaServer Pages
- XMLConfigUtil—Access to XML configuration information

5.12.1 CacheControl

The CacheControl class encapsulates the control of the cache in one place.

```
import javax.servlet.http.HttpServletResponse;
public final class CacheControl {
    public static final void setCache( HttpServletResponse res,
        boolean noCache) {
        setCache( res, noCache, 0);
    }
}
```



```

public static final void setCache( HttpServletResponse res,
    boolean noCache, int expiration) {
    if( noCache){
        res.setHeader("pragma", "no-cache");
        res.setHeader("Cache-Control", "no-cache");
        res.setHeader("Expires", "0");
    }
    else if( expiration > 0){
        res.setHeader("Expires", Integer.toString(expiration));
    }
}
}
}

```

5.12.2 Formatter

The Formatter class is used to format attributes in the view beans.

```

import java.math.BigDecimal;
import java.util.Date;
import java.text.DateFormat;
import java.util.TimeZone;

public final class Formatter {
    private static final String DEFAULT_ZONE = "GMT";
    public final static String getAsCurrency( double amount) {
        java.text.NumberFormat nf = new java.text.DecimalFormat();
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);
        return "$" + nf.format(amount);
    }
    public final static String getAsCurrency( BigDecimal amount) {
        return getAsCurrency(amount.doubleValue());
    }
    public final static String getFormattedDate( Date date) {
        return getFormattedDate( date, DEFAULT_ZONE);
    }
    public final static String getFormattedDate( Date date, String zone) {
        DateFormat df = new java.text.SimpleDateFormat("MM/dd/yyyy");
        df.setTimeZone(TimeZone.getTimeZone(zone));
        return df.format(date);
    }
}
}

```

5.12.3 XMLConfigUtil

The XMLConfigUtil class provides the getPageURI method to get URIs (URLs) from the XML configuration information to be used in the sendRedirect method.

```
package itso.bank.util;
import com.ibm.servlet.config.*;
import com.ibm.servlet.*;
import javax.servlet.*;
public class XMLConfigUtil {
    private XMLServletConfig servletconfig;
    private PageList pagelist;
    private String description;
    public XMLConfigUtil( Servlet servlet)
    {
        ServletContext servletcontext =
            servlet.getServletConfig().getServletContext();
        try{
            this.servletconfig(XMLServletConfig)servlet.getServletConfig();
        }
        catch(ClassCastException ex){
            servletcontext.log("Bad xml servlet config");
        }
        org.w3c.dom.Element element =
            servletconfig.getElement (PageList.ELEMENT_PAGELIST);
        if(element == null)
        {
            servletcontext.log("no pagelist found");
        }
        else
        {
            pagelist = new PageList(element, servletconfig);
            return;
        }
    }
    public String getPageURI( String pagename){
        return pagelist.getPageURI( pagename);
    }
}
```

Chapter 6. Deploying the Home Banking Application

This chapter describes the deployment of the HBA. The subsystems of the HBA were unit tested using VisualAge for Java. The complete testing was done on the deployed versions on the WebSphere Application Server.

To deploy our Home Banking Application, we need a Web server and a servlet engine. We have chosen to deploy the HBA on two platforms: Windows NT Workstation and AIX. Both platforms run the WebSphere Application Server, and we will use two different Web servers:

- Netscape Enterprise Server (NES) on Windows NT
- IBM HTTP Server on AIX and Windows NT

6.1 Installing the Servers

Refer to the Web server and WebSphere Application Server documentation for installation instructions.

6.2 Configuring the Servers

We need to configure the two Web servers as well as the WebSphere Application Server on each platform.

6.2.1 Configuring the Web Servers

Configuring Netscape Enterprise Server (Windows NT)

Configuring NES on Windows NT consisted of:

- Setting the Secure Sockets Layer (SSL) protocol
- Configuring the server properties

Setting the Secure Sockets Layer (SSL) Protocol

To enable SSL:

1. We generated our server's key-pair file (containing public and private keys) and used it to request and install the server certificate. The password specified when creating the key-pair is used for starting the server. Refer to the NES documentation for the steps to follow to generate the key-pair file.
2. We requested a certificate from a Certificate Authority, specifying the previously generated key-pair file. We requested a free trial Web server digital certificate from VeriSign. For the procedure to follow, refer to the

VeriSign Web site at the URL <http://www.verisign.com> and the NES Administrator Guide.

3. We installed the VeriSign certificate on the Web Server. The certificate was encrypted with our public key so that only we can decrypt it. The server used our key-pair file password to decrypt the certificate during the installation. For the details about the installation procedure, refer to the NES Administrator Guide.
4. After having generated a key-pair file and installed our certificate, we activated SSL on our Web server. Again, refer to the NES documentation for the details.

Configuring the Server Properties

We installed the Web server following the instructions in the NES installation documentation. After the server was installed we used the Netscape Administration Server at the URL <http://hostname:2720>. This was the port we chose for the administration server when we installed NES (Figure 104).

The name of our host was *barium*.

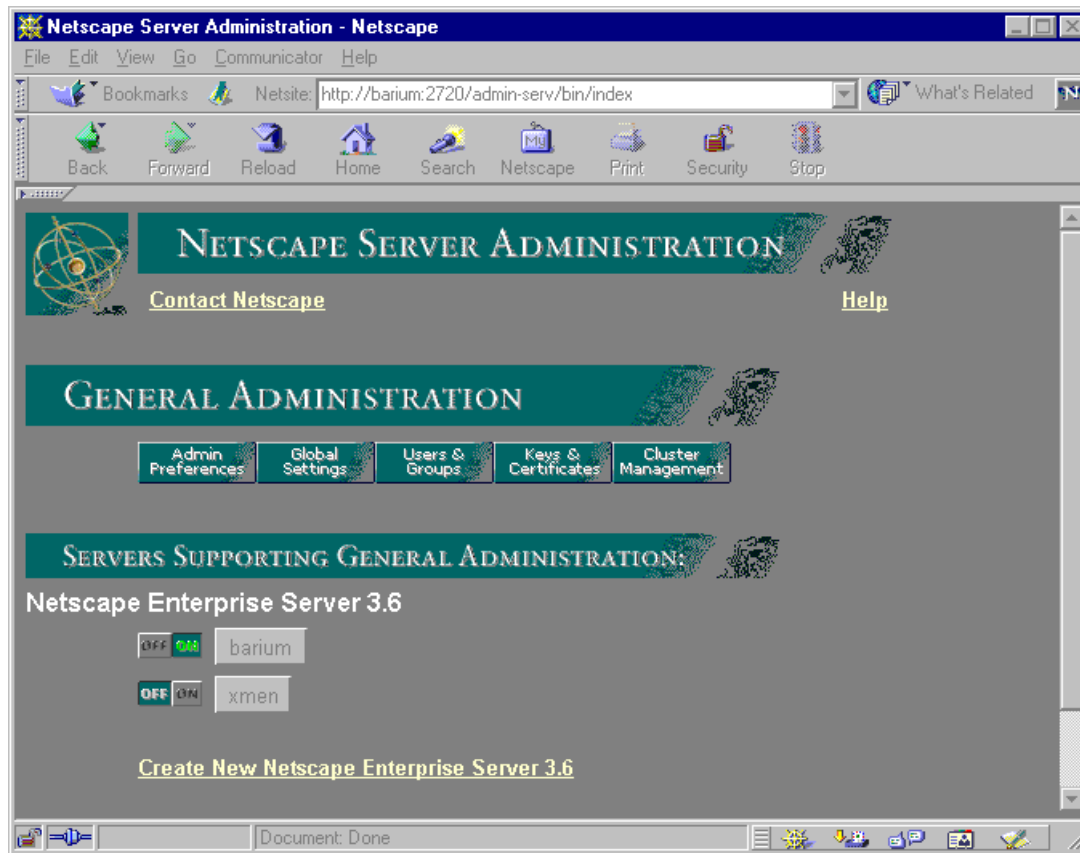


Figure 104. Netscape Administration Server on Windows NT

We chose to add a new Web server by clicking on the link **Create New Netscape Enterprise Server 3.6**. We entered all the values for our new Web server on the resulting page (Figure 105). We set the server port to port 443. Port 443 is the default port used for SSL communication. However, the Web server does not need to run on port 443 to enable SSL. For other settings, refer to the Netscape Administration Server documentation.

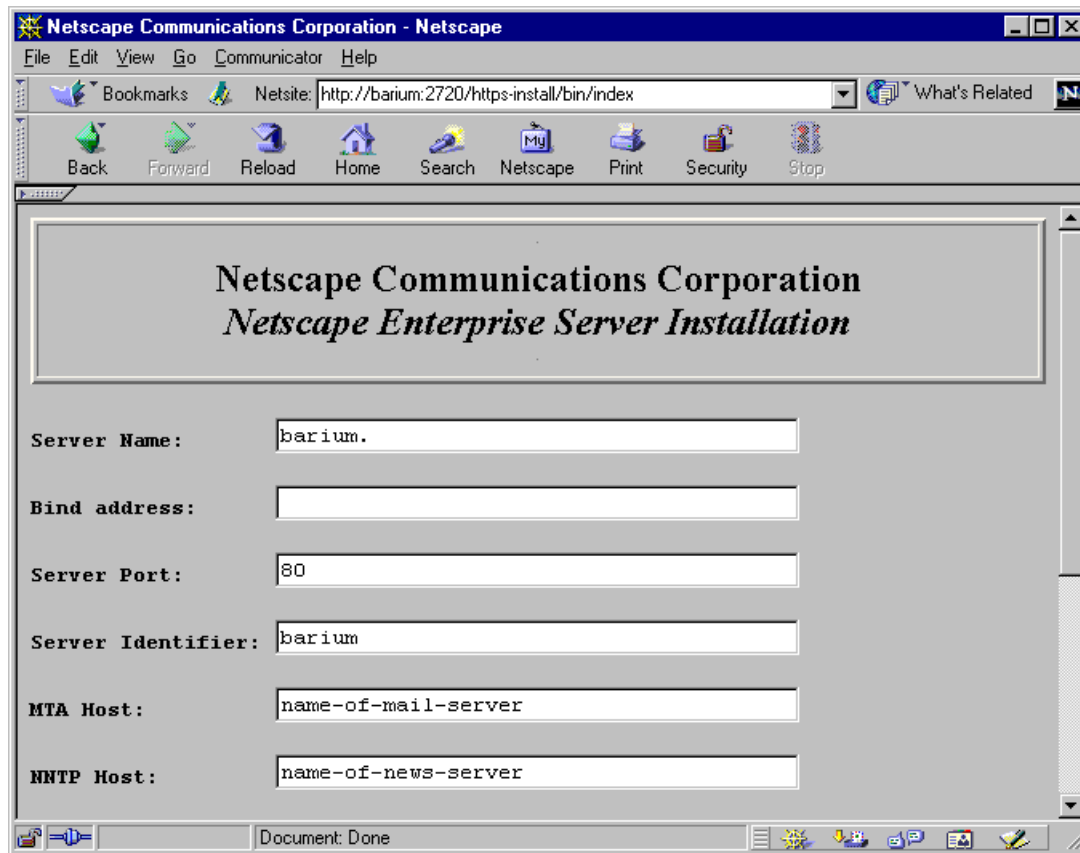


Figure 105. Netscape Enterprise Server (Create Server Menu)

Once the Web server has been created it is displayed on the main console of the administration page (Figure 104 on page 171). We then configured the document root directory for the Web server. We clicked on the **barium** button for "**barium**" and were presented with the settings for our Web server (Figure 106).

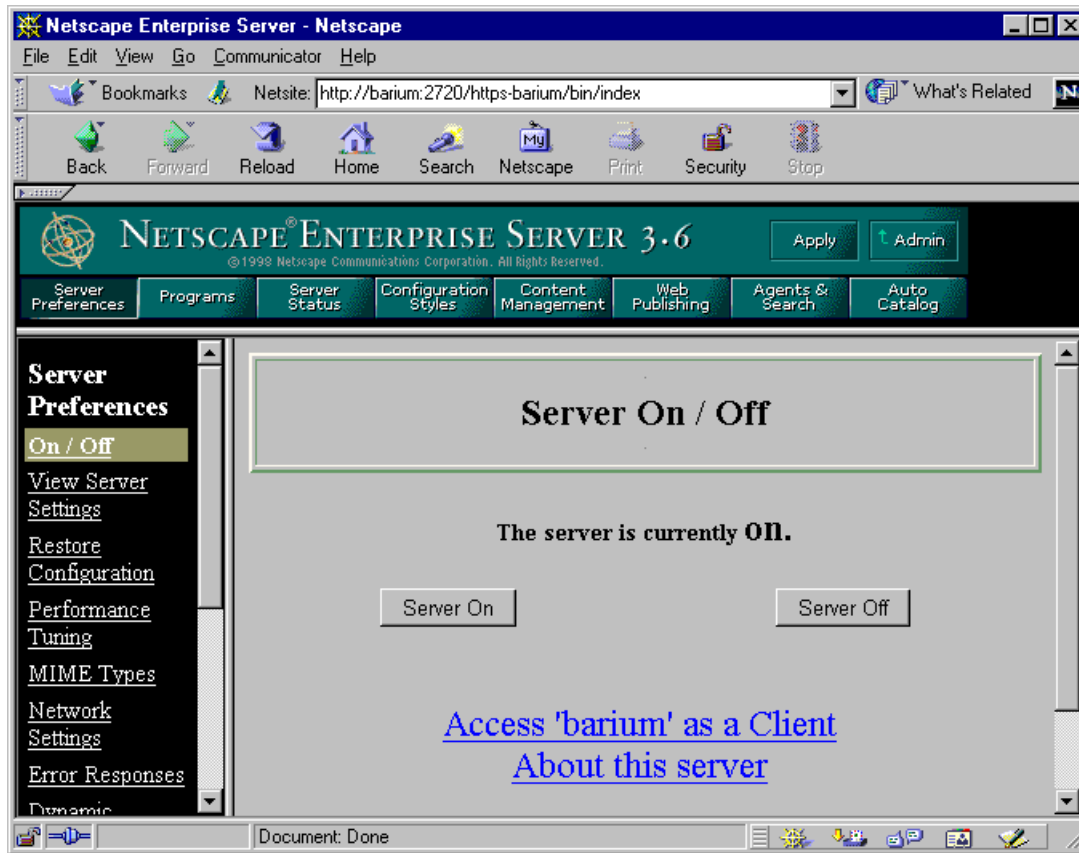


Figure 106. Web Server Menu

We then clicked on the **Content Management** button on the toolbar (Figure 107) and were presented with the Primary Document Directory page. In the Primary directory field, we typed `c:/www/html` for the document root directory on our server and clicked **OK**. (When we publish our Web site, we will publish it to this directory.) We then clicked **Save and Apply**. Now the server starts with our changes (Figure 108).

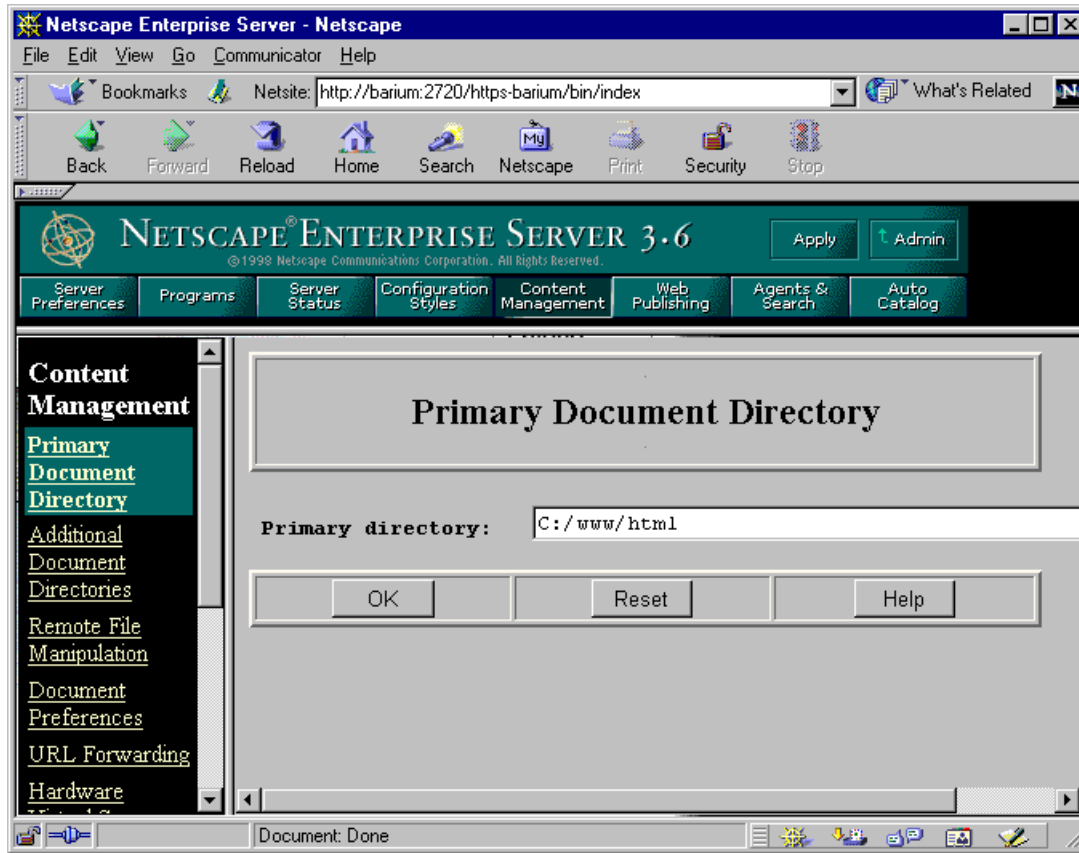


Figure 107. Setting the Document Root Directory

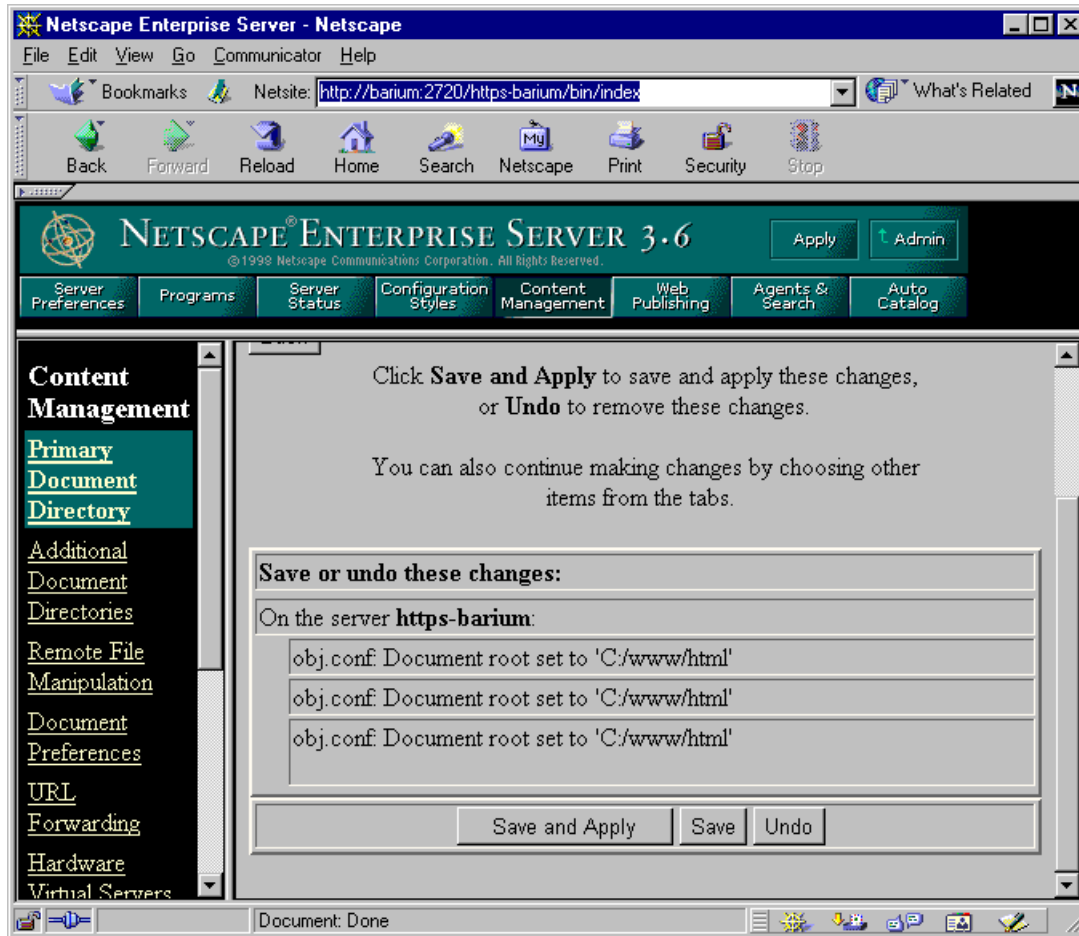


Figure 108. Applying the Document Root Directory Changes.

Configuring the IBM HTTP Server (AIX)

We chose to also deploy our application to the IBM HTTP Server on AIX to show the application running in a very scalable and robust environment. For detailed instructions on the following procedures, see the IBM HTTP Server documentation. We also deployed the HBA on the IBM HTTP Server running on Windows NT and the instructions are similar. The configuration steps for the IBM HTTP Server were similar to those for NES:

- Set the Secure Sockets Layer protocol.
- Configure the server properties.

Setting the Secure Sockets Layer Protocol

To enable SSL on our server, we completed the following steps:

1. We created a new key database, specifying a key database password using the IKEYMAN software (part of the IBM HTTP Server).
2. We created a new self-signed certificate using IKEYMAN and configured it as the default certificate in the database.
3. We set up a secure network connection for the IBM HTTP Server and stored the encrypted database password in a stash file.
4. We registered the server key database with the server. In order to perform this operation we need to change the configuration file (conf/httpd.conf) in the following sections (Steps 1, 2, 6 and 8 are not documented in the documentation that comes with the IBM HTTP Server):
 1. Add the LoadModule `ibm_ssl_module` `modules/IBMModuleSSL.dll` statement in order to load the DLL as indicated in the `httpd.conf.sample.ssl` file in the IBM HTTP Server conf directory.
 2. Change the port number from 80 to 443 (so that the HTTP protocol is disabled).
 3. Ensure that the line `Listen 443` is uncommented.
 4. Place the host name of the server in the virtual host stanza for port 443.
 5. Ensure that the `SSLEnable` directive is uncommented in the virtual host stanza.
 6. Set the `"SSLServerCert CertificateName"` directive.
 7. Set the `Keyfile` directive. It belongs outside of the virtual host stanza.
 8. Change the `Directory` directive to whatever has been set in the `DocumentRoot` directive.

The Web server will now run with the following settings:

- Support for SSL connections is turned on.
- Port 443 is used for SSL connections and port 80 is disabled.
- Client authentication is disabled.
- The server will use the strongest encryption level supported by both the client and the server.

Configuring the Web Server

We only had to do one thing to configure the IBM HTTP Server: set the `DocumentRoot` directory. We did this by editing the `httpd.conf` file in the IBM

HTTPServer conf directory and modifying the DocumentRoot property in the file to point to the root directory of the bank:
/usr/lpp/HTTPServer/share/htdocs/bank.

We then restarted our server in order for the changes to take affect. Under AIX we stopped and started the server using the command line. Under Windows NT we stopped and started the service from the Services console.

6.2.2 Deploying the HBA Application Classes

In our HBA application we have several packages in the WebSphere Bank Application project (Figure 109). We decided to deploy the application classes as one JAR file and the servlets as individual class files.

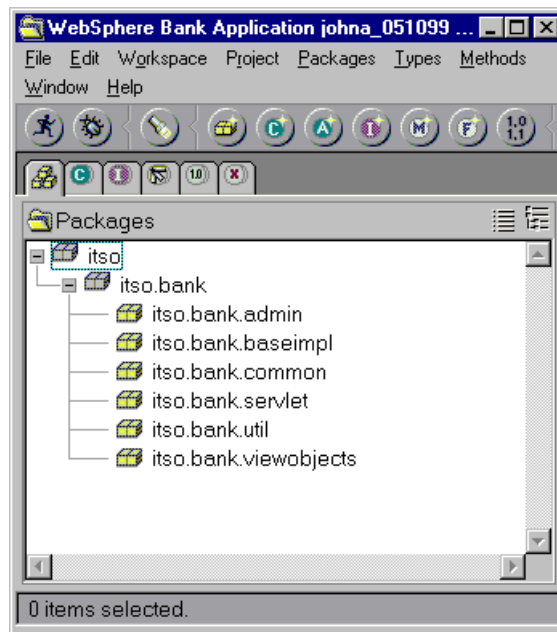


Figure 109. WebSphere Bank Application Packages

To create and export the JAR file, select the packages and select **Packages**→**Export**. This brings up the Export SmartGuide. Select the **Jar File** radio button and click **Next**. In the next dialog select the **.class** checkbox and specify the full path of the Jar file in the Jar file field. You can export the Jar file to any directory because they must then be imported into WebSphere Studio. Click **Finish** to create and export the Jar file (Figure 110).

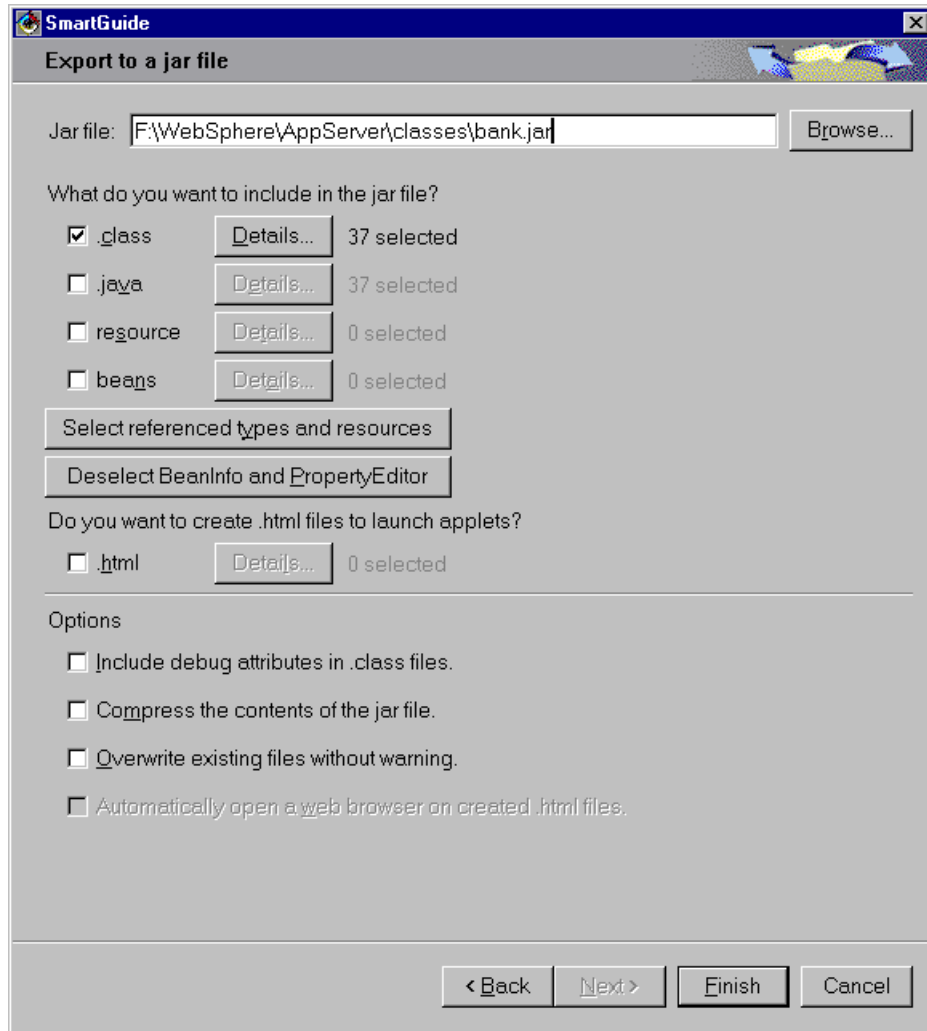


Figure 110. VisualAge SmartGuide

Now the servlets package needs to be exported. Select the servlets package and select **Packages**→**Export**. Select the **Directory** radio button and click **Next**. In the next dialog select the **.class** checkbox and enter the path where you want to temporarily store the servlets. Click **Finish** to export the classes.

Once the JAR file and servlets are exported, simply drag them and the servlet configuration files into the appropriate folders in WebSphere Studio as shown in Table 18.

Table 18. WebSphere Studio Code Folders

File(s)	WebSphere Studio Folder
bank.jar	classes
All servlet class files	servlet
All servlet configuration files	servlet

6.2.3 Deploying the HBA Web Site

The Web site for the HBA application is managed under WebSphere Studio. WebSphere Studio provides a publishing wizard to deploy the site. In order to deploy the HBA Web site open it in WebSphere Studio and select the Publishing view. In the Publishing view select the Assembly Stage that contains the server that has your publishing targets (Test in our case) and then select **File**→**Publish Whole Project**.

6.2.4 Configuring the WebSphere Application Server

To install WebSphere, refer to the documentation provided at <http://www.software.ibm.com/websphere> or the product documentation. After installing WebSphere we invoked the Administration facility by typing the URL <http://hostname:9527> to access the Administration Login page (Figure 111) and typed `admin/admin` for the user name and password.



Figure 111. WebSphere Administration Page

After logging into WebSphere we need to perform the following operations:

1. Add the bank.jar file to the WebSphere Application Server classpath.
2. Add the BankServlet to the WebSphere Application Server configuration.
3. Configure the BankServlet to load on WebSphere Application Server startup.
4. Add an implementation parameter to the BankServlet to specify the implementation we are using.
5. Set the timeout parameter for sessions to 30 minutes.

Add the bank.jar File to the Classpath

To add the BankServlet to WebSphere we need to go to the Servlet Management facility in WebSphere. From the Introduction page in WebSphere go to the Java Engine section found under Setup (Figure 112).

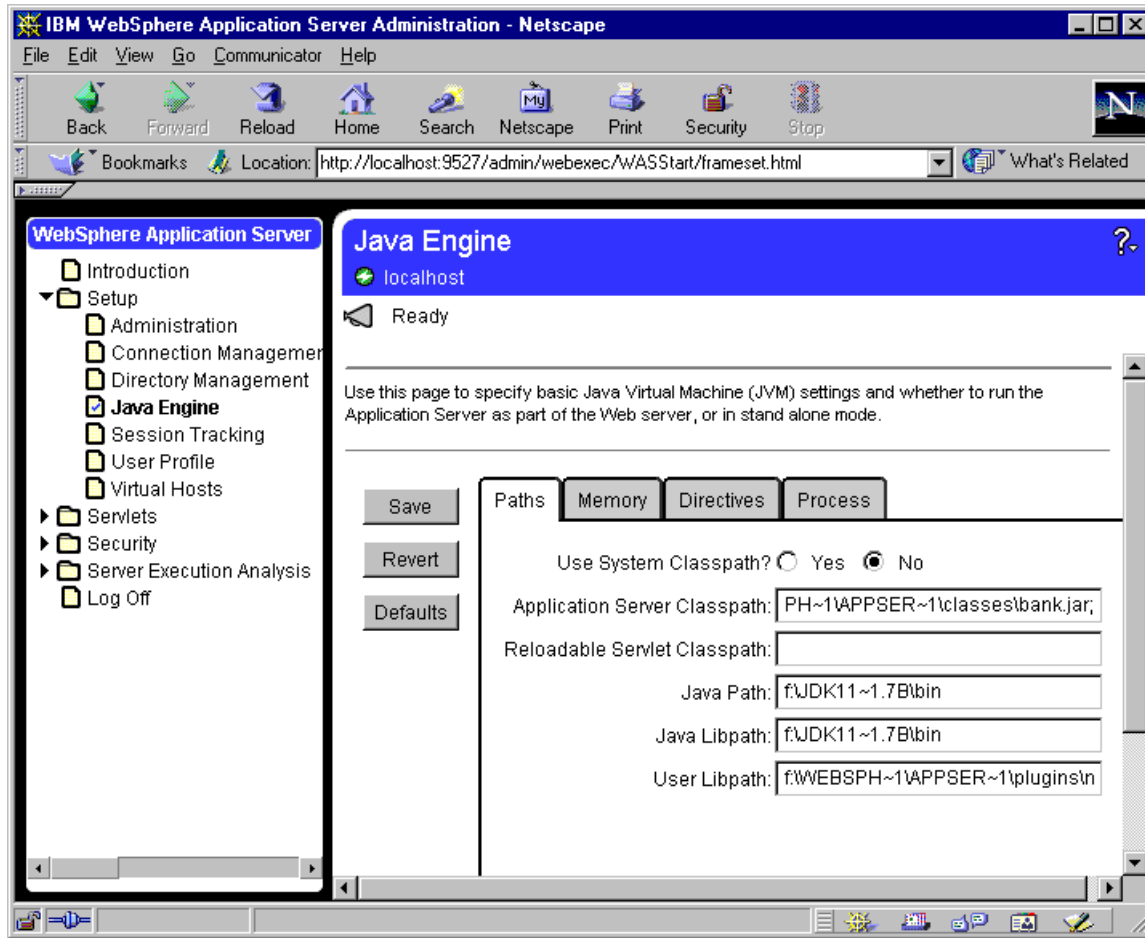


Figure 112. Adding bank.jar to the Classpath

Add the full path of bank.jar to the Application Server Classpath field. If you are working in an NT environment, make sure you use the 8 character DOS filename conventions for the path and that you end the line with a semicolon. You will need to restart the server now in order to perform the next step.

Adding the BankServlet to WebSphere Application Server

To add the BankServlet to WebSphere we need to go to the Servlet Management facility in WebSphere. From the Introduction page in WebSphere, go to the Servlet Configuration Section found under Servlets (Figure 113).

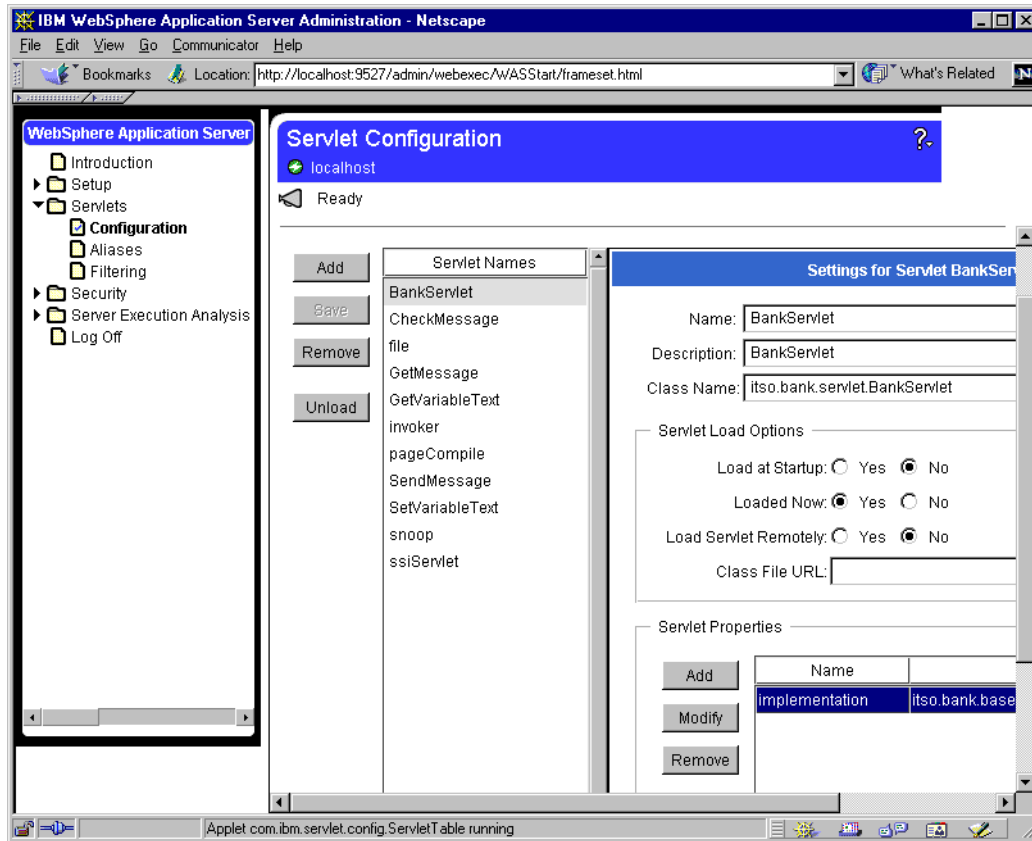


Figure 113. Servlet Configuration Facility

Click on the **Add** button and you are presented with the Add a New Servlet dialog (Figure 114). Enter BankServlet for the Servlet Name and itso.bank.servlet.BankServlet for the Servlet Class and click **Add**. The servlet is added to the list of servlets in the Servlet Configuration Facility.

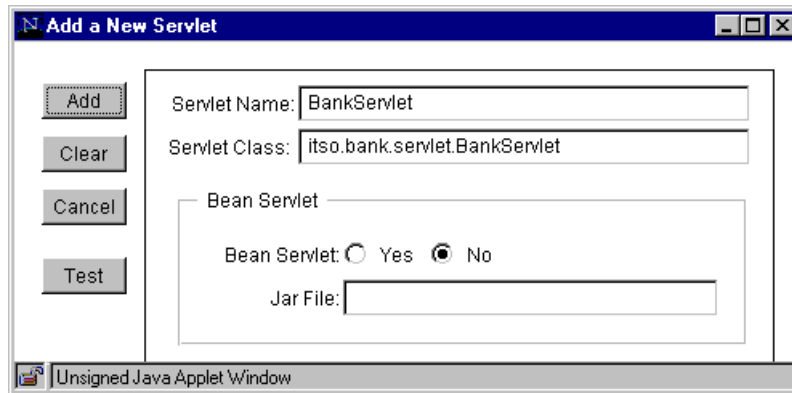


Figure 114. Add a New Servlet Dialog

Create the Implementation Parameter

To create the implementation parameter for the BankServlet select the BankServlet from the list of servlets in the Servlet Configuration Facility (Figure 113 on page 182). Then click **Add** under Servlets Properties. The cursor starts to blink in the Name Column under Servlets Properties. Enter a value of "implementation" for the Name. Click on the Value column and enter a value of "itso.bank.baseimpl". This is the name of the implementation package. After entering these values, click on the **Save** button to save the changes made to the BankServlet.

Load BankServlet on Startup

To instruct WebSphere to load the BankServlet at startup, click on **BankServlet** in the Servlet Configuration Facility (Figure 113 on page 182). Then click the **Load at Start** radio button. After making these changes, click on the **Save** button to save the changes to the servlet and then restart the server.

Set the Timeout Parameter

To set the timeout on sessions (the amount of time a session is idle before it is invalidated), select the **Intervals** tab in the **Setup**→**Session Tracking** page of the WebSphere Administration facility. In the Invalidate Time field enter 1800000 to set the timeout to 30 minutes or whatever time (in milliseconds) you have determined as a valid timeout parameter.

The HBA is now ready to be accessed by users.

Appendix A. HBA Use Cases

The use cases were defined during the analysis stage of HBA development and differ slightly from the design and implementation models. For example, use case UC02 was never implemented.

Each use case indicates a sequence of actions performed by this system in response to an event initiated by an actor to the system. Where not indicated, the Actor is the Authenticated User.

UC01 - Login/Authentication

Actors: User

Definition:

User enters userid and login password, submits entries

System responds

Uses/Extends: None

UC01A - Successful Login/Authentication

Actors: User

Definition:

User enters userid and login password, submits entries

System responds with the Accounts Menu Screen

Extends: Login/Authentication

Uses: None

UC01B - Unsuccessful Login/Authentication

Actors: User

Definition:

User enters userid and login password, submits entries

System responds with the Login Error screen

Extends: Login/Authentication

Uses: None

UC02 - Get Accounts Menu

Definition:

User chooses the Accounts option

System responds with the Accounts screen

Extends: None

Uses: Login Authentication

UC03 - Get Account Balance

Definition:

User chooses the Account Information option

System responds with the Account Information screen
User chooses an account
User chooses the Account Balance option
System responds with the Account Balance screen
Extends: None
Uses: Login/Authentication; Get Accounts Menu

UC04 - Get Account History

Definition:
User chooses the Account Information option
System responds with the Account Information screen
User chooses an account
User chooses the Account History option
System responds with the Account History screen
Extends: None
Uses: Login/Authentication; Get Accounts Menu

UC05 - Transfer funds between user accounts

Definition:
User chooses the Transfer Funds option
System responds with the Transfer Funds screen
User chooses source account
User chooses target account
User enters amount
User enters the Transaction password
User submits data
System responds
Extends: None
Uses: Login/Authentication; Get Accounts Menu

UC05A - Successful transfer funds between user accounts

Definition:
User chooses the Transfer Funds option
System responds with the Transfer Funds screen
User chooses source account
User chooses target account
User enters amount
User enters the Transaction password
User submits data
System responds with the Funds Transferred screen reporting information and date as transaction ID
Extends: Transfer funds between user accounts; Get Accounts Menu
Uses: Login/Authentication

UC05B - Unsuccessful transfer funds between user accounts

Definition:

User chooses the Transfer Funds option

System responds with the Transfer Funds screen

User chooses source account

User chooses target account

User enters amount

User enters the Transaction password

User submits data

System responds with the Transfer Funds screen showing an error message

Extends: Transfer funds between user accounts

Uses: Login/Authentication; Get Accounts Menu

UC06 - Pay bill

Definition:

User chooses Pay Bill option

System responds with the Pay Bill screen

User chooses source account

User chooses payee account

User enters amount

User enters the Transaction password

User submits data

System responds

Extends: None

Uses: Login/Authentication; Get Accounts Menu

UC06A - Successful Pay Bill

Definition:

User chooses Pay Bill option

System responds with the Pay Bill screen

User chooses source account

User chooses payee account

User enters amount

User enters the Transaction password

User submits data

System responds with the Bill Paid screen reporting information and date as transaction ID

Extends: Pay bill

Uses: Login/Authentication; Get Accounts Menu

UC06B - Unsuccessful Pay bill

Definition:

User chooses Pay Bill option

System responds with the Pay Bill screen

User chooses source account
User chooses payee account
User enters amount
User enters the Transaction password
User submits data
System responds with the Pay Bill screen showing an error message
Extends: Pay bill
Uses: Login/Authentication; Get Accounts Menu

UC07 - Payee Setup

Definition:
User chooses Pay Bill option
System responds with the Pay Bill screen
User chooses Payee Setup option
System sends the Payee List screen
Extends: None
Uses: Login/Authentication; Get Accounts Menu

UC08 - Payee Setup: add entry

Definition:
User chooses add Payee option
System responds with the Add Payee screen
User chooses an entry from the New Payee List
User submits data
System responds
Extends: None
Uses: Login/Authentication; Get Accounts Menu; Payee Setup

UC08A - Payee Setup: successful add entry

Definition:
User chooses add Payee option
System responds with the Add Payee screen
User chooses an entry from the New Payee List
User submits data
System responds with the updated Modify Payee List screen
Extends: None
Uses: Login/Authentication; Get Accounts Menu; Payee Setup

UC08B - Payee Setup: unsuccessful add entry

Definition:
User chooses add Payee option
System responds with the Add Payee screen
User chooses an entry from the New Payee List
User submits data

System responds with the Add Payee screen showing an error message
Extends: None
Uses: Login/Authentication; Get Accounts Menu; Payee Setup

UC09 - Payee Setup: delete entry

Definition:
User selects the Payee to delete
User chooses delete Payee option
System responds with the Confirm Delete Payee screen
User submits the confirmation
System responds with the updated Payee List screen
Extends: None
Uses: Login/Authentication; Get Accounts Menu; Payee Setup

UC10 - Change passwords

Definition:
User selects the User option
System responds with the User screen
User selects the Change Passwords option
System responds with the Change Password screen
User selects which password to change
User enters the old, new and confirmed new passwords
User submits changes
System Responds
Extends: None
Uses: Login/Authentication

UC10A - Successful change passwords

Definition:
User selects the User option
System responds with the User screen
User selects the Change Passwords option
System responds with the Change Password screen
User selects which password to change
User enters the old, new and confirmed new passwords
User submits changes
System responds with the Accounts screen
Extends: Change passwords
Uses: Login/Authentication

UC10B - Unsuccessful change passwords

Definition:
User selects the User option
System responds with the User screen

User selects the Change Passwords option
System responds with the Change Password screen
User selects which password to change
User enters the old, new and confirmed new passwords
User submits changes
System responds with the Change Password screen showing an error message
Extends: Change passwords
Uses: Login/Authentication

UC11 - Logout

Definition:
User selects Logout option
System logs out user and responds with the Logout screen
Extends: None
Uses: Login/Authentication

Appendix B. Working with the HBA Implementation

This appendix lists the steps involved in deploying the HBA or working with the HBA code. This appendix assumes that you have the Zip file which is available from:

`ftp://www.redbooks.ibm.com/redbooks/SG245423/`

B.1 Deployment

Prerequisites:

WebSphere Application Server Version 2.0

A WebSphere supported Web server

Steps:

To deploy the HBA:

1. Unzip `site.zip` to the document root directory of your Web Server.
2. Unzip `site_code.zip` to a temporary directory:
 - Copy `bank.jar` to the `AppServer\classes` directory of WebSphere.
 - Copy the `itso` directory to the `AppServer\servlets` directory of WebSphere. Make sure you are copying both the `.class` and `.servlet` files.
3. Edit the `servlets.properties` file:
 - On NT: `AppServer\properties\server\servlet\servletservice` (or use the WebSphere Application Server Administration utility).
 - On AIX: `IBMWebAS/properties/server/servlet/servletservice`

Change the `servlets.startup=invoker` line to:

```
servlets.startup=invoker BankServlet.
```

Add the following lines below `# Servlets added by the user`:

```
servlet.BankServlet.code=itso.bank.servlet.BankServlet  
servlet.BankServlet.initArgs=implementation=itso.bank.baseimpl
```

(or use the WebSphere Application Server Administration utility).

4. Edit the `bootstrap.properties` file in `AppServer\properties` and add the `bank.jar` file to the `CLASSPATH`.
 - For an NT environment: You must use the 8 character naming convention for this, for example:

```
f:\WEBSPH-1\APPSER-1\classes\bank.jar;
```

(or use the WebSphere Application Server Administration utility).

- For an AIX environment, use colons to separate entries.
5. Follow Chapter 6, “Deploying the Home Banking Application” on page 169 for configuring SSL and the Web servers and restart WebSphere and the Web server.

Removing or Changing the pre-filled User ID and Password fields

Some implementations may only have one level of password authentication. In that case the login and transaction passwords are the same. If you don't want the user ID and password fields pre-filled or you are using a different implementation you can change them by:

1. Open Login\Login.jsp in the document root directory.
 - Search for deepblue
 - Change it to the user ID you want or delete it.
 - Search for ibmibm
 - Change it to the password you want or delete it.
2. Open Accounts\Pay_Bill\pay_bill.jsp
 - Search for ibmibm
 - Change it to the password you want or delete it.
3. Open Accounts\Transfer_Funds\transfer_funds.jsp
 - Search for ibmibm
 - Change it to the password you want or delete it.

Notes

If you are deploying on AIX, check the file permissions. The user ID which runs the Web and application server must be able to access the HBA files and directories.

Also on AIX, the Java processes started by WebSphere Application Server must be stopped manually when restarting the server. On NT, shut down the WebSphere Application Server from the Services dialog.

B.2 Development

Currently, the VisualAge for Java Enterprise and Professional Updates only work on Windows NT. You must be using VisualAge for Java on Windows NT in order to develop using the update. We used WebSphere Studio 3.0 Beta 2 to develop the HBA. Other versions may not be compatible.

Prerequisites:

VisualAge for Java Version 2.0

VisualAge for Java Rollup 2

VisualAge for Java Enterprise or Professional Update (only on Windows NT)

NetObjects Fusion Version 4.0

WebSphere Studio Version 3.0 Beta 2

Steps:

Site (HTML and JSP) Development: Open the HBA.war archive using **File**→**Open Archive** in WebSphere Studio.

Servlet and Java Code Development: Extract HBA.dat from java_src.zip and import the DAT file into VisualAge for Java.

To run the system in VisualAge for Java:

1. Unzip site.zip to the document root directory of your Web Server that you specified when adding the WebSphere Test Environment to VisualAge for Java or publish the site using WebSphere Studio.
2. Unzip servlet_config.zip into the WebSphere Test Environment\servlets directory.
3. Edit the servlets.properties file in ide\project_resources\IBM Websphere Test Environment\properties\server\servlet\servletservice:
 - Change the servlets.startup=invoker line to:
servlets.startup=invoker BankServlet
 - Add the following lines below # Servlets added by the user:
servlet.BankServlet.code=itso.bank.servlet.BankServlet
servlet.BankServlet.initArgs=implementation=itso.bank.baseimpl
4. Add the WebSphere Bank Application to the ClassPath of the ServletRunner.

To deploy a new development version:

1. From WebSphere Studio:

- Publish the project to the correct directories for your Web server and WebSphere Application Server.

2. From VisualAge:

- Export the following packages to bank.jar in the AppServer\classes directory of WebSphere:

- itso.bank.admin
- itso.bank.baseimpl
- itso.bank.common
- itso.bank.util
- itso.bank.viewobjects

- Export the itso.bank.servlets class files (not a JAR file) to the AppServer\servlets directory of WebSphere.

Follow the steps outlined in B.1, “Deployment” on page 191.

The system has a dependency on `c:\temp\bank.ser` in Windows NT and `/tmp/bank.ser` on AIX. If this file cannot be created, the serialization of the bank will fail.

Appendix C. Special Notices

This publication is intended to help Web application developers to develop e-business applications using IBM tools. The information in this publication is not intended as the specification of any programming interfaces that are provided by WebSphere Application Server, WebSphere Studio, NetObjects Fusion or VisualAge for Java. See the PUBLICATIONS section of the IBM Programming Announcement for WebSphere Application Server, WebSphere Studio and VisualAge for Java 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:

IBM ®	DB2
VisualAge	WebSphere
TeamConnection	AIX
SP	TXSeries
Net.Data	

The following terms are trademarks of other companies:

VeriSign is a trademark of VeriSign, Inc.

ColdFusion and Allaire are trademarks of Allaire, Inc.

NetObjects, Fusion and ScriptBuilder are trademarks of NetObjects Inc.

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, Visual Source Safe, ASP, Active Server Pages 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/or other countries licensed exclusively through X/Open Company Limited.

SET 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.

Appendix D. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

D.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 201.

- *Programming with VisualAge for Java Version 2.0*, SG24-5264
- *IBM WebSphere and VisualAge for Java Database Integration with DB2, Oracle, and SQL Server*, SG24-5471
- *Enterprise JavaBeans Development Using VisualAge for Java*, SG24-5429
- *VisualAge for Java Enterprise Version 2: Persistence Builder with GUIs, Servlets, and Java Server Pages*, SG24-5426
- *VisualAge for Java Enterprise Version 2: Data Access Beans - Servlets - CICS Connector*, SG24-5265
- *VisualAge for Java Enterprise Version 2 Team Support*, SG24-5245
- *Application Development with VisualAge for Java Enterprise*, SG24-5081
- *Internet Security in the Network Computing Framework*, SG24-5220
- *Java Network Security*, SG24-2109

D.2 Redbooks on CD-ROMs

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at <http://www.redbooks.ibm.com/> for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	Collection Kit Number
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
RS/6000 Redbooks Collection (BkMgr)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

D.3 Other Publications

These publications are also relevant as further information sources:

- Akerley, John, Nina Li and Antonello Parlavecchia. 1999. *Programming with VisualAge for Java Version 2.0*. Englewood Cliffs, NJ; Prentice Hall; ISBN 0-13-021298-9
- Asbury, Stephen and Scott R. Weiner. 1999. *Developing Java Enterprise Applications*. New York, NY; John Wiley; ISBN 0-471-32756-5
- Booch, Grady. 1994. *Object-Oriented Analysis and Design with Applications* (Addison-Wesley Object Technology Series), Reading, MA; Addison-Wesley Publishing Company; ISBN 0805353402
- Cheswick, William R. and Steven M. Bellovin. 1994. *Firewalls and Internet Security : Repelling the Wily Hacker*. Reading, MA; Addison-Wesley Publishing Company; ISBN 0201633574
- Flanagan, David. 1997. *Java in a Nutshell; A Desktop Quick Reference*, Sebastopol, CA; O'Reilly & Associates; ISBN 156592262X

- Fowler, Martin, Kendall Scott (Contributor) and Ivar Jacobson. 1997. *Uml Distilled ; Applying the Standard Object Modeling Language*. Reading, MA; Addison-Wesley Publishing Company; ISBN 0-201-32563-2
- Gamma Erich, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*, Reading, MA; Addison-Wesley Publishing Company; ISBN 0-201-63361-2
- Garfinkel, Simson and Gene Spafford. 1997. *Web Security and Commerce*. Sebastopol, CA; O'Reilly & Associates; ISBN 1-56592-269-7
- Grand, Mark. 1998. *Patterns in Java*. New York, NY; John Wiley; ISBN 0-471-25839-3
- Horstmann, Cay S. and Gary Cornell. 1997. *Core Java 1.1; Fundamentals*, Englewood Cliffs, NJ; Prentice Hall; ISBN 0137669577
- Horstmann, Cay S. and Gary Cornell. 1997. *Core Java 1.1; Advanced Features*, Englewood Cliffs, NJ; Prentice Hall; ISBN 0137669658
- Hunter Jason and William Crawford. 1998. *Java Servlet Programming*, Sebastopol, CA; O'Reilly & Associates; ISBN 1-56592-391-X
- Jacobson, Ivar. 1992. *Object-Oriented Software Engineering ; A Use Case Driven Approach*, Reading, MA; Addison-Wesley Publishing Company; ISBN 0201544350
- Moss, Karl. 1998. *Java Servlets*, New York, NY; Computing McGraw-Hill, ISBN 0-07-913779-2
- Naughton, Patrick and Herbert Schildt. 1998. *Java The Complete Reference*. New York, NY; Osborne McGraw-Hill, ISBN 0-07-882231-9
- Nilsson Dale and Peter Jakab. 1999. *Developing JavaBeans Using VisualAge for Java*. New York, NY; John Wiley; ISBN 0-471-29788-7
- Rumbaugh, James et al. 1991. *Object-Oriented Modeling and Design*, Englewood Cliffs, NJ; Prentice Hall; ISBN 0136298419
- Taylor, Chris and Tim Kimmert. 1998. *Core Java Web Server*. Englewood Cliffs, NJ; Prentice Hall; ISBN 0-13-080559-9
- Bayeh, Elias. 1998. *The WebSphere Application Server architecture and programming model*. IBM Systems Journal Vol 37, No. 4, 1998
www.research.ibm.com/journal

D.4 Product Documentation

The following product documentation was helpful during the project:

VisualAge for Java

- Online product documentation
- PDF documentation:
 - JSP/Servlet Development Environment
 - Team Programming
- Web sites:
 - www.software.ibm.com/ad/vajava
 - www.software.ibm.com/vadd

WebSphere Application Server

- Online product documentation
- Bayeh, Elias. 1998. *The WebSphere Application Server architecture and programming model*. IBM Systems Journal Vol 37, No. 4, 1998
www.research.ibm.com/journal
- Web sites:
 - www.software.ibm.com/webservers

WebSphere Studio

- Web sites:
 - www.software.ibm.com/webservers

NetObjects Fusion

- Web sites:
 - www.netobjects.com

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** <http://www.redbooks.ibm.com/>

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 redbooks fax order form to:

	e-mail address
In United States	usib6fpl@ibmmail.com
Outside North America	Contact information is in the "How to Order" section at this site: http://www.elink.ibm.com/pbl/pbl/

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the "How to Order" section at this site: http://www.elink.ibm.com/pbl/pbl/

- **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.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 employees may register for information on workshops, residencies, and redbooks by accessing the IBM Intranet Web site at <http://w3.itso.ibm.com/> and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at <http://w3.ibm.com/> for redbook, residency, and workshop announcements.

Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

abstract class. A class that provides common behavior across a set of subclasses but is not itself designed to have instances. An abstract class represents a concept; classes derived from it represent implementations of the concept. See also *base class*.

access modifier. A keyword that controls access to a class, method, or attribute. The access modifiers in Java are public, private, protected, and package, the default.

accessor methods. Methods that an object provides to define the interface to its instance variables. The accessor method to return the value of an instance variable is called a *get* method or *getter* method, and the mutator method to assign a value to an instance variable is called a *set* method or *setter* method.

applet. A Java program designed to run within a Web browser. Contrast with application.

application. In Java programming, a self-contained, stand-alone Java program that includes a *main()* method. Contrast with applet.

application programming interface (API). A software interface that enables applications to communicate with each other. An API is the set of programming language constructs or statements that can be coded in an application program to obtain the specific functions and services provided by an underlying operating system or service program.

argument. A data element, or value, included as a parameter in a method call. Arguments provide additional information that the called method can use to perform the requested operation.

associated. In WebSphere Studio, a file that is marked as belonging to a site. Associated files display as non-dimmed icons in the File View.

attribute. A specification of an element of a class. For example, a customer bean could have a name attribute and an address attribute.

B

base class. A class from which other classes or beans are derived. A base class may itself be derived from another base class. See also *abstract class*.

bean. A definition or instance of a JavaBeans component. See also *JavaBeans*.

BeanInfo. (1) A companion class for a bean that defines a set of methods that can be accessed to retrieve information on the bean's properties, events, and methods. (2) In the VisualAge for Java IDE, a page in the Class Browser that provides bean information.

beans palette. In the Visual Composition Editor, a pane that contains beans that you can select and manipulate to create programs. You can add your own categories and beans to the beans palette.

break point. A point in a computer program where the execution will be halted.

browser. (1) In VisualAge for Java, a window that provides information about program elements. There are browsers for projects, packages, classes, methods, and interfaces. (2) An Internet-based tool that lets user browse Web sites.

C

category. In the Visual Composition Editor, a selectable grouping of beans on the palette. Selecting a category displays the beans belonging to that category. See also *beans palette*.

child. In WebSphere Studio, a file that is referenced by another file.

class. A template that defines properties, operations, and behavior for all instances of that template.

class hierarchy. The relationships among classes that share a single inheritance. All Java classes inherit from the Object class.

class library. A collection of classes.

class method. See *method*.

CLASSPATH. (1) In VisualAge for Java the lists of pathnames which will be searched for dynamically loaded classes, BeanInfo information and external source for debugging. (2) In your deployment environment, the environment variable that specifies the directories in which to look for class and resource files.

client/server. The model of interaction in distributed data processing where a program at one location sends a request to a program at another location and awaits a response. The requesting program is called a *client*, and the answering program is called a *server*.

Class Browser. In the VisualAge for Java IDE, a tool used to browse the classes loaded in the workspace.

component model. An architecture and an API that allows developers to define reusable segments of code that can be combined to create a program. VisualAge for Java uses the JavaBeans component model.

composite bean. A bean that is composed of other beans. A composite bean can contain visual beans, nonvisual beans, or both. See also *bean*, *nonvisual bean*, and *visual bean*.

concrete class. A non-abstract subclass of an abstract class that is a specialization of the abstract class.

connection. In the Visual Composition Editor, a visual link between two components that represents the relationship between the components. Each connection has a source, a target, and other prop-

erties. See also *event-to-method connection*, *parameter connections*, and *property-to-property connection*.

console. In VisualAge for Java, the window that acts as the standard input (System.in) and standard output (System.out) device for programs running in the VisualAge for Java IDE.

construction from parts. A software development technology in which applications are assembled from existing and reusable software components, known as *parts*. In VisualAge for Java, parts are called *beans*.

constructor. A special class method that has the same name as the class and is used to construct and possibly initialize objects of its class type.

container. A component that can hold other components. In Java, examples of containers include Applets, Frames, and Dialogs. In the Visual Composition Editor, containers can be graphically represented and generated.

current edition. The edition of a program element that is currently in the workspace. See also *open edition*.

custom link. In WebSphere Studio, a relationship between files that you identify and WebSphere Studio does not automatically recognize.

D

DNS. See "domain name server."

demarshal. To deconstruct an object so that it can be written as a stream of bytes. Synonym for *flatten* and *serialize*.

deserialize. To construct an object from a de-marshaled state. Synonym for *marshal* and *resurrect*.

domain. A domain name server (DNS) or Internet protocol (IP) address, for example, software.ibm.com or 123.45.67.8.

domain name server. A system for translating domain names such as www.software.ibm.com into numeric Internet protocol addresses such as 123.45.67.8.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Compare with *single-byte character set*.

dynamic information. Information that is created at the time the user requests it. Dynamic information changes over time so that each time users view it, they see different content.

E

edition. A specific “cut” of a program element. VisualAge for Java supports multiple editions of program elements. See also *current edition*, *open edition*, and *versioned edition*.

encapsulation. The hiding of a software object’s internal representation. The object provides an interface that queries and manipulates the data without exposing its underlying structure.

event. An action by a user program, or a specification of a notification that may trigger specific behavior. In JDK 1.1, events notify the relevant listener classes to take appropriate actions.

event-to-method connection. A connection from an event generated by a bean to a method of a bean. When the connected event occurs, the method is executed. See also *connection*.

F

FTP. See “file transfer protocol.”

factory. A nonvisual bean capable of dynamically creating new instances of a specified bean.

feature. (1) A component of VisualAge for Java that is installed separately using the QuickStart. (2) A method, field, or event that is available from a bean’s interface and to which other beans can connect.

field. See attribute

file transfer protocol. An international standard for transferring files from one computer to another across a network.

File View. In WebSphere Studio, graphical representation of all the files in your site arranged in folders.

flatten. Synonymous with *demarshal*.

Folder. In WebSphere Studio, a group of related files.

free-form surface. The open area of the Visual Composition Editor where you can work with visual and nonvisual beans. You add, remove, and connect beans on the free-form surface.

G

generated link. In WebSphere Studio, a link that is created by WebSphere Studio based on the parameters of a code file.

graphical user interface (GUI). A type of interface that enables users to communicate with a program by manipulating graphical features, rather than by entering commands. Typically, a GUI includes a combination of graphics, pointing devices, menu bars and other menus, overlapping windows, and icons.

H

HTML. See “hypertext markup language.”

HTTP. See “hypertext transfer protocol.”

home page. See “start page.”

hyperlinks. Areas on a Web page that, when clicked, connect you to other areas on the page or other Web pages.

Hypertext Markup Language (HTML). The basic language that is used to build hypertext documents on the World Wide Web. It is used in basic, plain ASCII-text documents, but when those documents are interpreted, or *rendered*, by a Web browser such as Netscape, the document can display for-

matted text, color, a variety of fonts, graphical images, special effects, hypertext jumps to other Internet locations, and information forms.

Hypertext Transfer Protocol (HTTP). The protocol for moving hypertext files across the Internet. Requires an HTTP client program on one end, and an HTTP server program on the other end. HTTP is the most important protocol used in the World Wide Web.

I

IDE. See Integrated Development Environment.

IP. See “Internet protocol address.”

IP number. An Internet address that is a unique number consisting of four parts separated by dots, sometimes called a *dotted quad* (for example: 198.204.112.1). Every Internet computer has an IP number, and most computers also have one or more domain names that are mappings for the dotted quad.

Import Wizard. A WebSphere Studio feature that copies an existing Web site into the WebSphere Studio environment.

inheritance. (1) A mechanism by which an object class can use the attributes, relationships, and methods defined in classes related to it (its base classes). (2) An object-oriented programming technique that allows you to use existing classes as bases for creating other classes.

inside link. In WebSphere Studio, a file within a WebSphere Studio site that links to other files in the site.

instance. Synonym for *object*, a particular instantiation of a data type.

integrated development environment (IDE). In VisualAge for Java, the set of windows that provide the user with access to development tools. The primary windows are Workbench, Class Browser, Log, Console, Debugger, and Repository Explorer.

interchange file. A file that you can export from VisualAge for Java that contains information about selected projects or packages. This file can then be imported into any VisualAge for Java session.

interface. A named set of method declarations that is implemented by a class. The Interface page in the Workbench lists all interfaces in the workspace.

Internet. The collection of interconnected networks that use TCP/IP and evolved from the ARPANET of the late 1960s and early 1970s.

Internet Protocol (IP). The protocol that provides basic Internet functions.

Internet protocol address. A numeric address that uniquely identifies every computer connected to a network. For example, 123.45.67.8

intranet. A private *network*, inside a company or organization, that uses the same kinds of software that you would find on the public Internet. Many of the tools used on the Internet are being used in private networks; for example, many companies have Web servers that are available only to employees.

J

JDBC. In JDK 1.1, the specification that defines an API that enables programs to access databases that comply with this standard.

Java. A programming language invented by Sun Microsystems that is specifically designed for writing programs that can be safely downloaded to your computer through the Internet and immediately run without fear of viruses or other harm to your computer or files.

Java archive (JAR). A platform-independent file format that groups many files into one. JAR files are used for compression, reduced download time, and security.

JavaBeans. The specification that defines the platform-neutral component model used to represent parts. Instances of JavaBeans (often called beans) may have methods, properties, and events.

K

keyword. A predefined word reserved for Java, for example, *return*, that may not be used as an identifier.

L

listener. In JDK 1.1, a class that receives and handles events.

local area network (LAN). A computer network located on a user's establishment within a limited geographical area. A LAN typically consists of one or more server machines providing services to a number of client workstations.

log. In VisualAge for Java, the window that displays messages and warnings during development.

M

MIME type(Multi-purpose Internet Mail Extensions). An international standard for categorizing types of Web files such as text and images.

MVC. See Model View Controller.

marshal. Synonymous with *deserialize*.

message. A communication from one object to another that requests the receiving object to execute a method. A method call consists of a method name that indicates the requested method and the arguments to be used in executing the method. The method call always returns some object to the requesting object as the result of performing the method. Synonym for *method call*.

method. A fragment of Java code within a class that can be invoked and passed a set of parameters to perform a specific task.

method call. Synonymous with message.

model. A nonvisual bean that represents the state and behavior of an object, such as a customer or an account. Contrast with *view*.

Model View Controller. An application architecture which separates the components of the application: the model represents the business logic or data; the view represents the user interface and the controller manages user input, or, in some cases the application flow.

mutator methods. Methods that an object provides to define the interface to its instance variables. The accessor method to return the value of an instance variable is called a *get* method or *getter* method, and the mutator method to assign a value to an instance variable is called a *set* method or *setter* method.

N

named package. In the VisualAge for Java IDE, a package that has been explicitly named and created.

nesting. In WebSphere Studio, the number of folder levels beneath other folders. One level of folders gives you a "nesting" of one. If that folder contains other folders, you have a nesting of two and so on.

nonvisual bean. In the Visual Composition Editor, a bean that has no visual representation at run time. A nonvisual bean typically represents some real-world object that exists in the business environment. Compare with *model*. Contrast with *view* and *visual bean*.

O

ODBC driver. An ODBC driver is a dynamic link library that implements ODBC function calls and interacts with a data source.

object. (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) A collection of data and methods that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and methods. Each object in the class is said to be an instance of the class. (3) An instance of an object class consisting of attributes, a data structure, and operational methods. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and methods as other instances of the object class, although it has unique values assigned to its attributes.

object class. A template for defining the attributes and methods of an object. An object class can contain other object classes. An individual representation of an object class is called an *object*.

object-oriented programming (OOP). A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on those data objects that constitute the problem and how they are manipulated, not on how something is accomplished.

Open Database Connectivity (ODBC). A Microsoft-developed C database API that allows access to database management systems calling callable SQL, which does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that allows users to add modules (database drivers) that link the application to their choice of database management systems at run time. Applications no longer need to be directly linked to the modules of all the database management systems that are supported.

open edition. An edition of a program element that can still be modified; that is, the edition has not been versioned. An open edition may reside in the workspace as well as in the repository.

operation. A method or service that can be requested of an object.

outside link. In WebSphere Studio, a link to a file that is located outside the current Web site.

P

package. A program element that contains related classes and interfaces.

palette. See *beans palette*.

parameter connection. A connection that satisfies a parameter of an action or method by supplying either a property's value or the return value of an action, method, or script. The parameter is always the source of the connection. See also *connection*.

parent. In WebSphere Studio, a file that contains a reference to another file.

parent class. The class from which another bean or class inherits data, methods, or both.

part. An existing, reusable software component. In VisualAge for Java, all parts created with the Visual Composition Editor conform to the JavaBeans component model and are referred to as beans. See also *nonvisual bean* and *visual bean*.

primitive bean. A basic building block of other beans. A primitive bean can be relatively complex in terms of the function it provides.

private. In Java, an access modifier associated with a class member. It allows only the class itself to access the member.

process. A collection of code, data, and other system resources, including at least one thread of execution, that performs a data processing task.

program. In VisualAge for Java, a term that refers to both Java applets and applications.

program element. In VisualAge for Java, a term referring to any of the entities under source control. Program elements are projects, packages, classes, interfaces, or methods.

project. In VisualAge for Java, the topmost kind of program element. A project contains Java packages.

promotion. Within a JavaBean, to make features of a contained bean available to be used for making connections. For example, a bean consisting of three push buttons on a panel. If this bean is placed in a frame, the features of the push buttons would have to be promoted to make them available from within the frame.

property. An initial setting or characteristic of a bean; for example, a name, font, text, or positional characteristic.

property sheet. In the Visual Composition Editor, a set of name-value pairs that specify the initial appearance and other bean characteristics.

property-to-property connection. A connection from a property of one bean to a property of another bean. See also *connection*.

protected. In Java, an access modifier associated with a class member. It allows the class itself, subclasses, and all classes in the same package to access the member.

protocol. (1) The set of all messages to which an object will respond. (2) Specification of the structure and meaning (the semantics) of messages that are exchanged between a client and a server. (3) Computer rules that provide uniform specifications so that computer hardware and operating systems can communicate.

prototype. A method declaration or definition that includes the name of the method, the return type and the types of its arguments. Contrast with *signature*.

publishing. In WebSphere Studio, the process of copying your site's files to Web servers.

Publishing View . In WebSphere Studio, a graphical representation of the stages (for example Test or Production) where you define the layout of your Web servers and identify the files you want in your Web site.

R

Relations View. In WebSphere Studio, a graphical representation of each file in your site and the links between those files.

Remote Method Invocation (RMI). In JDK 1.1, the API that enables you to write distributed Java programs, allowing methods of remote Java objects to be accessed from other Java virtual machines.

repository. In VisualAge for Java, the storage area, separate from the workspace, that contains all editions (both open and versioned) of all program elements that have ever been in the workspace, including the current editions that are in the workspace. You can add editions of program elements to the workspace from the repository.

Repository Explorer. In VisualAge for Java, the window from which you can view and compare editions of program elements that are in the repository.

resource file. A file that is referred to from your Java program. Examples include graphics and audio files.

Resources folder. In WebSphere Studio, the folder that physically holds a site's folders and files.

resurrect. Synonymous with *deserialize*.

RMI compiler. The compiler that generates stub and skeleton files that facilitate RMI communication. This compiler can be automatically invoked from the Tools menu item.

RMI registry. A server program that allows remote clients to get a reference to a server bean.

S

Scrapbook. In VisualAge for Java, the window from which you can write and test fragments of code, without having to define an encompassing class or method.

serialize. Synonymous with *demarshal*.

signature. The part of a method declaration consisting of the name of the method and the number and types of its arguments. Contrast with *prototype*.

single-byte character set. A set of characters in which each character is represented by a 1- byte code.

SmartGuide. In IBM software products, an interface that guides you through performing common tasks.

source link. In WebSphere Studio, a link you create to identify the source file of a publishable file.

Start page. The first page a user sees when browsing a Web site, also known as the "home page."

static information. Web files that do not change on every access.

sticky. In the Visual Composition Editor, the mode that enables an application developer to add multiple beans of the same class (for example, three push buttons) without going back and forth between the beans palette and the free-form surface.

superclass. See *abstract class* and *base class*.

T

tear-off property. A property that a developer has exposed as a variable to work with as though it were a stand-alone bean.

thread. A unit of execution within a process.

type. In VisualAge for Java, a generic term for a class or interface.

U

URL . See “uniform resource locator.”

Unicode. A character coding system designed to support the interchange, processing, and display of the written texts of the diverse languages of the modern world. Unicode characters are typically encoded using 16-bit integral unsigned numbers.

uniform resource locator (URL). A standard identifier for a resource on the World Wide Web, used by Web browsers to initiate a connection. The URL includes the communications protocol to use, the name of the server, and path information identifying the objects to be retrieved on the server.

user interface (UI). (1) The hardware, or software, or both that enables a user to interact with a computer. (2) The term *user interface* typically refers to the visual presentation and its underlying software with which a user interacts.

V

variable. (1) A storage place within an object for a data feature. The data feature is an object, such as number or date, stored as an attribute of the containing object. (2) A bean that receives an identity at run time. A variable by itself contains no data or program logic; it must be connected such that it receives run-time identity from a bean elsewhere in the application.

versioned edition. An edition that has been versioned and can no longer be modified.

versioning. The act of making an open edition a versioned edition; that is, making the edition read-only.

view. (1) A visual bean, such as a window, push button, or entry field. (2) A visual representation that can display and change the underlying model objects of an application. Views are both the end result of developing an application and the basic unit of composition of user interfaces. Compare with *visual bean*. Contrast with *model*.

visual bean. In the Visual Composition Editor, a bean that is visible to the end user in the graphical user interface. Compare with *view*. Contrast with *nonvisual bean*.

visual programming tool. A tool that provides a means for specifying programs graphically. Application programmers write applications by manipulating graphical representations of components.

Visual Composition Editor. In VisualAge for Java, the tool where you can create graphical user interfaces from prefabricated beans and define relationships (connections) between both visual and nonvisual beans. The Visual Composition Editor is a page in the class browser.

W

Web application. A software system that is designed to automate a business process and is delivered on intranets or the Internet.

Workbench. In VisualAge for Java, the main window from which you can manage the workspace, create and modify code, and open browsers and other tools.

workspace. The work area that contains all the code you are currently working on (that is, current editions). The workspace also contains the standard Java class libraries and other class libraries.

List of Abbreviations

ANSI	American National Standards Institute	MVS	Multiple Virtual Storage
API	application programming interface	NLS	National Language Support
ATM	automated teller machine	NT	new technology
AWT	Abstract Windowing Toolkit	ODBC	Open Database Connectivity
CAE	Client Access Enabler	OMG	Object Management Group
URL	uniform resource locator	OMT	object modeling technique
CLI	call level interface	OO	object-oriented
DB2	DATABASE 2	OOA	object-oriented analysis
DBCS	double-byte character set	OOD	object-oriented design
DBMS	database management system	ORB	Object Request Broker
DLL	dynamic link library	OS/2	Operating System/2
DNS	domain name server	OTS	object transaction service
DRDA	Distributed Relational Database Architecture	PIN	personal identification number
ECD	edit-compile-debug	RAD	rapid application development
ECI	external call interface	RDBMS	relational database management system
FTP	File Transfer Protocol	RMI	Remote Method Invocation
GUI	graphical user interface	SBCS	single-byte character set
HTML	Hypertext Markup Language	SDK	Software Developer's Kit
HTTP	Hypertext Transfer Protocol	SQL	structured query language
IBM	International Business Machines Corporation	TCP/IP	Transmission Control Protocol/Internet Protocol
IDE	integrated development environment	TP	transaction processing
IDL	interface definition language	UOW	unit of work
IIOP	Internet inter-ORB protocol	URL	uniform resource locator
IMS	Information Management System	WWW	World Wide Web
IOR	interoperable object reference		
ITSO	International Technical Support Organization		
JAR	Java archive		
JDK	Java Developer's Kit		
JNI	Java Native Interface		
JVM	Java Virtual Machine		
LAN	local area network		
MOFW	managed object framework		

Index

A

Account Balance JSP 41, 130
Account History JSP 40, 130
Account Information JSP 40, 41, 129
Account Information subsystem 122
Accounts JSP 39, 41, 44, 45, 54, 116, 122, 133
AccountServlet class 127
AccountServlet object 40, 41
AccountViewList bean 40
Active Server Pages 4
adaptor pattern 106
Add Payee JSP 43, 158
Already Logged In HTML page 122
Application Framework for e-business 1
Application Manager 109
ArrayOutOfBoundsException class 15
authentication 16, 19

B

Bank interface 100
BankAccount interface 100
BankAccountImpl class 107
BankAccountView class 107
BankCollection class 101
BankHome class 101
BankServlet 109
BankServlet class 111, 112, 183
BankSystem interface 100
BEAN tag 12
Bill Paid JSP 42, 132, 143
Bill Payment Subsystem 131
BillPaymentServlet class 134, 141

C

CacheControl class 166
caching 21
callPage method 15, 139, 147
Certificate Authority 169
Change Password JSP 45, 166
ChangePasswordServlet class 162
ChangePasswordServlet object 45
CheckingAccount interface 100
client side digital certificate 21
ColdFusion 4
com.ibm.servlet package

PageListServlet class 105
SERunner class 74
com.sun.server.http package
 HttpServletRequest class 15
 HttpServletResponse class 15
command pattern 33
Common Gateway Interface 4
confidentiality 19, 20
cookies 17, 97
CORBA 2
Customer interface 100
CustomerView 110
CustomerView bean 39

D

Data Encryption Standard 20
DeletePayee JSP 159
DES 20
digital certificates 21, 169
Document Root directory 65
DocumentRoot directive 177
domain firewall 32, 99
dynamic pages 4

E

EBAF 2
encryption 20
Enterprise JavaBeans 2

F

Formatter class 167
Funds Transferred JSP 44, 145, 149

H

hidden form fields 17
Home Banking Application
 Account Balance subsystem 40
 Account History subsystem 40
 Account Information subsystem 38
 Add Payee subsystem 42
 analysis object model 26
 Application Manager 38
 business model access 32
 business model implementation 103

- client-server interaction 32, 34
- Customer object 39
- Delete Payee subsystem 43
- error handling 37
- implementation 103
- JavaServer Page design 32, 36
- Login subsystem 39
- Pay Bill subsystem 41
- Payee Setup subsystem 42
- prototype 26
- requirements 23
- security model 28
- subsystem design 28
- Transfer Funds subsystem 44
- use cases 24
- User subsystem 44
- HTML
 - forms 6
 - hidden form fields 17
 - pages
 - Already Logged In 122
 - Unsuccessful Login 39, 116, 122
- HTTP 18
- HTTP header 22
- HTTPServer class 72
- HTTPSession 109

I

- IBM HTTP Server 169
- IKEYMAN 176
- INSERT tag 14
- integrity 20
- ISAPI 4
- ITSO Bank Error JSP 122
- itso.bank.baseimpl package 183
- itso.bank.common package
 - Bank interface 100
 - BankAccount interface 100
 - BankCollection class 101
 - BankHome class 101
 - BankSystem interface 100
 - BankTransactionException class 101
 - CheckingAccount interface 100
 - Customer interface 100
 - InvalidPasswordException class 101
 - InvalidPinException class 101
 - ITSOBankCommunicationException class 101
 - ITSOBankException class 101
- NotImplementedException class 101
- PayeeAccount interface 100
- SavingsAccount interface 100
- TransactionRecord interface 100
- UnauthorizedException class 101

itso.bank.servlet

- BankServlet 109, 111

itso.bank.servlet package

- AccountServlet class 127
- BankServlet class 111, 112, 183
- BillPaymentServlet class 134, 141
- ChangePasswordServlet class 162
- LoginServlet class 119
- MoneyTransferServlet class 133, 135
- PayeeServlet class 154
- TransferFundsServlet class 148

itso.bank.util package

- CacheControl class 166
- Formatter class 167
- XMLConfigUtil class 168

itso.bank.viewobjects

- CustomerView 110

itso.bank.viewobjects package

- AccountView class 130
- BankAccountView class 107, 108
- BankAccountViewList class 110, 128
- CustomerView class 108, 110, 118, 121, 128
- TransactionRecordView class 107

ITSOBankCommunicationException class 101

ITSOBankException class 101

J

- Java Servlet Development Kit 7, 72
- JavaServer Pages
 - Account Balance 41, 130
 - Account History 40
 - Account Information 40, 41
 - Accounts 39, 41, 44, 54, 116, 122, 133
 - Add Payee 43, 158
 - API 15
 - BEAN tag 12
 - Bill Paid 42, 143
 - Change Password 45, 166
 - declarations 10
 - Delete Payee 159
 - directives 10
 - elements 10
 - expressions 11

- Funds Transferred 44, 145, 149
- in WebSphere 87
- INSERT tag 14
- introduction 5, 10
- ITSO Bank Error 122
- Login 122
- Logout 38, 114
- Not Logged In 111
- Pay Bill 41, 42, 143
- Payee Setup 42, 43
- PayeeSetup 158
- REPEAT tag 15
- scriptlets 11
- specification 10, 37, 87
- tags 11, 36
- Transfer Funds 149
- User 166
- javax.servlet package
 - GenericServlet class 9
 - introduction 7
 - ServletConfig class 9
 - ServletContext class 9
 - ServletRequest class 9
 - ServletResponse class 9
- javax.servlet.http package
 - HttpServletRequest class 9
 - HttpServletResponse class 9
 - HttpSession class 15
 - introduction 7
- JSDK 7
- JSP Execution Monitor
 - introduction 77
 - using 80
- JSP Page Compile Generated Code project 76

L

- Login JSP 122
- Login Subsystem 115
- LoginServlet class 119
- LoginServlet object 39
- Logout JSP 38, 114

M

- MAC 20
- message authentication code 20
- META tags 22
- Model/View/Controller architecture 5, 34
- multitier architecture 3

N

- Net.Data 4
- NetObjects Fusion
 - custom templates 51
 - MasterBorders 51
 - prototyping with 26, 48, 50
 - Publishing Wizard 54
- NetObjects ScriptBuilder 57
- Netscape Enterprise Server 169
- non-repudiation 19, 20
- Not Logged In JSP 111
- NotImplementedException class 101
- NSAPI 4

P

- PageListServlet class 105
- Pay Bill JSP 41, 42, 131, 143
- Payee Setup JSP 42, 43, 158
- Payee Subsystem 149
- PayeeAccount interface 100
- PayeeServlet object 43
- private key 21
- proxy servers 21, 22
- public key 21
- putValue method 15

R

- Rational Rose 50
- REPEAT tag 15

S

- SavingsAccount interface 100
- scripting languages 4
- Secure Sockets Layer 20, 169
- security
 - introduction 19
- sendRedirect method 110, 139
- serialization 103
- SERunner class 74
- server plug-in technologies 4
- server side digital certificate 21
- SERVLET tag 38, 68, 111
- Servlets
 - XML Configuration 105
- servlets
 - accessing 6
 - debugging 72

- destroy method 9
- doGet method 9
- doPost method 9
- getServletConfig method 9
- getServletInfo method 9
- init method 8
- introduction 4
- life cycle 7
- service method 8
- Servlet API 6, 7, 18
- SERVLET tag 6
- ServletConfig class 9
- ServletContext class 9
- session management 18
- session management 109
- setAttribute method 15
- Software Configuration Management 48
- SSL 20
- sun.servlet.http package
 - HttpRequest class 15
 - HTTPServer class 72, 73

T

- TCP/IP 20
- TransactionRecord interface 100
- Transfer Funds JSP 44, 149
- Transfer Funds Subsystem 144
- TransferFundsServlet object 44

U

- Unsuccessful Login HTML page 39, 116, 122
- URL rewriting 18, 97
- User JSP 45, 166
- User subsystem 159

V

- VeriSign 169
- VisualAge Developers Domain 73
- VisualAge for Java
 - introduction 71
 - source control 48

W

- Web application 3
- Web programming model 3
- WebSphere Application Server
 - connection management 96

- debugging 92
- introduction 85
- Server Manager 87
- sessions 97
- WebSphere Studio
 - Applet Designer 57
 - creating links 68
 - File View 57
 - import 48, 62
 - introduction 57
 - link types 58
 - Page Designer 60
 - Publishing View 58
 - Relations View 57
 - Report Generation 58
 - source control 48
 - Views 57
 - Web Development Workbench 57
- WebSphere Test Environment
 - errors 75
 - generated code 80
 - initialization 75
 - introduction 73
 - usage in HBA development 49

X

- XML 37
- XMLConfigUtil class 168

ITSO Redbook Evaluation

Developing an e-business Application for the IBM WebSphere Application Server
SG24-5423-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at <http://www.redbooks.ibm.com/>
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?

Customer **Business Partner** **Solution Developer** **IBM employee**
 None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

What other redbooks would you like to see published?

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

SG24-5423-00
Printed in the U.S.A.

Developing an e-business Application for the IBM WebSphere Application Server

SG24-5423-00

