WebSphere

Application Server Guide

Version 1.0

SC34-4767-00

WebSphere

Application Server Guide

Version 1.0

SC34-4767-00

Note

Before using this information and the product it supports, be sure to read the general information under “Appendix A. Notices” on page 101.

First Edition (June 1998)

This edition applies to Version 1.0 of IBM WebSphere Application Server and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

Contents

Welcome!

This book contains information you will need to:

- Install the IBM WebSphere™ Application Server
- Use the Application Server Manager to configure and monitor servlets on your Application Server
- Develop servlets and Web applications that use the Application Server class extensions and additions to the Java Servlet API
- Run the sample applications and servlets shipped with the Application Server

The most current version of this book is available in HTML and PDF formats on the library page of the Application Server Web site:

`http://www.software.ibm.com/webservers/`

Chapter 1. Getting started

IBM WebSphere Application Server lets you achieve your ″write once, use everywhere″ goal for servlet development. The product consists of a Java-based servlet engine that is independent of both your Web server and its underlying operating system. The Application Server offers a choice of server plug-ins that are compatible with the most popular server application programming interfaces (APIs). The supported Web servers are:

- Domino Go Webserver
- Netscape Enterprise Server
- Netscape FastTrack Server
- Microsoft** Internet Information Server
- Apache Server

In addition to the servlet engine and plug-ins, Application Server provides:

- Implementation of the JavaSoft Java Servlet API, plus extensions of and additions to the API.
- Sample applications that demonstrate the basic classes and the extensions.
- The Application Server Manager, a graphical interface that makes it easy to set options for loading local and remote servlets, set initialization parameters, specify servlet aliases, create servlet chains and filters, monitor resources used by the Application Server, monitor loaded servlets and active servlet sessions, log servlet messages, and perform other servlet management tasks.
- Connection management feature that caches and reuses connections to your JDBC (Java Database Connectivity)-compliant databases. When a servlet needs a database connection, it can go to the pool of available connections. This eliminates the overhead required to open a new connection each time.
- Additional Java classes, coded to the JavaBeans specification, that allow programmers to access JDBC-compliant databases. These data access Beans provide enhanced function while hiding the complexity of dealing with relational databases. They can be used in a visual manner in an integrated development environment (IDE).
- Support for a new technology for dynamic page content called JavaServer Pages (JSP). JSP files can include any combination of inline Java, <SERVLET> tags, NCSA tags, and JavaBeans.
- CORBA Support, an object request broker (ORB) and a set of services that are compliant with the Common Object Request Broker Architecture (CORBA).

Software requirements

Operating system

The Application Server supports the following operating systems:

- Microsoft NT** Version 4.0
- AIX Version 4.1.5 or higher
- Sun** Solaris Version 2.5.1 SPARC with the Native Threads Package

Note:

1. On Version 2.5.1, the Native Threads support requires two patches which resolve thread synchronization problems. The patches are 103566-08 and 103640-08.
2. If you are installing on Netscape Enterprise Version 3.x or Domino Go Webserver Version 4.6.x and have the SunOS patch 104283-03 installed, do *one* of the following:
   - Uninstall patch 104283-03
   - Install the SunOS kernel update patch 104283-04, available soon from Sun
   - Run the Web server as root and set the server.user property in /opt/*applicationserver_root*/properties/server/servlet/server.properties to root

- Sun Solaris Version 2.6 SPARC with the Native Threads Package

  Note: If you are installing on Netscape Enterprise Version 3.x or Domino Go Webserver Version 4.6.x, do *one* of the following:
  - Install the SunOS kernel update patch 105181-05, available from Sun
  - Run the Web server as root and set the server.user property in /opt/*applicationserver_root*/properties/server/servlet/server.properties to root

Server

The Application Server requires one of the following servers:

- Lotus Domino Go Webserver Version 4.6.1 or 4.6.2 for Windows NT, Sun Solaris, and AIX
- Microsoft Internet Information Server 2.x, 3.x, or 4.0
- Netscape Enterprise Version 2.01 or higher for Windows NT, Sun Solaris, and AIX

  Note: The Application Server runs best on Version 3.51; we recommend you use this version.
- Netscape FastTrack Server Version 2.01 or higher for Windows NT, Sun Solaris, and AIX
- Apache Version 1.2.x for Sun Solaris and AIX

Java Development Kit

The following levels of JavaSoft's Java Development Kit (JDK) are compatible:

- Windows NT 4.0: JDK 1.1.4 or 1.1.6

  The JDK 1.1.6 is shipped with the Application Server.
- AIX 4.1.5 and higher: JDK 1.1.4

  The JDK 1.1.4 is shipped with the Application Server.
- Sun Solaris 2.5.1 and higher: JDK 1.1.4 or 1.1.6 with the Native Threads Package

  The JDK 1.1.6 is shipped with the Application Server.

JDK 1.1.5, other than the version available from the SunSoft Web site, has a memory leak problem and is not recommended for any platform.

Java Servlet Development Kit

The Application Server includes the Java Servlet Development Kit (JSDK) Version 1.1 .class files, which you will need if you want to develop servlets. To obtain the Javadoc for those .class files, refer to: http://jserv.javasoft.com/products/java-server/documentation/webserver1.1/apidoc/packages.html

Browser (for the Application Server Manager)

The Application Server Manager is the user interface for managing servlets. To run the Manager, you need an appletviewer or a browser that supports JDK 1.1, for example:

- Netscape Communicator 4.03 or 4.04 with the JDK 1.1 patch, or 4.05 (available at http://developer.netscape.com/software/jdk/download.html)
- Microsoft Internet Explorer 4
- Sun HotJava 1.1

Note: Some older browsers do not properly handle text that was enabled for National Language Support. If you are seeing extraneous characters on the user interface, such as 'sEnable' instead of 'Enable' on the Session State panel, you can correct it by upgrading your browser.

Installing

To install the Application Server:

1. Insert the installation CD into your CD-ROM drive.
2. Go to the subdirectory for your operating system.
3. Run the setup file.

A series of installation panels will guide you through the installation.

Notes:

1. The product readme file contains known problems, restrictions, and considerations to review before installing.
2. See the extended installation instructions below, if you:
   - Already installed Go Webserver
   - Are using Apache server on AIX or Solaris
   - Are using Netscape Enterprise or FastTrack Server on AIX or Solaris

Installation with Go Webserver previously installed

Before installing the WebSphere Application Server on Domino Go Webserver 4.6.1, uninstall the Java servlet component of Go Webserver. This component is uninstalled automatically when you install the WebSphere Application Server on Windows NT. If installing on another operating system, follow the instructions:

Uninstalling the current Java support on AIX

1. Log on as root.
2. Shut down the Webserver if it is currently running.
3. Run the System Management Interface Tool (smit).
4. Select software installation and maintenance.

5. Select maintain installed software.
6. Select remove software products. On the resulting screen:
   a. Click the list button next to the software field.
   b. Select all internet_server.java.* files.
   c. Select yes for remove dependent software.
   d. Click OK.
7. Click OK.
8. When prompted, confirm that you want to uninstall.
9. After the Java support for Go Webserver has been uninstalled, exit smit.

Uninstalling the current Java support on Solaris

1. Log on as root.
2. Shut down the Webserver, if it is running.
3. Run admintool.
4. Open the browse menu and select software.
5. Select the Lotus Domino Go Webserver -- Java Servlets component.
6. Open the edit menu and select delete.
7. When prompted to confirm that you want to remove the package, click delete.

Installation for Apache server on AIX

After completing the WebSphere Application Server installation on AIX using the three simple steps described above, follow these additional steps, in which *applicationserver_root* is the name of the root directory in which your WebSphere Application Server is installed.

1. Copy the WebSphere Application Server plug-in *applicationserver_root*/plugins/aix/apache_124_aix_adapter.o to your Apache source tree *Apache_src_root*/src.
2. Add the following as the last line of the Configuration file *Apache_src_root*/src/Configuration:

   ```
   Module servlet_express_module apache_124_sun_adapter.o
   ```

   Note: Case is important, and the line should start in column 1.
3. Run Apache Configure:

   ```
   cd Apache_src_root/src
   ./Configure
   ```
4. Recompile and install Apache httpd (if runtime differs from your source code path):

   ```
   make
   ```

   Note: The apache_124_sun_adapter.o module must be linked to the httpd executable.
5. Update the Apache srm configuration file *Apache_install_root* /config/srm.conf:
   - Add following directory mappings (in the order shown) for WebSphere Application Server Web accessible resources and sample applications:
     – Alias /*applicationserver_root*/samples/* /*applicationserver_root*/samples/*
     – Alias /*applicationserver_root*/doc/* /*applicationserver_root*/doc/*
     – Alias /*applicationserver_root*/* /*applicationserver_root*/web/*

– Alias */applicationserver_root*/system/admin/*
/*applicationserver_root*/system/admin/*

- Add the following line as the last line of the file:

```
NcfservletConfig ncf.jvm.properties
applicationserver_root/properties/server/servlet/servletservice/jvm.properties
```

Note: Type it all on one line.

6. Verify the ncf.native.apache.outofproc.port (in *applicationserver_root*/properties/server/servlet/servletservice/jvm.properties) is not already in use (check /etc/services).
7. Update the /usr/bin/apache_servlet_eng_runner.sh shell script. You must provide values for:
   - JAVA_HOME (the JDK root directory)
   - CLASSPATH (should include the JDK classes and the WebSphere Application Server ibmwebas.jar, jst.jar, jsdk.jar, x509v1.jar, databeans.jar, and ibmjbrt.jar files)
   - PATH (should include $(JAVA_HOME)/bin)
8. Start Apache httpd.

See the product guide for information about migrating existing servlets and configuring servlet parameters. It also contains instructions for shutting down WebSphere Application Server when the Apache server is shut down.

Note: stdout for all servlets goes to *applicationserver_root*/logs/ncf.log. When WebSphere Application Server is installed, logging is disabled.

Installation for Apache server on Solaris

After completing the WebSphere Application Server installation on AIX using the three simple steps described above, follow these additional steps, in which *applicationserver_root* is the name of the root directory in which your WebSphere Application Server is installed.

1. Copy the WebSphere Application Server plug-in *applicationserver_root*/plugins/aix/apache_124_aix_adapter.o to your Apache source tree *Apache_src_root*/src.
2. Add the following as the last line of the Configuration file *Apache_src_root*/src/Configuration:

```
Module servlet_express_module apache_124_sun_adapter.o
```

Note: Case is important, and the line should start in column 1.

3. Run Apache Configure:

```
cd Apache_src_root/src
./Configure
```

4. Recompile and install Apache httpd (if runtime differs from your source code path):

```
make
```

Note: The apache_124_sun_adapter.o module must be linked to the httpd executable.

5. Update the Apache srm configuration file *Apache_install_root* /config/srm.conf:
   - Add following directory mappings (in the order shown) for WebSphere Application Server Web accessible resources and sample applications:

- Alias /*applicationserver_root*/samples/* /*applicationserver_root*/samples/*
- Alias /*applicationserver_root*/doc/* /*applicationserver_root*/doc/*
- Alias /*applicationserver_root*/* /*applicationserver_root*/web/*
- Alias /*applicationserver_root*/system/admin/* /*applicationserver_root*/system/admin/*

• Add the following line as the last line of the file:

```
NcfservletConfig ncf.jvm.properties
applicationserver_root/properties/server/servlet/servletservice/jvm.properties
```

Note: Type it all on one line.

6. Verify the ncf.native.apache.outofproc.port (in *applicationserver_root*/properties/server/servlet/servletservice/jvm.properties) is not already in use (check /etc/services).
7. Start Apache httpd.

See the product guide for information about migrating existing servlets and configuring servlet parameters. It also contains instructions for shutting down WebSphere Application Server when the Apache server is shut down.

Note: stdout for all servlets goes to *applicationserver_root*/logs/ncf.log. When WebSphere Application Server is installed, logging is disabled.

Installation for Netscape Enterprise or FastTrack Server on AIX

Note: The *applicationserver_root* will be /usr/lpp/servlet, where *applicationserver_root* refers to the root directory where WebSphere Application Server is installed.

1. Log on as root.
2. Stop the Netscape Administration Server, if it is running.
3. Set the following environment variables:
   - NS2_CONFIG_PATH (If you want to install the plug-in for Netscape Enterprise or FastTrack Server Version 2.01, set this variable to the path that contains the obj.conf file.)
   - NS3_CONFIG_PATH (If you want to install the plug-in for Netscape Enterprise or FastTrack Server Version 3.0x, set this variable to the path that contains the obj.conf file.)
   - NS3_CONFIG_PATH (If you want to install the plug-in for Netscape Enterprise or FastTrack Server Version 3.0x, set this variable to the path that contains the obj.conf file.)
4. Change the permission on the file install.sh.
5. To start the installation, run install.sh.
6. If installing on Netscape 2.0x, verify that the ncf.native.outofproc.port (in *applicationserver_root*/properties/server/servlet/servletservice/jvm.properties) is not already in use (check /etc/services).
7. Restart your Web server.
8. If you use Netscape Administration Server to update your server's object configuration file (obj.conf), you must save the WebSphere Application Server changes before you use Netscape Administration Server to make any other server configuration changes, otherwise the changes will be lost.

   To save the WebSphere Application Server changes:

- Start the Netscape Administration Server.
- Select the Web server instance on which WebSphere Application Server is installed.
- Select Apply.

See the product guide for information about migrating existing servlets and configuring servlet parameters.

Note: stdout for all servlets goes to *applicationserver_root*/logs/ncf.log. When WebSphere Application Server is installed, logging is disabled.

Installation for Netscape Enterprise or FastTrack Server on Solaris

Note: The *applicationserver_root* will be /usr/lpp/servlet, where *applicationserver_root* refers to the root directory where WebSphere Application Server is installed.

1. Log on as root.
2. Stop the Netscape Administration Server, if it is running.
3. Set the following environment variables:
   - NS2_CONFIG_PATH (If you want to install the plug-in for Netscape Enterprise or FastTrack Server Version 2.01, set this variable to the path that contains the obj.conf file.)
   - NS3_CONFIG_PATH (If you want to install the plug-in for Netscape Enterprise or FastTrack Server Version 3.0x, set this variable to the path that contains the obj.conf file.)
   - NS3_CONFIG_PATH (If you want to install the plug-in for Netscape Enterprise or FastTrack Server Version 3.0x, set this variable to the path that contains the obj.conf file.)
4. Change the permission on the file install.sh.
5. To start the installation, run install.sh.
6. If installing on Netscape 2.0x, verify that the ncf.native.outofproc.port (in *applicationserver_root*/properties/server/servlet/servletservice/jvm.properties) is not already in use (check /etc/services).
7. Restart your Web server.
8. If you use Netscape Administration Server to update your server's object configuration file (obj.conf), you must save the WebSphere Application Server changes before you use Netscape Administration Server to make any other server configuration changes, otherwise the changes will be lost.

   To save the WebSphere Application Server changes:
   - Start the Netscape Administration Server.
   - Select the Web server instance on which WebSphere Application Server is installed.
   - Select Apply.

See the product guide for information about migrating existing servlets and configuring servlet parameters.

Note: stdout for all servlets goes to *applicationserver_root*/logs/ncf.log. When WebSphere Application Server is installed, logging is disabled.

Uninstalling

On Windows NT:

1. From the Windows Start menu, choose Control Panel.
2. Click the Add/Remove programs icon.
3. Follow the instructions on the Install/Uninstall tab.

On AIX and Sun Solaris:

1. Go to the directory in which you installed the Application Server.
2. Check the file permissions to ensure undo.sh is executable.
3. Execute undo.sh.

Accessing the product documentation

To view the copies of the documentation (this book and the product Javadoc) installed on your Application Server, use one of the following methods:

- To access the documentation via your Web server, use your browser to open http://*your.server.name*/*applicationserver_root*/doc/index.html.
- To access the documentation as local files on the Web server:
  - For Windows NT, open the file *applicationserver_root*\doc\index.html
  - For AIX and Sun Solaris, open the file *applicationserver_root*/doc/index.html

The most current version of this book is available in HTML and PDF formats on the Application Server Web site. To access the online book, open the URL:

`http://www.software.ibm.com/webservers/sedoco.html`

Chapter 2. Using the Application Server Manager

Configure and manage servlets using the Application Server Manager, a Java applet. This chapter describes how to start and use the Application Server Manager and provides an overview of the tasks that you can perform using the Manager.

Starting the Application Server Manager

The Application Server Manager provides a graphical user interface for configuring and managing servlets running on your Web server. Most changes you make to configuration parameters from within the Application Server Manager take effect immediately and do not require you to restart the Web server.

Restart the Web server, however, if you:

- Change the port number for the Application Server Manager
- Change the parameters on the Basic Setup page
- Manually edit the jvm.properties file to control Native DLL or Java standard out logging

To start the Application Server Manager, enter this URL from your Web browser:

`http://`*`your.server.name`*`:9090/`

where *your.server.name* is the fully qualified name of your host. (If you start Application Server Manager in a browser on the same machine on which you installed Application Server, use `http://localhost:9090/` for better performance.)

The Application Server Manager applet starts and displays the login page.

To log in to the Application Server Manager for the first time, enter admin as the login user ID and password, and click OK. For security reasons, you should change the login password.

Note: To run the Application Server Manager, you need an appletviewer or a browser that supports Java Development Kit (JDK) 1.1, for example:

- Netscape Communicator 4.03 or 4.04 with the JDK 1.1 patch, or 4.05 (available at http://developer.netscape.com/software/jdk/download.html)
- Microsoft Internet Explorer 4
- Sun HotJava 1.1

Using the Application Server Manager

After you log in, the Application Server Manager displays the Services page, which shows the current state of the Web server and lists the services that are installed and running on your machine. The service provided is the Application Server servlet administration service, which you use to configure and manage servlets. This service listens on port 9090, unless you change the port using the Properties page.

Current status and summary information for each service is displayed on the Services page. You can stop, restart, and shut down services from this page by selecting the buttons at the bottom of the page.

Configure and manage servlets by highlighting the Application Server servlet administration service, then clicking Manage. A new window opens, displaying the Application Server Manager configuration interface.

To set parameters and view information about servlet activity, use the navigation buttons at the top of the Application Server Manager interface to display the available configuration and administration tasks. Select a task from the tree-view to display parameters and information for that task.

If you modify parameters for a task, select Save or Revert before viewing another task. If you select Save, Application Server saves the changes you made to the Application Server properties file. If you select Revert, changes you made to parameters for the current task without saving are replaced with the previously saved values.

Read the rest of this chapter for general information about configuring and managing servlets. For more detailed task information and descriptions of configuration parameters, click Help from the Application Server Manager configuration interface.

Performing servlet configuration and administration tasks

The navigation buttons at the top of the Application Server Manager window enable you to:

- Configure setup parameters
- Monitor servlet activity
- Establish and maintain security
- Manage servlets

Configuring setup parameters

Before using Application Server to manage servlets, configure basic setup parameters for servlet activity and the properties of the Application Server Manager itself. To configure properties for the Application Server Manager, select Application Server from the top of the tree on the Service page, and click the Properties button.

To configure setup parameters for servlet activity, click the server in the tree on the Services page, and click Manage. Click the Setup button from the top of the Application Server Manager interface to display the Setup tasks.

Use this task...	To set up...
Basic	Path information for Java files, whether to use system classpath, name of the Java compiler
User profile	Whether to use user profiles and user profile parameters
Session tracking	Whether session tracking is enabled and session tracking parameters
Log files	The types of messages to log, how to store and display log information, and maximum log file size

Use this task...	To set up...
Virtual hosts	Names of virtual hosts and their associated document root directories
Connection management	Connection pools

In addition to the log files you can set up using the Application Server Manager, the Application Server supports two other types of logging:

- Native DLL logging
- Java standard out logging

See the online help for Log Files for more information.

Monitoring servlet activity

Monitor servlet activity by viewing the output of various log files, viewing the status of loaded servlets, and viewing, in real time, how resources are being used. Click the Monitor button from the top of the Application Server Manager interface to display the Monitor tasks.

Use this task...	To monitor...
Log output	Information collected in the log files
Resource usage	How service resources are being used. The monitored resources are memory, a pool of handler objects, requests to the service, and the service response time.
Active sessions	Information about the user sessions that are currently active on the Web server, including information about individual sessions and summary information for all active sessions
Loaded servlets	Status and statistics for individual servlets

Establishing and maintaining security

Establish and maintain security by defining users, groups, resources, and access control lists. By assigning specific access settings to each user, group, and resource, you can control precisely how the resources of a service are used, and by whom. Click the Security button from the top of the Application Server Manager interface to display the Security tasks.

Use this task...	To specify...
Users	Who can access Web pages served by the Application Server and other resources, such as servlets
Groups	Named lists of users
Access control lists	Specific access permissions for users and groups
Resources	Security parameters for specific directories, files, and servlets on the Application Server

Managing servlets

Servlets placed in the *applicationserver_root*\servlets directory are automatically loaded and reloaded (if updated) when requested. You can also use the Application Server Manager to manage servlets more directly by defining initialization parameters and creating servlet aliases and filters. To manage servlets, click the Servlets button from the top of the Application Server Manager interface

to display the Servlets tasks.

Use this task...	To...
Add servlets	Add a servlet that you want to manage
Servlet aliases	Specify a path-mapping rule so users can use a shortcut URL to invoke a specific servlet
Filtering	Associate servlets with MIME-types so that each time a response with a specific MIME-type is generated, a particular servlet is invoked
Configure servlets	Define configuration information and initialization parameters for individual servlets, such as the associated class file for the servlet, whether the servlet loads at startup, and whether the Web server loads the servlet from a remote location

Migrating servlets

To use the Application Server Manager to manage servlets that existed on your Web server before you installed Application Server, you must first migrate those servlets. To migrate existing servlets, move the servlets from their current location to *applicationserver_root*\servlets. The Application Server monitors this directory and automatically reloads servlets when the servlets change.

If you have servlets in other directories and do not want to move them to the *applicationserver_root*\servlets directory, you can specify additional directories to be monitored:

1. Open the servlets.properties file, which is in the *applicationserver_root*\properties\server\servlet\servletservice directory.
2. Set the servlets.classpath to the fully-qualified path of the directory or directories to be monitored. For example:

```
servlets.classpath=c:\aaa;\d:bbb;d:\ccc  (Windows NT)
servlets.classpath=/tmp/aaa:/user/jones/servlets (AIX and Sun Solaris)
```

Use the Servlets Configuration page in the Application Server Manager to reconfigure previous servlet parameters.

Verifying Application Server configuration settings

To verify that your Application Server configuration settings work correctly, invoke the snoop servlet provided with Application Server. Use your Web browser to open the servlet URL:

http://*your.server.name*/servlet/snoop

The snoop servlet should echo back information about the HTTP request sent by the client and the initialization parameters of the servlet.

Troubleshooting servlet problems

Read this section for information that might help you if you have problems executing servlets or using the Application Server Manager.

Servlet parameter changes not updated

If you change a servlet parameter and the new value is not updated, ensure that the directory containing the *.properties files does not contain any backup files with a .properties extension. If you make a backup copy of any of the *.properties files, use a different file name extension, such as .bak.

Application Server does not shut down when Apache is shut down

Due to programmatic limitations with the Apache API, the Application Server out-of-process engine does not automatically stop when the Apache Web server is shut down. To automatically shut down the Application Server when the Apache server is stopped, put the commands below in a shell script and execute the script after Apache stops. If you currently use an automated script to stop Apache, add these lines. Before using the script, edit the userid variable.

If you do not use these automated commands, shut down the out-of-process engine manually by issuing a kill command for both the out-of-process engine and the Java virtual machine process. If you do not, you will not be able to restart Application Server.

```
#!/bin/ksh
# userid of apache server
#NOTE: Please edit this value to correspond to the userid that the web server runs
# as specified in the User directive of httpd.conf
user=nobody

os=′uname -a|awk  '{print $1}''

# clean-up Application Server out-of-process procs
echo "bypass fix for Application Server residual OutOfProc"

if [[ $os =   "SunOS" ]]; then
   pid=′ps -eaf|grep -v grep|grep "^ *${user}"|grep "native_threads"|awk '{print $2}''
fi

if [[ $os =   "AIX" ]]; then
   pid=′ps -eaf|grep -v grep|grep "^ *${user}"|grep "OutOfProcessEng"|awk '{print $2}''
fi

if [[ $pid -ne 0 ]]; then
   echo "killing Application Server OutOfProcessEng pid $pid";
   kill $pid;
fi
```

Chapter 3. Using the servlet APIs

This section:

- Provides an overview of servlets
- Describes the Java class packages for writing Java servlets
- Helps you understand how the Java classes work

Overview of servlets

Similar to the way applets run on a browser and extend the browser's capabilities, servlets run on a Java-enabled Web server and extend the server's capabilities. Servlets are Java programs that use the Java Servlet Application Programming Interface (API) and the associated classes and methods. In addition to the Java Servlet API, servlets can use Java class packages that extend and add to the API.

Servlets extend server capabilities by creating a framework for providing request and response services over the Web. When a client sends a request to the server, the server can send the request information to a servlet and have the servlet construct the response that the server sends back to the client.

A servlet can be loaded automatically when the Web server is started, or it can be loaded the first time a client requests its services. After loading, a servlet continues to run, waiting for additional client requests.

What servlets can do

Servlets perform a wide range of functions. For example, a servlet can:

- Create and return an entire HTML Web page containing dynamic content based on the nature of the client request
- Create a portion of an HTML Web page (an HTML fragment) that can be embedded in an existing HTML page
- Communicate with other server resources, including databases and Java-based applications
- Handle connections with multiple clients, accepting input from and broadcasting results to the multiple clients. A servlet can be a multi-player game server, for example.
- Open a new connection from the server to an applet on the browser and keep the connection open, allowing many data transfers on the single connection. The applet can also initiate a connection between the client browser and the server, allowing the client and server to easily and efficiently carry on a conversation. The communication can be through a custom protocol or through a standard such as IIOP.
- Filter data by MIME type for special processing, such as image conversion and server-side includes (SSI)
- Provide customized processing to any of the server's standard routines. For example, a servlet can modify how a user is authenticated.

Servlet lifecycle

The lifecycle of a servlet begins when it is loaded into Web server memory and ends when the servlet is terminated or reloaded.

Initialization

The servlet is loaded during the following times:

- Automatically at server startup, if that option is configured
- At the first client request for the servlet after server startup
- When the servlet is reloaded

After the servlet is loaded, the server creates an instance of the servlet and calls the servlet init() method. During the initialization, the servlet initialization parameters are passed in the servlet configuration object.

Request processing

A client request arrives at the server. (The client request will already be at the server if the client request initiated the servlet load.) The server creates a Request object and a Response object that are specific to the request. The server invokes the servlet service() method, passing the Request and Response objects.

The service() method gets information about the request from the Request object, processes the request, and uses methods of the Response object to pass the response back to the client. The service() method can invoke other methods to process the request, such as doGet(), doPost(), or methods you write.

Termination

When the server no longer needs the servlet or a new instance of the servlet is being reloaded, the server invokes the servlet's destroy() method.

Servlet advantages

Because Java servlets have advantages over CGI programs and low-level Web server APIs, such as Internet Services Application Programming Interface (ISAPI) and Netscape Connection Application Programming Interface (NSAPI), you might choose to use Java servlets to extend your server's functions. Java is rapidly becoming an industry standard. Learning Java lets you use a single, flexible, and powerful tool to meet most of your programming needs. Java's cross-platform and cross-server support enable you to maximize your servlet development investment.

Advantages over CGI programs

CGI is a standard supported by most Web servers. It defines how information can be exchanged between the Web server and external programs, namely the CGI programs. CGI programs are usually written in interpreted languages, such as UNIX shell scripts or PERL, although they can be written in compiled languages, usually C. Java servlets and CGI programs allow you to extend only the server's service processing step.

CGI programs often have performance problems because:

- Interpreted languages are slower than compiled languages.

- CGI programs run in a separate process from the server. This requires significant start-up time every time a client requests a CGI program.

In contrast, servlets can run in the same process as the server. After a servlet is loaded, it remains loaded, eliminating the start-up time for subsequent client requests.

Advantages over ISAPI and NSAPI programs

Low-level server APIs are usually written as C programs. In contrast, servlets are written in Java. Java, a new language based on C syntax, has benefited from the many years of C development.

- Java programs do not have memory leaks caused by memory allocation errors. C programmers can spend much time searching for and fixing memory leaks.
- Java programs do not use pointers. Wayward pointers in C programs are notorious for causing difficult-to-fix bugs, segmentation violations, and potential security problems.
- Java programs are more portable than C programs. Threads, for example, which are not a native part of C, are native to Java. Not all C compiler vendors implement threads the same, causing portability problems.

Servlet packages

Several packages that you need to write your own servlets are available in the Java Servlet Development Kit (JSDK) Version 1.1, which is included with the Application Server. They include the javax.servlet and javax.servlet.http packages, which serve as the basis for all servlets. Find a complete description of these packages at http://jserv.javasoft.com/products/java-server/documentation/webserver1.1/apidoc/packages.html.

In addition, the Application Server includes its own packages that extend and add to the JSDK to make it easier to maintain user information, create dynamic, personalized Web pages, and analyze Web site usage. The Javadoc for the Application Server APIs is installed in the product doc/apidocs directory. You can also find the Javadoc on the Application Server library page at http://www.software.ibm.com/webservers/.

The Application Server API packages are:

- **com.ibm.servlet.personalization.sessiontracking**

  This Application Server extension to the JSDK records the referral page that led a visitor to your Web site, tracks the visitor's position within the site, and associates user identification with the session. IBM has also added session clustering support to the API.
- **com.ibm.servlet.personalization.userprofile**

  Provides methods to maintain detailed information about your Web visitors and incorporate it in your Web applications, so that you can provide a personalized user experience.
- **com.ibm.servlet.personalization.sam**

  Lets you register your own Web applications with the Site Activity Monitor, an applet that gives you a dynamic, real-time view of the activity on your Web site.
- **com.ibm.servlet.servlets.personalization.util**

  Includes specialized servlets that allow Web site administrators to dynamically post bulletins and enable Web site visitors to exchange messages.

Extending HttpServlet

If writing servlets that handle HTTP requests, create your servlets by extending HttpServlet. The HttpServlet class is a subclass of GenericServlet, with specialized methods for handling HTML forms.

HTML forms, defined by the <FORM> and </FORM> tags, let Web users send data from their client browsers to the server for processing. They typically include input fields (such as text entry fields, check boxes, radio buttons, and selection lists) and a button to submit the data. They also specify which program or servlet the server should execute when the information is submitted. The information can be passed to the server using GET or POST, depending on the HTML form. The HttpServlet methods let you easily determine if the input data was from a POST or GET, and then process it accordingly.

Required servlet methods

HttpServlet classes contain the init(), destroy(), and service() methods. The init() and destroy() methods are inherited. For general servlets, define the service() method. For HTTP servlets, define one or more of the following methods as needed:

- doGet()
- doPost()
- doPut()
- doDelete()

Note: The HttpServlet class also provides methods for the HTTP1.1 extensions: OPTIONS, PUT, DELETE, and TRACE. If you override the HttpServlet service() method, you will lose the default processing for their corresponding methods.

For additional tips, see “Using common methods” on page 87.

init()

The init() method executes only one time during the lifetime of the servlet. It executes when the server loads the servlet. You can configure the server to load the servlet when the server starts or when a client first accesses the servlet. The init() is not repeated regardless of how many clients access the servlet.

The default init() method is usually adequate but can be overridden with a custom init() method, typically to manage servlet-wide resources. For example, you might write a custom init() to load GIF images only one time, improving the performance of servlets that return GIF images and have multiple client requests. Another example is initializing a database connection.

destroy()

The destroy() method executes only one time — it executes when the server stops and unloads the servlet. Typically, servlets are stopped as part of the process of bringing the server down.

The default destroy() method is usually adequate, but can be overridden, typically to manage servlet-wide resources. For example, if a servlet accumulates statistics

while it is running, you might write a destroy() method that saves the statistics to a file when the servlet is unloaded. Another example is closing a database connection.

service()

The service() method is the heart of the servlet. Unlike init() and destroy(), it is invoked for each client request.

In HttpServlet, the service() method already exists. The default service function invokes the do function corresponding to the method of the HTTP request. If the HTTP request method is GET, then doGet() is called. If the request method is POST, then doPost() is called. A servlet should override the do functions for the HTTP methods that the servlet supports.

If the form information is provided by GET, the service() method invokes the doGet() method. If the form information is provided by POST, the service() method invokes the doPost() method. You must provide code to override doGet() or doPost() to retrieve the information and to prepare the response.

When a client invokes a servlet directly by the servlet's URL, the client uses GET. For example, requesting `http://myserver/servlet/MyServlet` results in a GET. When a client invokes a servlet through an HTML form submission, the form tag specifies if the method is GET or POST. For example, `<FORM method=POST action=/servlet/MyServlet>` results in a POST. If the form tag specified `method=GET`, it would result in a GET. Whether to override doGet, doPost, or both methods depends on how you want clients to be able to invoke your servlet:

- For your servlets to be invoked by a GET request only, override doGet only.
- For your servlets to be invoked by a POST request only, override doPost only.
- For your servlet to be invoked by either type of request, override both doGet and doPost.

Exchanging information with the client

When the server calls an HTTP servlet's service() method, it passes two objects as parameters:

- HttpServletRequest
- HttpServletResponse

They are commonly called the Request and Response objects. It is through these two objects that the servlet communicates with the server, and ultimately with the client. The servlet can invoke the Request object's methods to get information about the client environment, the server environment, and any information provided by the client (for example, form information set by GET or POST). The servlet invokes the Response object's methods to send the response that it has prepared back to the client. These same objects are also passed to the doGet() and doPut() methods.

Getting Request information

The Request object has specific methods to retrieve information provided by the client:

- getParameterNames()

- getParameter(), which is deprecated
- getParameterValues()

Additionally, the getQueryString() method is available for HttpServletRequest objects. You will also find useful the utility methods parseQueryString() and parsePostData() in the HttpUtils object.

If your servlet is derived from HttpServlet, use an HttpServletRequest object to retrieve the data in the query string. For example, to pass the first name John and the last name Doe, the client requests http://your.server.name/servlet/MyServlet?firstname=John&lastname=Doe. The getParameterNames() method returns a string array containing the firstname and lastname parameters in the above example. The getParameter() method, when provided a parameter name, returns its corresponding value. In the above example, it could return “John” or “Doe” for the parameters firstname and lastname, respectively.

These methods fully decode the parameter names and values. For example, a %23 entered in the query string in the URL is returned as a # by these methods. It is possible that one parameter name has been encoded with several values (for example, topping=onions&topping=pepperoni). In this case, the getParameterValues() method, when provided the topping parameter, returns a string array of all the values (for example, “onions” and “pepperoni”).

Returning Response information

The Response object creates a response to return to the requesting client. Its methods allow you to set the Response header and the Response body. The Response object also has the getWriter() method to return a PrintWriter object. Use the print() and println() methods of the PrintWriter object to write the servlet Response back to the client. You can also use the HTML template syntax described in “Developing HTML templates for variable data” on page 69.

Maintaining state in your Web applications

Together, the JSDK API and the Application Server extensions provide a framework for building sophisticated state models on top of the Web’s stateless protocol. Using this *session tracking* framework, your server can maintain state information by coordinating server requests from individual users into a session.

A *session* is a series of requests originating from the same user, at the same browser. The class that handles the session tracking is com.ibm.servlet.personalization.sessiontracking.IBMSessionContextImpl, called the *Session Tracker*. The Session Tracker extends com.sun.server.http.session.SessionContextImpl. By default, it is always active with the Application Server.

There are several stages to interacting with the Session Tracker:

- Establishing a session

  As specified in the JSDK API, you obtain an HttpSession object using the HttpServletRequest object’s getSession() method. When you first obtain the HttpSession object, the Session Tracker creates a unique session ID and typically sends it back to the browser as a cookie. Each subsequent request from this user

passes this cookie that contains the session ID. The Session Tracker uses this session ID to find the user's existing HttpSession object.

Note: If the browser does not accept cookies, this implementation does not work. See "Maintaining state without cookies" on page 25 for the alternate way to maintain state. See http://www.netscape.com/newsref/std/cookie_spec.html for a detailed explanation of cookies.

- Accessing session data values

  After a session is established, you can access the available session data: the referral page, user position, user name, session start time, and last access time.

- Storing user-defined data in the session

  After a session is established, you can add user-defined data to the session.

- Using the HttpSessionBindingListener interface

  Before a session ends, objects stored in a session which implement the HttpSessionBindingListener interface are notified of the session's impending termination. This enables you to perform post-session processing, such as saving data to a database.

  Note: See "Session clustering" for information about the HttpSessionBindingListener interface in a session cluster environment.

- Ending a session

  A session can be terminated automatically by the Session Tracker or specifically by your servlet processing.

Configuring the Session Tracker

Use the Application Server Manager to configure the Session Tracker parameters:

1. Open the Application Server Manager at http://*your.server.name*:9090/ and log in.
2. In the list of Services, double-click the Application Server instance to manage.
3. On the next page, select the Setup button.
4. Select Session Tracking from the tree-view section of the Setup page to display the Session Tracking parameters. Review and change as needed the parameters on all of the tabs. Select Help for explanations of the parameters.

Session clustering

Session Tracker allows for more than one instance of the Application Server to share a common pool of sessions (*session cluster*). Each instance of the Application Server can be configured for one of three modes:

Standalone host
: The instance of the Application Server does not participate in a Web server cluster. It maintains its own session information, does not request session information from a server, and does not respond to client requests for session information.

Session cluster client
: In an environment where Web servers are clustered by using a product, such as IBM eNetwork Dispatcher, the instance of the Application Server contacts the specified session cluster host.

Session cluster server
In a clustered Web server environment, a specified server in the cluster serves as the cluster host for sessions.

To configure an instance of the Application Server to use one of these Session Tracker modes:

1. Open the Application Server Manager at http://*your.server.name*:9090/ and log in.
2. In the list of Services, double-click the Application Server instance to manage.
3. On the next page, select the Setup button.
4. Select Session Tracking from the tree-view section of the Setup page to display the Session Tracking parameters.
5. Select the Host tab.
6. Set the mode:
   - To enable standalone host mode:
     a. Set Standalone Host to Yes.
     b. Leave the Session Cluster Server and Protocol Switch Host fields blank. When Standalone Host is set to Yes, any values in those fields are ignored.
   - To enable session cluster client mode:
     a. Set Standalone Host to No.
     b. In the Session Cluster Server field, specify the hostname or IP address of another instance of the Application Server that will maintain the common pool of sessions.
   - To enable session cluster server mode:
     a. Set Standalone Host to No.
     b. Leave the Session Cluster Server field blank, which causes this instance of the Application Server to be set up as the session server.
7. (Optional) If this instance of the Application Server is configured for URL rewriting and protocol switching, and the Web servers are clustered, specify the hostname or IP address of the cluster host in the Protocol Switch Host field. If you specify a value, Web browsers use the cluster hostname and that host in determining whether protocol switching will occur.

The following caveats regard how session tracking works in a clustered Web server environment:

- Whenever a session is obtained using the getSession method of the HttpServletRequest implementation, the cluster server places a lock on the session and propagates the session wherever it is needed. After any modifications are made to the session and the service method of the HttpServletRequest implementation has ended, the session is automatically sent back to the server to update its copy of the session and the lock is released. The session obtained using the HTTP request can be thought of as the servlet's current session, or the session associated with or owned by the current HTTP request.
- A session can also be obtained by using the getSession method of the HttpSessionContext implementation (IBMSessionContextImpl). A servlet can access the HttpSessionContext getIds method and then access sessions other than the one associated with a HTTP request using the context's getSession method. When a session is obtained this way, the programmer must manually unlock the

session with the sync method of IBMSessionContextImpl. The sync method automatically updates the cluster server's version of the session as well.

- When HttpSessionBindingListener and HttpSessionBindingEvent are used in a clustered Web server environment, the event will be fired in the Application Server where the session resides. This location will be the cluster server, if the session has timed out. If the session is invalidated using a call to the invalidate method of the session object, the location can be either any of the possible cluster clients of the cluster client.
- After an instance of the Application Server has been configured as a session cluster client, all other configuration parameters (except for the enable session support parameter) are accessed from the specified session cluster server. The local copy of the parameter settings (the session.properties file) remains on the session cluster client disk. Except for changes to the enable session support parameter, any attempts to change the parameters at the session cluster client are ignored by the Session Tracker while it is a session cluster client. When a Session Tracker is changed from a session cluster client to one of the other modes, the Session Tracker uses the parameter settings. This design guarantees a consistent configuration across the session cluster.
- In the current Java Servlet API (as specified by Sun Microsystems), the definition of the putValue() method of the HttpSession interface does not account for the possibility of a clustered environment. If you add an object that does not implement the Serializable interface to a session, you do not have a way to propagate the object along with a given session, as the session is serialized across the cluster. The object will not be sent from the cluster client to the cluster server when session updates are made in the cluster client. To make your applications portable to a clustered environment, you must make any objects placed in a session serializable.

Establishing a session and storing data

The following example shows how to establish a session, store application-specific data in the session, and generate output. The doGet() method in this sample servlet returns an HTML page displaying the number of times the servlet has been accessed on this session. When you run this servlet, every time you reload the page, the servlet increments a counter and displays the total again.

The first part of the doGet() method associates the Session object with the user making the request. The second part of the method gets an integer data value from the Session object and increments it. The third part outputs an HTML page that includes the current value of the counter.

```
public void doGet (HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException
  {
      boolean create = true;
    // Part 1: Get the Session object
      HttpSession session = request.getSession(create);

   // Part 2: Get the session data value
     Integer ival = (Integer)
       session.getValue ("sessiontest.counter");
     if (ival == null) ival = new Integer (1);
     else ival = new Integer (ival.intValue () + 1);
     session.putValue ("sessiontest.counter", ival);

   // Part 3: Output the page
     response.setContentType("text/html");
     PrintWriter out = response.getWriter();
     out.println("<html>");
```

```
    out.println("<head><title>Session Tracking Test</title></head>");
    out.println("<body>");
    out.println("<h1>Session Tracking Test</h1>");
    out.println ("You have hit this page " + ival + " times");
    out.println("</body></html>");
  }
```

Obtaining a session: Part 1 of this example obtains an HttpSession object via a call to the instance method of the implementation of the JSDK HttpServletRequest interface, specifically the getSession method of the HttpServletRequest interface implementation. It uses a Boolean value previously set to true, so that an HttpSession is created if one does not exist.

```
boolean create = true;

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

Storing user-defined data in the session: When you have obtained a HttpSession object, you can get the session data values. The HttpSession object has methods similar to those in java.util.Dictionary for adding, retrieving, and removing arbitrary Java objects.

In Part 2 of the example, the servlet reads an Integer object from the HttpSession, increments it, and writes it back.

```
Integer ival = (Integer)
session.getValue ("sessiontest.counter");
if (ival == null) ival = new Integer (1);
else ival = new Integer (ival.intValue () + 1);
session.putValue ("sessiontest.counter", ival);
```

Note: You can use any name to identify values in the HttpSession, such as sessiontest.counter used in this example. When choosing names, remember that the HttpSession object is shared among servlets that the user might access. Servlets can access or overwrite one another's values in the HttpSession object. Consider adopting a site-wide naming convention to avoid collision between servlets, while allowing sharing of values as needed.

Ending a session

When a user session ends, the Session Tracker invalidates and removes from the system the session's HttpSession object and the object's contained data values. After invalidation, if the user attempts another request, the Session Tracker detects that the user's HttpSession was invalidated and creates a new HttpSession. Data from the user's previous session is lost.

The Session Tracker automatically invalidates sessions, or you can specifically request invalidation in your servlets.

- Automatic invalidation

  HttpSession objects that have no page requests for a specified period of time are automatically invalidated by the Session Tracker. This period of time is set by the sessionInvalidationTime parameter. The default is 30 minutes. To specify a different interval, use the Application Server Manager to set the Invalidation Interval parameter on the Interval tab.
- Requested invalidation

  Invalidate HttpSession objects by calling HttpSession.invalidate(). This causes the session to be invalidated immediately.

Using the HttpSessionBindingListener interface

The Session Tracker supports the JSDK's javax.servlet.http.HttpSessionBindingListener interface and the javax.servlet.http.HttpSessionBindingEvent, which allow you to implement customized end-of-session processing.

When an HttpSession contains an object that implements the HttpSessionBindingListener, the listener notifies the object when the Session Tracker notifies the listener that the object is being bound and unbound from a session. Use HttpSession.putValue() to bind an object to a session and HttpSession.removeValue() to unbind it from a session. Session invalidation also implies that any objects currently bound to a session will be unbound.

Because the HttpSession object itself is passed as part of the HttpSessionBindingEvent, the HttpSessionBindingListener interface lets you save any part of the session before the session is removed. It does not matter whether the session ends automatically or by a specific request. The interface contains two abstract methods, which you override with your own processing:

- valueBound() is called when the object is being bound into a session
- valueUnbound() is called when the value is being unbound from a session

Maintaining state without cookies

The Session Tracker uses a unique session ID to match user requests with their HttpSession objects on the server. When the user first makes a request and the HttpSession object is created, the session ID is sent to the browser as a cookie. On subsequent requests, the browser sends the session ID back as a cookie and the Session Tracker uses it to find the HttpSession associated with this user.

There are situations, however, in which cookies will not work. Some browsers do not support cookies. Other browsers allow the user to disable cookie support. In such cases, the Session Tracker must resort to a second method, URL rewriting, to track the user session.

With URL rewriting, all links that you return to the browser or redirect have the session ID appended to them. For example, this link:

```
<a href="/store/catalog">
```

is rewritten as

```
<a href="/store/catalog;$sessionid$DA32242SSGE2">
```

When the user clicks this link, the rewritten form of the URL is sent to the server as part of the client's request. The Session Tracker recognizes ;$sessionid$DA32242SSGE2 as the session ID and uses it to obtain the proper HttpSession object for this user.

If you code your Web pages as Java Server Pages (JSP), the JSP processor automatically handles URL rewriting. If a JSP page contains a link, the link is wrapped in an encode URL method when the page is processed. See "Developing JavaServer Pages (JSP)" on page 59 for more information about JSP.

To maintain session information without cookies for URLs that are not invoked from a JSP file, you will need to enable URL rewriting as described in the sections that follow.

Enabling URL rewriting

To rewrite the URLs you are returning to the browser, call the encodeUrl() method in your servlet before sending the URL to the output stream. For example, if a servlet that does not use URL rewriting has:

```
out.println("<a href=\"/store/catalog\">catalog<a>");
```

replace it with:

```
out.println("<a href=\"");

out.println(response.encodeUrl ("/store/catalog"));
out.println("\">catalog</a>");
```

To rewrite URLs that you are redirecting, call the encodeRedirectUrl() method. For example, if your servlet has:

```
response.sendRedirect ("http://myhost/store/catalog");
```

replace it with:

```
response.sendRedirect (response.encodeRedirectUrl("http://myhost/store/catalog"));
```

The encodeUrl() and encodeRedirectUrl() methods are part of the HttpServletResponse object. They are distinct because they follow different rules for determining if a URL should be rewritten. They both perform two functions:

1. Determine URL rewriting: The method determines if the URL needs to be rewritten. Rules for URL rewriting are somewhat complex. In general, if the server detects that the browser supports cookies, the URL is not rewritten. The server tracks information indicating whether or not a particular user's browser supports cookies.
2. Return the URL: If the method determines that the URL needs to be rewritten, the session ID is appended to the URL and returned. Otherwise, the URL is returned unmodified.

URL rewriting and multiple servlets

Unlike maintaining sessions with cookies, maintaining sessions with URL rewriting impacts the application writer because each servlet in the application must use URL encoding for every HREF attribute on <A> tags. Sessions will be lost if one or more servlets in an application does not call encodeURL or encodeRedirectURL.

The JSP processor automatically uses URL rewriting to encode HREFs when the Session Tracker is configured to enable URL rewriting. (See "Configuring the Session Tracker" on page 21 for details.) In addition to enabling URL rewriting, you need a servlet or a JSP file that serves as an entry point. That entry is not dependent on sessions for its processing, but it contains encoded HREFs to servlets in the application that are dependent on sessions.

If you want to use URL rewriting to maintain state, do not include links to parts of your Web applications in plain HTML files (files with .html extension). This restriction is necessary because URL encoding cannot be used in plain HTML files. To maintain state using URL rewriting, every page that the user requests during the session must be a page that can be processed by JSP. If you have such plain

HTML files in your Web application and portions of the site that the user might access during the session, convert them to JSP files.

Managing user authentication

The hosting HTTP server provides authorization for Web pages and servlets. However, you can provide your own access control logic based on data you maintain either in an existing customer database or in one you create for your Web site applications.

If you do your own authentication and use the Application Server Session Tracker and the IBM implementation of the HttpSession interface (IBMSessionContextImpl and IBMSessionData), your servlets can call setUserName() on the HttpSession object to make your authenticated user name available for the remainder of the session. The hosting server will continue to use its own authenticated name for its access control, while your applications can use the results of getUserName() for their access control.

When a servlet obtains an HttpSession, the Application Server Session Tracker first calls the servlet API to get the authenticated name (from the hosting server, the Application Server, or both). The authenticated name could be null. If the name is null, the Session Tracker returns the value ″anonymous″ for the HttpSession's user name until a call to setUserName() is made. If the application needs to know if the name being returned is the original name from the HTTP server, or one from a setUserName() call, call isAppUserName() to find out.

Choose, application by application, whether to do your own authentication or rely on the hosting HTTP server. Mixing these two methods within a site is possible, but might cause confusion. To have a single logon across your site, or if you will be archiving data for later analysis, do not mix the two methods. An alternative is to extend your Web server using NSAPI.

Maintaining persistent user information

The Application Server includes classes in the com.ibm.servlet.personalization.userprofile package that make it easier to maintain persistent information about your Web site visitors and use that information to customize your Web pages. The UserProfile class holds basic information about the user.

The UserProfile object is associated with the user's HttpSession object through the unique user name. The value of the user name is set and returned by the SessionData getUserName() and setUserName() methods. (Alternatively, you can use the UserProfile class and assign unique user names without using the HttpSession object.) The user name can serve as the key to access existing customer databases or to implement persistent data about the user.

The UserProfile class includes data members for a visitor's complete name, postal and e-mail addresses, and telephone numbers, and has fields to store language of choice, employment, and user-defined group information. In addition, it has a generic message, a shopping cart, and a clipboard that is a Java hashtable. This allows you to easily incorporate other objects of your choice into these data members and handle them as part of the UserProfile class. Because these objects

are persistent across successive instantiations of UserProfile, they must be serializable. If using IBM DB2, the objects will be stored in the database. Otherwise, they will be stored as files.

To successfully use the UserProfile class, you must have a JDBC-accessible database where you store the UserProfile data and the HTTP server must have access to it. Use the Application Server Manager to configure UserProfile with the following information:

- The name of the actual database product (such as IBM DB2)
- The name of the JDBC driver for the database
- The name of the database used to store tables and data. Create the database before you use UserProfile.
- The ID of the database owner
- The name of the database table that holds the UserProfile data. The UserProfile class will automatically create this table.
- The user ID for accessing the database and its tables. The user ID can differ from the owner ID. If the field is left blank, the user ID is set to the user ID that the Application Server runs under.
- The password associated with the above user ID (not owner ID) for accessing the database

Extending the UserProfile class

You can extend the UserProfile class to create a subclass that better fits your business needs. For example, you can add the capability to store data items other than those already specified in the class. You can also add new function, such as the capability to search for a user profile record in the database by using application-specific search criteria.

The Application Server Manager lets you specify whether the Application Server should use the default UserProfile class or your specialized subclass to maintain UserProfile objects across all your Web applications. The static addUserProfile() method automatically constructs the appropriate object according to this setting.

For your convenience, the Application Server includes an example of how to extend the UserProfile class to add fields. The example illustrates the steps you take to change the methods of the base class to incorporate new data members. Find this example class (UserProfile2.java) and the base UserProfile class source code in the path *applicationserver_root*\samples\userprofile.

Chapter 4. Using the connection manager

This chapter:

- Describes the problems faced by Web-based applications that access data servers
- Discusses how the connection manager solves these problems for Web-based Java servlet applications
- Describes the connection manager architecture, so you can choose the best configuration for your situation
- Tells you where to find configuration instructions and guidelines
- Illustrates how a servlet uses the connection manager
- Introduces APIs you will be concerned with
- Discusses other servlet coding considerations

Connection manager overview

The term data servers is a convenient shorthand for referring to many different data sources, including:

- Relational databases such as DB2, Oracle, Informix, and Sybase
- Other types of products, whose special services for managing and accessing data vary by product

Typically, the data server is part of a ″three tier″ application in which several machines are networked:

- The first tier is the user's machine, providing the visual interface to the application. The interface is often a Web browser.
- The second tier machine contains the business logic of the application, which often runs in conjunction with a Web server, such as the Lotus Domino Go Webserver.
- The third tier machine runs the data server, which manages the data that is used by the application.

Technical documentation on the three tier architecture usually describes significant advantages in managing and scaling three tier applications compared to other application architectures. The advantages of the three tier architecture will not be further discussed here. The connection manager runs under the Application Server and, if you are using the three tier architecture, both the Application Server and the connection manager run on the second tier machine.

Web-based applications and traditional, non-Web applications interact similarly with data servers. Typically, an application connects to the data server, spending server resources to create the connection. The application uses the connection to interact with the data server – retrieving data, updating data, and so on. The application disconnects from the data server, using further resources.

For a non-Web application, the overhead required to make and break connections is small compared to the resources required for the entire interaction. For example, consider intra-company applications. These typically have a small, predictable

number of users with relatively long user interaction times. The length of interactions drives resource usage, as relatively little time is spent connecting and disconnecting.

In contrast, Web-based applications accessing data servers incur higher and less predictable overhead, because users connect and disconnect more frequently. Often, more resources are spent connecting and disconnecting than are spent during the interactions themselves. Users interactions are typically shorter, due to the ″surfing″ nature of the Internet. Users are often from outside the company (Internet, rather than intranet) making usage volumes larger, and harder to predict.

For example, consider a Web−based application that interacts with three users for one minute each. At the same time, a cash machine (running on a non-Web application) outside a bank performs one transaction that lasts three minutes. If it takes the same amount of time for each application to make and break a connection, the Web application clearly incurs higher overhead per unit of interaction time. Both applications might spend the same amount of time establishing a single connection, but the Web-based application spends less time *using* the connection – it lacks the same ″return on investment.″

The connection manager lets you control and reduce the resources used by your Web-based applications. The connection manager spreads the connection overhead across several user requests by establishing a pool of connections, which user servlets can use. Put another way, each user request incurs the overhead of only a fraction of the cost of a connect/disconnect. After the initial resources are spent to produce the connections in the pool, additional connect/disconnect overhead is insignificant because the existing connections are reused repeatedly.

The servlets use the connection pool as follows: When a user makes a request over the Web to a servlet, the servlet uses an existing connection from the pool, meaning the user request does not incur the overhead of a data server connect. When the request is satisfied, the servlet returns the connection to the connection manager pool for use by other servlets. The user request therefore does not incur the overhead of a data server disconnect.

The connection manager also lets you control the number of concurrent connections to a data server product. This is particularly useful if the data server license agreement limits you to a certain number of users – create a pool for the data server and set the Maximum Connections parameter for the connection manager pool equal to the maximum number of users permitted by the data server product. Note that this does not guarantee compliance if you connect to the data server with other programs, bypassing the connection manager.

Connection manager architecture

The connection manager maintains a pool of open data server connections to specific data server products. Each data server can have one or more identical or non-identical pools. Multiple data servers can be supported by one running instance of the connection manager.

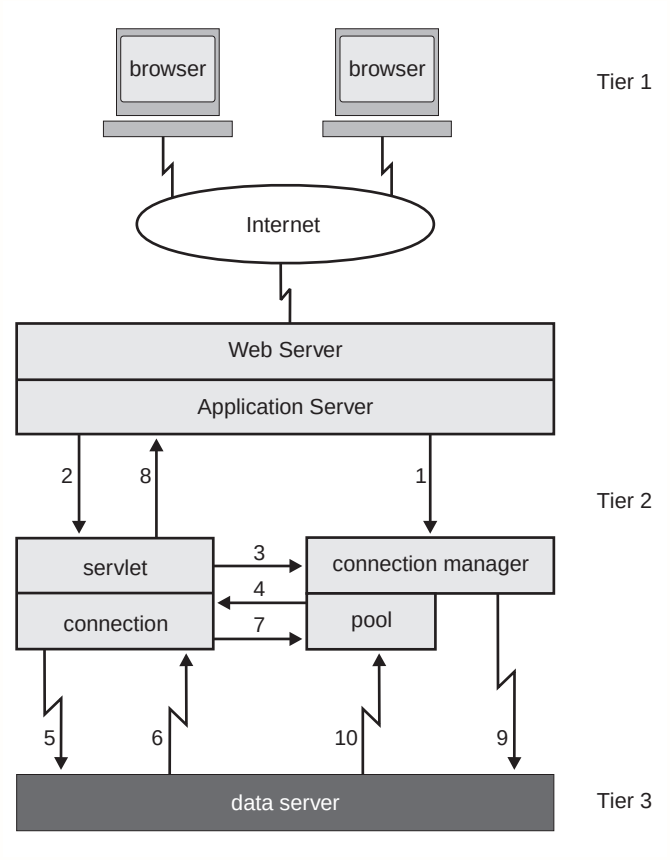

Figure 1. Connection manager architecture

Figure 1 illustrates the typical interactions between the connection manager and a servlet seeking to use a connection from the connection manager's connection pool. The following list traces the sequence followed when a connection is available when requested. As you will see later, the sequence changes if a connection is not available when the servlet requests one.

1. The connection manager, which runs under the Application Server, is loaded by the Application Server when the first servlet tries to communicate with the connection manager. The connection manager stays loaded as long as the Application Server is running.
2. The Application Server passes a user request to a servlet.
3. The servlet uses methods of the connection manager to request a connection from the pool.
4. The pool gives the servlet a connection.
5. The servlet uses the connection to talk directly to the data server, using the standard APIs for the specific data server.
6. The data server returns data through the connection to the servlet.
7. When the servlet ends communications with the data server, the servlet returns the connection to the pool for use by another user request.
8. The servlet sends the response back through the Application Server to the user.

A connection is not always available from the pool when the servlet requests it (in Figure 1, see the action labeled 3). In this case, the connection manager communicates directly with the data server. It:

- requests a new connection (9)
- adds the connection to the pool (10)
- gives the new connection to the servlet (4)

(The numbers refer to Figure 1 on page 31.)

Note that the connection manager will not add a new connection to the pool if the specified limit for the number of connections in the pool has been reached.

Creating a new connection for the pool is a high overhead task and the new connection will use resources on the data server machine. For this reason, the connection manager seeks to maximize the probability that a servlet can get an existing connection from the pool. At the same time, the connection manager must minimize the idle connections in the pool, as they are a significant waste of resources.

The connection manager works with the servlet to perform these minimizing and maximizing tasks. For optimal performance, choose the proper settings for the connection manager parameters. See the Application Server Manager online help for a detailed discussion of these parameters.

The connection manager maintains the verify timestamp, last-used timestamp, and in-use flag of each connection. When a servlet first gets a connection, the connection's verify timestamp and last-used timestamp are set to the current time and the connection's in-use flag is set to true.

If the servlet is going to use the connection to communicate with the data server multiple times over an extended period, you may want to add code to the servlet to verify that it still owns the connection just before each use of the connection. (This is because the connection manager can be configured to take a connection away from a servlet that has not used the connection for a length of time specified by the connection manager Maximum Age parameter.) To verify that it still owns a connection, the servlet invokes the verifyIBMConnection() method that in turn gets the connection manager to check the verify timestamp of the connection. If the servlet still owns the connection, the last-used timestamp is automatically updated to the current time, as part of invoking the verifyIBMConnection() method. The servlet then uses the connection to communicate with the data server, confident that the connection will work. When the servlet finishes with the connection, it releases it back to the connection pool. The connection manager sets the in-use flag to false, and sets the verify and last-used timestamps to the current time.

The connection manager removes idle connections from the pool because they are wasting the significant resources. To decide which connections are idle, the connection manager checks the connection flags and timestamps in a periodic reap of the connection pool:

1. The connection manager looks at the last-used timestamp of the in-use connections. If the time between the last-used and current time exceeds the Maximum Age configuration parameter, the connection is assumed to be an orphan connection, meaning the owning servlet has died or is otherwise unresponsive. The orphan connection is returned to the pool for use by another servlet, its in-use flag is set to false, and its verify and last-used timestamps are set to the current time.
2. The connection manager examines connections not in use by any servlet – those connections whose in-use flags are false. If the time between the last-used and current time exceeds the Maximum Idle Time configuration parameter, the

connection is assumed idle. Idle connections are removed from the pool, down to the lower limit specified by Minimum Connections configuration parameter.

Additional parameters, discussed in the Application Server Manager online help, complete the picture of how the connection manager does its work.

Servlet-connection manager interaction and related APIs

The sample servlet code (“Sample connection manager servlet” on page 38) assumes you can write servlets and are familiar with JDBC APIs. Currently, the only underlying data servers supported by the connection manager are relational database products that support JDBC, or relational database products that can be reached using the JDBC-ODBC bridge. The sample servlet code assumes that the underlying server is one of these.

How servlets use the connection manager

All servlets using the connection manager will follow these steps (the steps are marked in the sample servlet, which you might also want to look at now):

1. Create the connection specification:

   The servlet prepares a specification object identifying information necessary for connecting to the underlying data server. Some of the information is unique to the specific data server, while some is general, applying to any underlying data server. The servlet either prepares the specification only once and uses it for all user requests, or it prepares a new specification for each user request. If a new specification is prepared for each user request, it must be done before step 3, which uses the specification.

   It is worth noting that different specifications can be used to get connections with different properties. For example, a data server may allow access to certain critical data only if the connection was created with a specific user ID. Therefore, the specification must properly identify the user ID in order to get the suitable connection to the data server.

2. Connect to the connection manager:

   The servlet gets a reference to the connection manager in order to communicate with the connection manager. This needs to be done only once in the lifetime of the servlet.

3. Get a connection manager connection:

   The servlet asks the connection manager for a connection to a specific data server using the connection specification prepared in step 1. The connection object returned is from a connection manager pool, and is an instance of a class defined in the connection manager APIs – it is not an object from a class in the API set of the underlying data server. Call this first connection a connection manager connection, or a CM connection for short. Usually, your servlet gets a CM connection for every user request.

4. Use the CM connection to access a pre-established data server connection:

   The servlet invokes a method on the CM connection returned in step 3, retrieving an object defined in the API set of the underlying data server. Call this object a data server connection (or a data connection for short) to distinguish it from the CM connection.

The data connection, unlike the CM connection, is from the underlying data server API set. The data connection is not *created* for the servlet – the servlet instead uses the pre-established data connection by virtue of owning a CM connection pool from the pool.

The data connection will be used for the actual interactions with the data server, using the methods from the underlying data server API set. For example, in the sample servlet code the underlying data server will be a JDBC database and the data connection object will be from the Connection class in the JDBC APIs. The JDBC APIs are found in the java.sql package.

5. Interact with the data server:

   The servlet interacts with the data server – retrieving data, updating data, and so on – using methods of the data connection object. These methods will be specific to the underlying data server, because the data connection will actually be from the API set of the underlying data server. The data connection for a different underlying data server will have different methods.

   Full documentation for the methods will be found with the API documentation for the specific data server product. For example, for a JDBC data connection, you will need to look at the documentation for the java.sql package and at any documentation that comes with the JDBC-enabled relational database you are using.

   Note: If you use the data connection for more than one data server interaction within the same user request, you may want to verify before the additional interactions that your servlet still owns the associated CM connection. The connection manager periodically checks a last-used timestamp to see if your servlet has been using the CM connection, and if not, the connection manager assumes that your servlet has failed or has otherwise become unresponsive. It takes the CM connection away. Verifying the CM connection (using the verifyIBMConnection() method) also updates the last-used timestamp.

6. Release the connection:

   The servlet returns the CM connection to the connection manager pool, freeing the connection for use by another servlet or by another request to the same servlet.

7. Prepare and send the response:

   The servlet prepares and returns the response to the user request. In this step you will probably not be using any connection manager APIs.

Connection manager APIs

Some of the connection manager APIs will now be discussed, to illustrate how they relate to the steps discussed above. The APIs will be covered in the approximate order that they are used in the sample servlet (“Sample connection manager servlet” on page 38). The complete APIs are covered in the connection manager Javadoc. You can find links to the connection manager Javadoc in the documentation on your Application Server machine. Use your browser to:

- For Windows NT, open the file *applicationserver_root*\doc\index.html
- For AIX, open the file *applicationserver_root*/doc/index.html
- For Sun Solaris, open the file *applicationserver_root*/doc/index.html

IBMJdbcConnSpec class

Create a specification object of this class to record the specifications for the connection to the desired data server. This is typically done in step 1 on page 33. Note that the specification object does not actually set the specifications, but is used as an argument by another method, the method that sets the specifications (see the getIBMConnection() method in the IBMConnMgr class below). Currently, the connection manager supports connections only to JDBC compliant data servers, so only a JDBC specification class is available. When other data servers become supported, additional specification classes will be made available. After constructing the specification object, its possible to use get and set methods to specify the connection requirements, but you will usually specify all the requirements during the initial construction of the specification object. The constructor details are:

```
public IBMJdbcConnSpec(String poolName, boolean waitRetry, String
dbDriver, String url, String user, String password)
```

Parameters:

The first two parameters are common to all data servers.

poolName
: The connection manager pool containing the connection type you want. See the Webmaster for the pool name.

waitRetry
: Whether to wait for a connection to free up if the pool does not currently have an available connection. (To wait, specify true). The Webmaster uses the Connection Time Out parameter to set the length of the wait for the entire pool, so if a connection cannot be made available by that time, your request will fail. Specify false for an immediate failure if a connection from the pool is not available.

The remaining parameters are specific to JDBC data servers.

dbDriver
: The driver class provided with a specific JDBC product, or the name of the driver providing the JDBC-ODBC bridge. See the Webmaster or database administrator for the driver name. See the Driver class in the java.sql package for more information.

url A database URL. See the getConnection() method in the DriverManager class in the java.sql package for more information.

user The database user on whose behalf the connection is being made. See the getConnection() method in the DriverManager class in the java.sql package for more information.

password
: The user password. See the getConnection() method in the DriverManager class in the java.sql package for more information.

See the Javadoc for the Application Server APIs for information about the get and set methods for the IBMJdbcConnSpec class.

IBMConnMgrUtil class

Use a method of this class to get a reference to the connection manager. This is typically done in step 2 on page 33. You will use the reference to communicate with the connection manager and use its services. Only one class or static method is of interest:

```
public static IBMConnMgr getIBMConnMgr()
```

Returns:

A reference to the connection manager

IBMConnMgr class

The running instance of the connection manager is an instance of this class. You will get a reference to the connection manager in step 2 on page 33. Your servlet never *creates* an instance of the connection manager, but instead uses a reference to the *existing* instance. For details, see the getIBMConnMgr() method of the IBMConnMgrUtil class, discussed above.

Only one method of this class is of interest – the getIBMConnection() method to get a CM connection from the pool (used in step 3 on page 33).

getIBMConnection():

```
public IBMConnection getIBMConnection(IBMConnSpec connSpec)
throws IBMConnMgrException
```

This method gets a CM connection from the pool for use by your servlet, if such a connection is available. Note, the only parameter passed is a specification object (created in step 1 on page 33) for the connection. If a CM connection is not immediately available, and if waitRetry in the specification object is set to true, the servlet can wait for a CM connection to become available. The length of the wait is set by the Webmaster using the Connection Time Out parameter. The Webmaster can also disable the wait or extend the wait indefinitely. If a CM connection does not become available after the wait period, the getIBMConnection() method will throw an IBMConnMgrException exception. If waitRetry is false, failure to get a CM connection causes the getIBMConnection() method to throw the exception right away.

Note the returned IBMConnection object is a ″general″ connection object and must be cast to a connection object specific to the underlying data server you need to access. For example, in the sample servlet, the returned IBMConnection object is cast to an IBMJdbcConn object. Without the appropriate cast, the connection object cannot use its methods to get to the underlying data server in step 5 on page 34.

Parameters:

connSpec
: An extension of the IBMConnSpec class, containing detailed connection requirements for a specific underlying data server. This will be an IBMJdbcConnSpec object (created in step 1 on page 33) in the sample servlet.

Returns:

An IBMConnection object from the connection manager pool

IBMJdbcConn class

The IBMConnection object retrieved in step 3 on page 33 needs to be cast to a connection object with respect to the underlying data server to be accessed. Otherwise, there is no access to the APIs associated with the underlying data server. Currently, only one such specialized connection class is available – the IBMJdbcConn class for JDBC data server access. Access to the APIs of the JDBC server through a IBMJdbcConn object is typically established in step 4 on page 33.

The IBMJdbcConn class has one method of interest:

```
public Connection getJdbcConnection()
```

Returns:

A Connection object to the underlying JDBC data server

The connection class is from the JDBC API and is documented with the java.sql package. Methods of the Connection class let you interact with the JDBC data server. Elsewhere in this chapter the Connection class is referred to more generically as the data connection, to distinguish it from the CM connection. Recall that the CM connection is not part of the API set of the underlying data server.

IBMConnection class

The IBMJdbcConn class is an extension of the IBMConnection class. Thus, many of the methods in the IBMConnection class are also in instances of the IBMJdbcConn class. Two methods of the IBMConnection class (also in IBMJdbcConn) are of interest – the verifyIBMConnection() method to verify that the CM connection is still valid (optionally used in step 5 on page 34), and the releaseIBMConnection() method to return the CM connection to the pool (used in step 6 on page 34).

verifyIBMConnection():

```
public boolean verifyIBMConnection()
throws IBMConnMgrException
```

The connection manager might take a CM connection away from a servlet if the CM connection has been inactive for a specified length of time. (Check with the Webmaster to determine the behavior configured for the connection manager). If you use your data connection for several interactions within one user request, you may want to invoke the verifyIBMConnection() method before each interaction to check whether your servlet still owns the associated CM connection from the pool. If the servlet still owns the connection, invoking the method will also reset a last-used timestamp.

If you anticipate that all your servlet interactions with the data server (from one user request) will complete within a few seconds, and if the Maximum Age parameter for the connection manager pool is at least several minutes long, then there is probably no need to use the verifyIBMConnection() method — the request will complete long before there is any chance that the connection manager will take away the connection.

Returns:

True if the servlet still owns the CM connection, otherwise false

releaseIBMConnection():

```
public void releaseIBMConnection()
throws IBMConnMgrException
```

When a servlet no longer needs the CM connection object, the servlet uses this method to release the connection back to the pool. This should be done at the end of each user request.

Sample connection manager servlet

The sample servlet should be easy to follow, based on the previous discussions, comments in the code, and the choice of variable names. The sample below follows the steps outlined in the "How servlets use the connection manager" on page 33.

```
// ********************************************************************
// * IBMConnMgrTest.java - test the connection manager                *
// ********************************************************************
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import java.util.*;
import com.ibm.servlet.connmgr.*;

public class IBMConnMgrTest extends HttpServlet
{
   // ********************************************************************
   // * Variables                                                        *
   // ********************************************************************
   // Use to communicate with connection manager.
   static IBMConnMgr connMgr = null;

   // Use later in init() to create JDBC connection specification.
   static IBMConnSpec spec   = null;      // the spec
   static String DbName      = null;      // database name
   static String Db          = "db2";     // JDBC subprotocol for DB2
   static String poolName    = "JdbcDb2"; // from Webmaster
   static String jdbcDriver  = "COM.ibm.db2.jdbc.app.DB2Driver";
   static String url         = null;      // constructed later
   static String user        = null;      // user and password could
   static String password    = null;      // come from HTML form
   static String owner       = null;      // table owner

   // Name of property file used to complete the user, password,
   // DbName, and table owner information at runtime.  ".properties"
   // extension assumed.
   static final String CONFIG_BUNDLE_NAME = "login";

   // ********************************************************************
   // * Initialize servlet when it is first loaded                       *
   // ********************************************************************
   public void init(ServletConfig config)
   throws ServletException
   {
      super.init(config);
      try
      {
         // Get information at runtime (from an external property file
         // identified by CONFIG_BUNDLE_NAME) about the database name
         // and the associated database user and password.  This
         // information could be provided in other ways.  It could be
         // hardcoded within this application, for example.
         PropertyResourceBundle configBundle =
         (PropertyResourceBundle)PropertyResourceBundle.
         getBundle(CONFIG_BUNDLE_NAME);
         DbName   = configBundle.getString("JDBCServlet.dbName");
```

```
      url      = "jdbc:" + Db + ":" + DbName;
      user     = configBundle.getString("JDBCServlet.dbUserid");
      password = configBundle.getString("JDBCServlet.dbPassword");
      owner    = configBundle.getString("JDBCServlet.dbOwner");
   }
   catch(Exception e)
   {
      System.out.println("read properties file: " + e.getMessage());
   }

   try
   {
      // **********
      // * STEP 1 *
      // **********
      // Create JDBC connection specification.
      spec = new IBMJdbcConnSpec
             (poolName,    // pool name from Webmaster
              true,        // waitRetry
              jdbcDriver,  // Remaining four
              url,         // parameters are
              user,        // specific for a
              password);   // JDBC connection.

      // **********
      // * STEP 2 *
      // **********
      // Get a reference to the connection manager.
      connMgr = IBMConnMgrUtil.getIBMConnMgr();
   }
   catch(Exception e)
   {
      System.out.println("set connection spec, get connection manager: " +
                         e.getMessage());
   }
}

// ****************************************************************
// * Respond to user GET request                                  *
// ****************************************************************
public void doGet(HttpServletRequest req, HttpServletResponse res)
{
   IBMJdbcConn cmConn   = null;
   Connection dataConn  = null;
   Vector firstNameList = new Vector();
   try
   {
      // **********
      // * STEP 3 *
      // **********
      // Get an IBMJdbcConn object (cmConn) meeting JDBC
      // connection specs, from the connection manager pool.
      cmConn = (IBMJdbcConn)connMgr.getIBMConnection(spec);

      // **********
      // * STEP 4 *
      // **********
      // Get a Connection object (dataConn).  This is
      // an object from the java.sql package and it is used
      // for JDBC access.
      dataConn = cmConn.getJdbcConnection();

      // **********
      // * STEP 5 *
      // **********
      // Run DB query - create a Vector of first names of all people
      // whose last name is 'PARKER'.
```

```
   // Standard JDBC coding follows.  Change the query for your
   // specific situation.
   Statement stmt = dataConn.createStatement();
   String query   = "Select FirstNme " +
                    "from " + owner + ".Employee " +
                    "where LASTNAME = 'PARKER'";
   ResultSet rs   = stmt.executeQuery(query);
   while(rs.next())
   {
      firstNameList.addElement(rs.getString(1));
   }
   // Invoke close() on stmt, which also closes rs, freeing
   // resources and completing the interaction.  You must
   // not, however, close the dataConn object.  It must
   // remain open and under the control of connection manager
   // for possible use by other requests to this servlet or
   // to other servlets.
   stmt.close();
   // If you now want to use the dataConn object again for
   // another interaction, you may want to make sure that you
   // still own the associated cmConn object.  Use the
   // verifyIBMConnection() method to do this, which also
   // updates a last-used timestamp.  If the
   // verifyIBMConnection() method fails, you will need
   // to get new instances of the cmConn and dataConn objects
   // before you can use them.  The actual need to use
   // the verifyIBMConnection() method depends on the
   // Maximum Age parameter set for the connection manager
   // and the length of time that you anticipate your
   // servlet will need to use the cmConn and dataConn
   // objects.
}
catch(Exception e)
{
   System.out.println("get connection, process statement: " +
                      e.getMessage());
}
// **********
// * STEP 6 *
// **********
// Release the connection back to the pool.
finally
{
   if(cmConn != null)
   {
      try
      {
         cmConn.releaseIBMConnection();
      }
      catch(IBMConnMgrException e)
      {
         System.out.println("release connection: " + e.getMessage());
      }
   }
}
// **********
// * STEP 7 *
// **********
// Prepare and return HTML response - say Hello to everyone
// whose last name is 'Parker' and address them with their first
// and last name.
res.setContentType("text/html");
// Next three lines prevent dynamic content from being cached
// on browsers.
res.setHeader("Pragma", "no-cache");
res.setHeader("Cache-Control", "no-cache");
res.setDateHeader("Expires", 0);
```

```
      try
      {
         ServletOutputStream out = res.getOutputStream();
         out.println("<HTML>");
         out.println("<HEAD><TITLE>Hello DBWorld</TITLE><HEAD>");
         out.println("<BODY>");
         if(firstNameList.isEmpty())
         {
            out.println("<H1>Nobody named Parker</H1>");
         }
         else
         {
            for(int i = 0; i < firstNameList.size(); i++)
            {
               out.println("<H1>Hello " +
                           firstNameList.elementAt(i) +
                           " Parker</H1>");
            }
         }
         out.println("</BODY></HTML>");
         out.close();
      }
      catch(IOException e)
      {
         System.out.println("HTML response: " + e.getMessage());
      }
   }
}
```

How the sample servlet uses the connection manager

The sample servlet performs the following steps during its interactions with the connection manager. The steps correspond to the sequence of servlet-connection manager interactions introduced in “How servlets use the connection manager” on page 33.

1. Create the connection specification:

   The connection specification named spec is created in the init() method when the servlet is first loaded. In the sample servlet, information needed to create the spec object is read into the servlet earlier in the init() method. This information comes from an external Java properties file named login.properties. Alternatively, you could hardcode the information into your servlet — the user, password, and other related information.

   It is also possible to create a new specification for each user request. This would normally be done just before step 3 in the sample servlet. In this way, the spec object could be created based on information submitted with the user request.

2. Connect to the connection manager:

   The servlet establishes communications with the connection manager just once in the servlet’s lifecycle – when the servlet is loaded. This is conveniently done in the servlet’s init() method, which is provided for such initialization tasks. Use the static method getIBMConnMgr() of the IBMConnMgrUtil class to establish the communications. Note that the single instance of the connection manager, namely connMgr, is used by all user requests to the sample servlet.

3. Get a connection manager connection:

   Use the connection manager connMgr to get the CM connection cmConn from the pool. Note that cmConn is declared within the doGet() method, so that every user request gets its own CM connection. Also note that cmConn is cast

to the IBMJdbcConn class so that cmConn will have methods available to it to get at the APIs of the underlying data server.

4. Get a data server connection:

   Use the CM connection cmConn to get the data connection dataConn. Note that dataConn is a Connection object from the API set of the underlying data server. You will be using APIs of the underlying data server, rather than connection manager APIs, for interactions with the data server.

   Note: cmConn and dataConn are declared within the doGet() method, giving each user request its own copies of the variables. If instead cmConn and dataConn had been declared outside of all the methods of the servlet, all user requests would use the same copy of each connection. In the latter case, concurrent user requests would mix their data server interactions on the same data connection, with unpredictable results.

5. Interact with the data server:

   Use the dataConn object from the Connection class in the JDBC APIs to communicate with DB2. The Statement object stmt and ResultSet object rs also belong to the JDBC APIs, documented in the Javadoc for the java.sql package. It is worthwhile to reiterate that access to the underlying data server is achieved using the APIs provided with that data server. Data servers may differ in their API sets. Regardless, the data server API sets are not part of the connection manager APIs.

   Note also that no dataConn.close() statement appears anywhere in the servlet. Management of the connection, such as connecting to and disconnecting from the underlying data server, must be handled by the connection manager.

6. Release the connection:

   Release the connection as soon as your servlet is through interacting with the data server. The connection will return to the pool where it can be used again. The connection manager is probably configured to take a connection away from a servlet if the connection has been idle for a period of time, thus reclaiming unreleased connections. Until that time period expires, however, the connection resource is unavailable for use.

   Notice how releasing the connection is done in a finally block. This ensures that the connection is released if the previous try block completes normally, or if the previous try block fails for any reason.

7. Prepare and send the response:

   Prepare and send the response to the user, based on information retrieved in step 5.

Running the sample servlet

The Application Server includes the source file and the compiled class file for the IBMConnMgrTest servlet. See “Chapter 10. Running sample servlets and applications” on page 93 for details on where to find the source file and how to run the compiled servlet. If you want to make changes to the source file and test the resulting compiled servlet, be sure to follow the guidelines in “Chapter 7. Placing servlets on your Application Server” on page 77. Since you will be compiling a servlet that accesses IBM DB2, you need to have a proper reference to the DB2 Java APIs (db2java.zip) in your CLASSPATH. The Application Server Java Classpath must also be set to properly reference the DB2 Java APIs (db2java.zip) in order to run the compiled servlet. See the Webmaster to ensure the proper setup of the Application Server Java Classpath.

Coding considerations

Custom pools

Review with the Webmaster the needs of your servlet and the connection manager configuration options. Even if there is only one underlying data server, consider creating several connection pools for the data server in order to meet the performance and resource access needs of several classes of servlets.

Custom error handling

You may find it useful to modify the error handing code of your servlet — you can design the error messages to assist you and the Webmaster in tuning a connection manager pool for a specific class of servlets. Assume, for example, that due to many requests coming in over the Internet, a connection manager pool contains the maximum number of connections set by its Maximum Connections parameter.

If your servlet waitRetry parameter is set to true, your servlet request for a connection from the pool will fail if the servlet waits for a free connection for a time that exceeds the Connection Time Out parameter set in the connection manager pool. This behavior was discussed earlier in this chapter. If your servlet waitRetry parameter is set to false, your servlet request for a connection from the pool will fail right away. This behavior was also discussed earlier in this chapter.

You can modify your servlet to more clearly identify that these events are happening, and this can help in tuning the connection manager pool to minimize the failures. Initially, a catch block in your servlet might look something like:

```
catch(Exception e)
 {
   System.out.println(e.getMessage());
 }
```

You would begin by modifying the catch block to invoke the errorHandler() method:

```
catch(Exception e)
  {
    System.out.println(errorHandler(e));
  }
```

The errorHandler() method is a new method that you add to your servlet. The errorHandler() method in turn calls the connMgrExceptionHandler() method, which you also add to your servlet. The two methods are shown in the code below, which you can modify to meet your specific needs:

```
protected String errorHandler(Exception e)
  {
     String retString = null;
     String className = e.getClass().getName();
     if(className.equals(IBMConnMgrConstants.ExceptionClassName))
     {
        retString  = connMgrExceptionHandler((IBMConnMgrException)e);
     }
     else
     {
        retString = e.getMessage();
     }
     return retString;
  }
```

```
protected String connMgrExceptionHandler(IBMConnMgrException e)
{
   StringBuffer retString = new StringBuffer(e.getMessage());
   String reason = e.getReason();
   if(reason.equals(IBMConnMgrConstants.ConnectionTimeOut))
   {
      retString.append(" The Connection wait timed out.  Consult ");
      retString.append("with your Webmaster to tune the ");
      retString.append("Connection Time Out pool parameter.");
   }
   else if(reason.equals(IBMConnMgrConstants.MaxConnsExceeded))
   {
      retString.append(" The maximum number of connections in the ");
      retString.append("pool has been reached.  Consult with your ");
      retString.append("Webmaster to tune the Maximum ");
      retString.append("Connections pool parameter.");
   }
   return retString.toString();
}
```

Look at the IBMConnMgrConstants class in the connection manager Javadoc to get an idea of other exceptions you can write special code for to assist in debugging a servlet or tuning a connection manager pool for a servlet.

You may also want to look at the Javadoc for the IBMConnMgr class — the class has methods that you may want to use to monitor the behavior of a connection pool, but the methods will not be discussed further here.

Security

In the sample servlet, note that the connection specification information is read from data hardcoded in a login.properties file, and that the same connection specification is used for all user requests. It is also possible for a new connection specification to be created for each user request, based on information passed in with the user request. This might involve, for example, passing user and password information over the Internet, with obvious security issues. Hardcoding user and password information into files located on the Web server machine is clearly more secure.

Alternatively, you can specify initialization parameters to be read in by the servlet when the servlet is loaded. See “Chapter 2. Using the Application Server Manager” on page 9. The initialization parameters can be read in the servlet’s init() method using the getInitParameterNames() and getInitParameter() methods. This information can then be used to create a connection specification, allowing servlets to be more generic. User, password, database name information, as well as other information, is specified only when the servlet is loaded, without the need to recompile the servlet for each change.

Using the verifyIBMConnection() method

If your servlet uses a connection for multiple interactions with the data server, for each user request, taking place over a period of time, you may want to use the verifyIBMConnection() method before each additional interaction to make sure that your servlet still owns the connection. The reasons for this were discussed earlier in this chapter.

As an example of how this might be coded, consider step 5 of the sample servlet (“Sample connection manager servlet” on page 38). Just after the interaction

completed with stmt.close(), you want to begin another interaction with the data server. The code fragment below (to be added after stmt.close() in step 5 of the sample servlet) shows how you might do this.

```
try
{
   if(!cmConn.verifyIBMConnection())
   {
      // Get valid cmConn and dataConn objects if cmConn
      // has "timed-out".
      cmConn = (IBMJdbcConn)connMgr.getIBMConection(spec);
      dataConn = cmConn.getJdbcConnection();
   }
}
catch(IBMConnMgrException e)
{
   System.out.println(e.getMessage());
}
// Now use dataConn for another query, update, etc, using
// cmConn and dataConn objects known to be good because of
// the verify.
// INSERT INTERACTION CODE HERE
// Complete the interaction.
stmt.close();
```

Remember that using the verifyIBMConnection() method is warranted only if your servlet has multiple interactions with the data server and if it is likely to take a length of time to respond to the request that is comparable to the Maximum Age parameter of the connnection pool.

Monitoring the connection manager

The Application Server Manager provides a monitor for the connection manager — it is named ″DB Pool Connections″ in the monitor list in the Application Server Manager. See “Chapter 2. Using the Application Server Manager” on page 9 for information on how to start the Application Server Manager and on how to display the connection manager monitor. The connection manager monitor online help gives full details on what information the monitor provides. You can use the information to see how connection pools are performing and to suggest possible changes in the connection pool parameters. You can monitor a pool after you change the parameters in order to see the change in pool behavior and to assist you in further tuning of the pool.

You can pick the specific pool you want to monitor from a selection list. The resulting monitor display consists of two parts:

1. The top part lists each connection in the pool that you have selected.
2. The bottom part lists summary information about the pool as a whole.

Information provided by the connection list (top part of the monitor) includes:

- Whether the connection is in use by a servlet.
- The servlet class that currently owns the connection (or the servlet class that last released the connection if it is free).
- The verify and last used timestamps for the connection.
- Additional information, some of which may depend on the type of data server.

Information provided by the pool summary (bottom part of the monitor) includes:

- Pool Name:

This is the unique name of a pool, used by the servlet programmer to access the pool and used within the monitor to identify a pool and the statistics associated with it.

- Total connections:

  This is a cumulative total of only the successful requests for connections made to the pool.

- Requests:

  This is a cumulative total of both the successful and the unsuccessful requests for connections made to the pool.

- Waiting:

  This is the number of connection requests currently waiting for a connection from the specified pool. When a servlet's waitRetry parameter is set to true, the servlet can wait for a short time for a connection if the pool does not have one immediately available. Requests waiting for a connection will fail if the connection is not made available within the time specified by the Connection Time Out parameter. See the Rejected statistic below for the number of requests that actually fail. If the servlets using this pool are important, you may want to increase the Maximum Connections parameter for the pool to reduce the chances that a connection request will have to wait.

- Rejected:

  This is the number of connection requests that were refused a connection. It is the cumulative total of the Waiting requests that failed to get a connection, plus failed requests from servlets whose waitRetry parameter was set to false and thus failed right away when a connection was not available. Generally, you will want Rejected to be a small compared to Total connections, particularly if important servlets are using the pool. Look at increasing the Maximum Connections and the Connection Time Out parameters of the connection pool to keep Rejected a small percentage of Total connections.

- Orphaned:

  This is the number of connections taken away from servlets that have died or otherwise become unresponsive, or taken away from servlets that may have been coded improperly. The connections are taken away from the servlets in the periodic reap process and are returned to the pool so that other servlets can use them. Something is probably wrong if the Orphaned statistic is other than zero, and you should look at making servlet coding changes. If the value is other than zero, it may mean that connections are not being used and yet the connections are not available to be used by a new request. This can happen if a servlet fails without explicitly releasing connections that it owns, or if the servlet performs normally but the programmer neglects to explicitly release the connection after sending the response back to the user — the servlet should probably should be changed to use the releaseIBMConnection() method in case of a servlet failure and at the the end of a successful response. The Orphaned statistic can also be other than zero if a servlet neglects to periodically verify with the connection manager that it still needs to hold a connection over an extended period of time (see the discussion earlier in this chapter on using the verifyIBMConnection() method).

- Idled:

  This is the cumulative number of connections that the periodic reap process has removed from the pool (and disconnected from the data server). The connections may be removed after they remain idle (unassigned to any servlets) beyond a certain time. If the Inactive statistic is high compared to Total connections, it may mean that there is a lot of connect/disconnect overhead compared to the number of requests actually serviced. This in turn might be due to excessive

fluctuations in the number of user requests — connections are created to satisfy a temporary peak, and then discarded during a temporary lull. You might want to make the pool less sensitive to such fluctuations by increasing the Reap Time or Maximum Idle Time pool parameters.

Review the connection manager monitor online help for other statistics that may be of use in managing and tuning the connection pools and the servlets that use the pools.

Chapter 5. Using data access JavaBeans

Java applications (and servlets) that access JDBC compliant relational databases typically use the classes and methods in the java.sql package to access the data. This is shown in the sample servlet discussed in “Chapter 4. Using the connection manager” on page 29. Instead of using the java.sql package, you can use the classes and methods in the IBM data access JavaBeans.

As their name implies, the data access JavaBeans are Java classes coded to the JavaBeans specifications. They provide a rich set of features and enhanced function over what is provided in the java.sql package, yet they manage to hide much of the complexity associated with accessing relational databases.

The data access classes, because they are JavaBeans, can be used in integrated development environments (IDEs) such as the IBM product VisualAge for Java. This allows the programmer to manipulate Java classes in a visual way, rather than editing lines of Java code. Because JavaBeans are also Java classes, programmers can use them like ordinary classes when writing Java code.

It is the second approach that will be discussed in this chapter – using the data access JavaBeans like ordinary Java classes in writing a program. The discussion will cover:

- Features of the data access JavaBeans – why you might want to use them.
- A sample servlet using the JavaBeans – how to use them.

The sample servlet uses the connection manager to provide the connection between the JDBC database and the data access JavaBeans. (See “Chapter 4. Using the connection manager” on page 29). For a brief summary:

- The connection manager maintains a pool of existing connections to JDBC compliant relational databases.
- The main advantage of the connection manager is that a servlet can use an existing connection from the connection manager pool to respond to a user request arriving over the Internet (when the user request requires access to a relational database). The servlet is more responsive because it does not incur the delay and overhead of getting and later discarding a new connection for each new user request.
- Another advantage of the connection manager is its ability to help control resource usage of the database server despite the volatile nature of user requests on the Internet.

Features

The data access JavaBeans offer:

- **Caching of query results:** SQL query results can be retrieved all at once and placed into a cache. The application (or servlet) can move forward and backward through the cache or jump directly to any result row in the cache. Contrast this to facilities in the java.sql package, in which rows are retrieved from the database one at a time, in the forward direction only, and a newly retrieved row overlays the last retrieved row unless additional code is written to

expand functionality. For large result sets, the data access JavaBeans provide ways to retrieve and manage *packets*, subsets of the complete result set.

- **Updates through result cache:** The servlet can use standard, familiar Java statements (rather than SQL statements) to change, add, or delete rows in the result cache. Changes to the cache can be propagated to the underlying relational table.
- **Query parameter support:** The base SQL query is defined as a Java String, with parameters replacing some of the actual values. When the query is run, the data access JavaBeans provide a way to replace the parameters with values made available at runtime. For example, the user might enter the SQL query parameters into an HTML form submitted from a browser.
- **Metadata support:** A StatementMetaData object is used to contain the base SQL query. Higher levels of data (*metadata*) can be added to the object to help pass parameters into the query and work with the returned results. Query parameters can be Java datatypes convenient for the Java program. When the query is run, the parameters are automatically converted using metadata specifications into forms suitable for the SQL datatype. For reading the query results, the metadata specifications can automatically convert the SQL datatype into the Java datatype most convenient for the Java application.

For additional information, see the links to the data access JavaBeans Javadoc in the documentation on your Application Server machine. Use your browser to:

- For Windows NT, open the file *applicationserver_root*\doc\index.html
- For AIX, open the file *applicationserver_root*/doc/index.html
- For Sun Solaris, open the file *applicationserver_root*/doc/index.html

Sample servlet

The sample servlet discussed in this chapter uses the data access JavaBeans and is based on a prior sample servlet discussed in “Chapter 4. Using the connection manager” on page 29. The prior sample servlet uses the connection manager to get a connection to a JDBC relational database. It then uses classes from the java.sql package to interact with the database, using the connection.

The sample servlet in this chapter uses the data access JavaBeans, rather than the classes in the java.sql package, to interact with the database. For convenience, call the sample servlet in this chapter the DA (for data access JavaBeans) sample servlet, and call the sample servlet on which it is based the CM (for connection manager) sample servlet.

The DA sample servlet benefits from the performance and resource management enhancements made possible by the connection manager. In addition, the programmer coding the DA sample servlet benefits from the additional features and functions provided by the data access JavaBeans.

The DA sample servlet differs from the CM sample servlet in only a few parts. The discussion the DA sample servlet covers only the changes – review “Chapter 4. Using the connection manager” on page 29 to refresh your understanding of the CM sample servlet. The DA sample servlet shows the basics of the connection manager and the data access JavaBeans, but keeps other code to a minimum, so the servlet is not entirely realistic. You are expected to be familiar with basic servlet and JDBC coding

The DA sample servlet IBMDataAccessTest.java follows. An overview and detailed discussions follow the sample code.

```
// ********************************************************************
// * IBMDataAccessTest.java - test the data access JavaBeans          *
// ********************************************************************
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import java.util.*;
import java.math.BigDecimal;
import com.ibm.db.*;
import com.ibm.servlet.connmgr.*;

public class IBMDataAccessTest extends HttpServlet
{
   // ********************************************************************
   // * Variables                                                        *
   // ********************************************************************
   // Use to communicate with connection manager.
   static IBMConnMgr connMgr = null;

   // Use later in init() to create JDBC connection specification.
   static IBMConnSpec spec   = null;      // the spec
   static String DbName      = null;      // database name
   static String Db          = "db2";     // JDBC subprotocol for DB2
   static String poolName    = "JdbcDb2"; // from Webmaster
   static String jdbcDriver  = "COM.ibm.db2.jdbc.app.DB2Driver";
   static String url         = null;      // constructed later
   static String user        = null;      // user and password could
   static String password    = null;      // come from HTML form
   static String owner       = null;      // table owner

   // Name of property file used to complete the user, password,
   // DbName, and table owner information at runtime.  ".properties"
   // extension assumed.
   static final String CONFIG_BUNDLE_NAME = "login";

   // Single metaData object, used by all user requests, will be
   // fully defined in the init() method when the servlet is loaded.
   static StatementMetaData metaData = null;

   // ********************************************************************
   // * Initialize servlet when it is first loaded                       *
   // ********************************************************************
   public void init(ServletConfig config)
   throws ServletException
   {
      super.init(config);
      try
      {
         // Get information at runtime (from an external property file
         // identified by CONFIG_BUNDLE_NAME) about the database name
         // and the associated database user and password and table owner.
         // This information could be provided in other ways.  It could be
         // hardcoded within this application, for example.
         PropertyResourceBundle configBundle =
         (PropertyResourceBundle)PropertyResourceBundle.
         getBundle(CONFIG_BUNDLE_NAME);
         DbName   = configBundle.getString("JDBCServlet.dbName");
         url      = "jdbc:" + Db + ":" + DbName;
         user     = configBundle.getString("JDBCServlet.dbUserid");
         password = configBundle.getString("JDBCServlet.dbPassword");
         owner    = configBundle.getString("JDBCServlet.dbOwner");
      }
      catch(Exception e)
```

```
    {
       System.out.println("read properties file: " + e.getMessage());
    }

    try
    {
       // **********
       // * STEP 1 *
       // **********
       // Create JDBC connection specification.
       spec = new IBMJdbcConnSpec
              (poolName,    // pool name from Webmaster
               true,        // waitRetry
               jdbcDriver,  // Remaining four
               url,         // parameters are
               user,        // specific for a
               password);   // JDBC connection.

       // **********
       // * STEP 2 *
       // **********
       // Get a reference to the connection manager.
       connMgr = IBMConnMgrUtil.getIBMConnMgr();
    }
    catch(Exception e)
    {
       System.out.println("set connection spec, get connection manager: " +
                          e.getMessage());
    }

    // Add data access JavaBeans code.
    // Query string, with :idParm and :deptParm parameters.
    String sqlQuery = "SELECT ID, NAME, DEPT, COMM " +
                      "FROM " + owner + ".STAFF " +
                      "WHERE ID >= ? " +
                      "AND DEPT =  ? " +
                      "ORDER BY ID ASC";
    // Start defining the metaData object based on the query string.
    metaData = new StatementMetaData();
    metaData.setSQL(sqlQuery);
    try
    {
       // Add some more information to the metaData to make Java
       // programming more convenient.  The addParameter() method allows
       // us to supply an input parameter for the query using a
       // convenient Java datatype, doing a conversion to the datatype
       // actually needed by SQL.  The addColumn() method makes things
       // convenient in the other direction, retrieving data in a
       // datatype convenient for Java programming, doing a conversion
       // from the underlying SQL datatype.  The addTable() method
       // identifies the relational table and makes it possible for
       // result cache changes to be folded back onto the table.
       metaData.addParameter("idParm",   Integer.class,    Types.SMALLINT);
       metaData.addParameter("deptParm", String.class,     Types.SMALLINT);
       metaData.addColumn("ID",          String.class,     Types.SMALLINT);
       metaData.addColumn("NAME",        String.class,     Types.VARCHAR);
       metaData.addColumn("DEPT",        Integer.class,    Types.SMALLINT);
       metaData.addColumn("COMM",        BigDecimal.class, Types.DECIMAL);
       metaData.addTable("STAFF");
    }
    catch(DataException e)
    {
       System.out.println("set metadata: " + e.getMessage());
    }
}

// *************************************************************
```

```
// * Respond to user GET request                                    *
// ******************************************************************
public void doGet(HttpServletRequest req, HttpServletResponse res)
{
   IBMJdbcConn cmConn   = null;
   Connection dataConn  = null;
   SelectResult result  = null;
   try
   {
      // **********
      // * STEP 3 *
      // **********
      // Get an IBMJdbcConn object (cmConn) meeting JDBC
      // connection specs, from the connection manager pool
      cmConn = (IBMJdbcConn)connMgr.getIBMConnection(spec);

      // **********
      // * STEP 4 *
      // **********
      // Get a Connection object (dataConn).  This is
      // an object from the java.sql package and it is used
      // for JDBC access.
      dataConn = cmConn.getJdbcConnection();

      // **********
      // * STEP 5 *
      // **********
      // Make use of the externally managed connection dataConn gotten
      // through the connection manager.  Our dataAccessConn object
      // should be local (a new object for each request), since there
      // may be multiple concurrent requests.
      DatabaseConnection dataAccessConn =
      new DatabaseConnection(dataConn);
      // Begin building our SQL select statement - it also needs to be
      // local because of concurrent user requests.
      SelectStatement selectStatement = new SelectStatement();
      selectStatement.setConnection(dataAccessConn);
      // Attach a metadata object (which includes the actual SQL
      // select in the variable sqlQuery) to our select statement.
      selectStatement.setMetaData(metaData);
      // Make use of the facilities provided through the metadata
      // object to set the values of our parameters, and then execute
      // the query.  Values for dept and id are usually not hardcoded,
      // but are provided through the user request.
      String  wantThisDept  = "42";
      Integer wantThisId    = new Integer(100);
      selectStatement.setParameter("deptParm", wantThisDept);
      selectStatement.setParameter("idParm",   wantThisId);
      selectStatement.execute();
      // The result object is our cache of results.
      result = selectStatement.getResult();
      // Try an update on the first result row.  Add 12.34 to the
      // existing commission, checking first for a null commission.
      BigDecimal comm = (BigDecimal)result.getColumnValue("COMM");
      if(comm == null)
      {
         comm = new BigDecimal("0.00");
      }
      comm = comm.add(new BigDecimal("12.34"));
      result.setColumnValue("COMM", comm);
      result.updateRow();
      // Close the result object - no more links to the relational
      // data, but we can still access the result cache for local
      // operations, as shown in STEP 7 below.
      result.close();
   }
   catch(Exception e)
```

```
{
   System.out.println("get connection, process statement: " +
                      e.getMessage());
}

// **********
// * STEP 6 *
// **********
// Release the connection back to the pool.
finally
{
   if(cmConn != null)
   {
      try
      {
         cmConn.releaseIBMConnection();
      }
      catch(IBMConnMgrException e)
      {
         System.out.println("release connection: " + e.getMessage());
      }
   }
}
// **********
// * STEP 7 *
// **********
// Prepare and return HTML response
res.setContentType("text/html");
// Next three lines prevent dynamic content from being cached
// on browsers.
res.setHeader("Pragma", "no-cache");
res.setHeader("Cache-Control", "no-cache");
res.setDateHeader("Expires", 0);
try
{
   ServletOutputStream out = res.getOutputStream();
   out.println("<HTML>");
   out.println("<HEAD><TITLE>Hello DBWorld</TITLE><HEAD>");
   out.println("<BODY>");
   out.println("<TABLE BORDER>");
   // Note the use of the result cache below.  We can jump to
   // different rows.  We also take advantage of metadata
   // information to retrieve data in the Java datatypes that
   // we want.
   result.nextRow();
   out.println("<TR>");
   out.println("<TD>" + (String)result.getColumnValue("ID") + "</TD>");
   out.println("<TD>" + (String)result.getColumnValue("NAME") + "</TD>");
   out.println("<TD>" + (Integer)result.getColumnValue("DEPT") + "</TD>");
   out.println("<TD>" + (BigDecimal)result.getColumnValue("COMM") + "</TD>");
   out.println("</TR>");
   result.previousRow();
   out.println("<TR>");
   out.println("<TD>" + (String)result.getColumnValue("ID") + "</TD>");
   out.println("<TD>" + (String)result.getColumnValue("NAME") + "</TD>");
   out.println("<TD>" + (Integer)result.getColumnValue("DEPT") + "</TD>");
   out.println("<TD>" + (BigDecimal)result.getColumnValue("COMM") + "</TD>");
   out.println("</TR>");
   out.println("</TABLE>");
   out.println("</BODY></HTML>");
   out.close();
}
catch(Exception e)
{
```

```
            System.out.println("HTML response: " + e.getMessage());
        }
    }
}
```

Overview

Steps 1, 2, 3, 4, and 6 of the CM sample servlet are unchanged in the DA sample servlet – the steps involve connection manager setup, getting a connection from the pool, and releasing the connection back to the pool. The main changes to the DA sample servlet are:

- A metaData variable has been added to the Variables section at the start of the code. The metaData variable is a StatementMetaData data access JavaBean.
- The init() method of the DA sample servlet has been modified slightly (with respect to the CM sample servlet). New code has been appended to the init() method to do a one-time initialization on the metaData object when the servlet is first loaded.
- Step 5 of the DA sample servlet has been entirely rewritten (with respect to the CM sample servlet) to use the data access JavaBeans to do the SQL query, rather than use classes in the java.sql package. The query is run using the selectStatement object, which is a SelectStatement data access JavaBean.
- Step 7 of the DA sample servlet has been entirely rewritten (with respect to the CM sample servlet) to use the query result cache retrieved in Step 5 to prepare a response to the user. The query result cache is a SelectResult data access JavaBean.

Details

Refer back to the DA sample servlet code IBMDataAccessTest.java while reading this section. The following discussion covers only the *changes* made to the DA sample servlet, with respect to the CM sample servlet, to convert it to use the data access JavaBeans.

The metaData variable: This variable is declared in the Variables section at the start of the code, outside of all methods. This allows a single instance to be used by all incoming user requests. The specification of the variable is completed in the init() method.

The init() method: New code has been added to the init() method. The new code begins by creating the base query object sqlQuery as a String object. Note the :idParm and :deptParm parameter placeholders. The sqlQuery object is used to specify the base query within the metaData object. Finally, the metaData object is provided higher levels of data (metadata), in addition to the base query, that will help with running the query and working with the results. The code shows that:

- The addParameter() method notes that when running the query, the parameter idParm will be supplied as a Java Integer datatype, for the convenience of the servlet , but that idParm should be converted (through the metaData object) to do a query on the SMALLINT relational datatype of the underlying relational data when running the query.

  A similar use of the addParameter() method for the deptParm parameter notes that for the same underlying SMALLINT relational datatype, the second parameter will exist as a different Java datatype within the servlet – as a String rather than as an Integer. Thus parameters can be Java datatypes convenient for

the Java application, and can automatically be converted by the metaData object to be consistent with the required relational datatype when the query is run.

- The addColumn() method performs a function somewhat similar to the addParameter() method. For each column of data to be retrieved from the relational table, the addColumn() method maps a relational datatype to the Java datatype most convenient for use within the Java application. The mapping is used when reading data out of the result cache and when making changes to the cache (and then to the underlying relational table).
- The addTable() method explicitly specifies the underlying relational table. This information is needed if changes to the result cache are to be propagated to the underlying relational table.

Step 5: This step is part of the process of responding to the user request. When steps 1 through 4 have already run, the dataConn Connection object from the connection manager is available for use. The code shows:

1. The dataAccessConn object (a DatabaseConnection JavaBean) is created to establish the link between the data access JavaBeans and the database connection – the dataConn object.
2. The selectStatement object (a SelectStatement JavaBean) is created, pointing to the database connection through the dataAccessConn object, and pointing to the query through the metaData object.
3. The query is ″completed″ by specifying the parameters using the setParameter() method.
4. The query is executed using the execute() method.
5. The result object (a SelectResult JavaBean) is a cache containing the results of the query, created using the getResult() method.
6. The data access JavaBeans offer a rich set of features for working with the result cache – at this point the code shows how the first row of the result cache (and the underlying relational table) can be updated using standard Java coding, without the need for SQL syntax.
7. The close() method breaks the link between the result cache and the underlying relational table, but the data in the result cache is still available for local access within the servlet. After the close(), the database connection is unnecessary. Step 6 (which is unchanged from the CM sample servlet) releases the database connection back to the connection manager pool for use by another user request.

Step 7: The result cache, no longer linked to the underlying relational table, can still be accessed for local processing. In this step, the response is prepared and sent back to the user. The code shows the following:

- The nextRow() and previousRow() methods are used to navigate through the result cache. Additional navigation methods are available.
- The getColumnValue() method is used to retrieve data from the result cache. Because of properties set earlier in creating the metaData object, the data can be easily cast to formats convenient for the needs of the servlet.

A possible simplification: If you do not need to update the relational table, you can simplify the sample servlet:

- At the end of the init() method, you can drop the lines with the addColumn() and addTable() methods, since the metaData object does not need to know as much if there are no relational table updates.

- You will also need to drop the lines with the setColumnValue() and updateRow() methods at the end of step 5, since you are no longer updating the relational table.
- Finally, you can remove most of the type casts associated with the getColumnValue() methods in step 7. You will, however, need to change the type cast to (Short) for the ″ID″ and ″DEPT″ use of the getColumnValue() method.

Running the sample servlet

The Application Server includes the source file and the compiled class file for the IBMDataAccessTest servlet. See “Chapter 10. Running sample servlets and applications” on page 93 for details on where to find the source file and how to run the compiled servlet. If you want to make changes to the source file and test the resulting compiled servlet, be sure to follow the guidelines in “Chapter 7. Placing servlets on your Application Server” on page 77. Since you will be compiling a servlet that accesses IBM DB2, you need to have a proper reference to the DB2 JDBC drivers (db2java.zip) in your CLASSPATH. The Application Server Java Classpath must also be set to properly reference the DB2 JDBC drivers (db2java.zip) in order to run the compiled servlet. See the Webmaster to ensure the proper setup of the Application Server Java Classpath.

Chapter 6. Developing dynamic Web pages

This section describes how to use JavaServer** Pages, HTML templates for variable data, site-wide bulletins, and messaging to develop dynamic Web pages.

Developing JavaServer Pages (JSP)

The Application Server supports a powerful, new approach to dynamic page content: JavaServer Pages (JSP). The JSP function in the Application Server is release-level code that is based on an early version of the Sun Microsystems JSP Specification.

JSP is an easy-to-use solution for generating HTML pages with dynamic content. A JSP file contains combinations of HTML tags, NCSA tags (special tags that were the first method of implementing server-side includes), <SERVLET> tags, and JSP syntax. JSP files have the extension .jsp. The DisplayData.jsp file in "A sample JSP file" on page 66 provides an example.

One of the many advantages of JSP is that it enables you to effectively separate the HTML coding from the business logic in your Web pages. Use JSP to access reusable components, such as servlets, JavaBeans, and Java-based Web applications. JSP also supports embedding inline Java code within Web pages.

HTML tags

JSP supports all valid HTML tags. Refer to your favorite HTML reference for a description of those tags.

<SERVLET> tags

Using the <SERVLET> tag is one method for embedding a servlet within a JSP file. The tag syntax is:

```
<SERVLET NAME="servlet_name" CODE="servlet_class_name" CODEBASE="URL_for_remote_loading"
   initparm1="initparm_value">
<PARAM NAME="parm_name" VALUE="param_value">
</SERVLET>
```

See "Specifying a servlet within the <SERVLET> tag" on page 82 for details.

NCSA tags

You might have legacy SHTML files that contain NCSA tags (instead of servlets) for server-side includes. If the Application Server supports the NCSA tags in your SHTML files, you can convert the SHTML files to JSP files and retain the NCSA tags. The Application Server supports the following NCSA tags through JSP:

- config
- echo var=*variable*

  See "Appendix B. Supported NCSA tags" on page 103 for a list of the supported variables.
- exec

- filesize
- include
- lastmodified
- Commands for formatting size and date outputs

JSP syntax

JSP syntax consists of these formats, as described in detail in the sections that follow:

- Directives, enclosed within <%@ and %>
- Declarations, enclosed within <SCRIPT> and </SCRIPT>
- Scriptlets, enclosed within <% and %>
- Expressions, enclosed within <%= and %>
- <BEAN> tags
- IBM extensions to JSP. See "Developing HTML templates for variable data" on page 69.

Specifying directives

Use JSP directives to specify:

- The scripting language being used
- The interfaces a servlet implements
- The classes a servlet extends
- The packages a servlet imports

The general syntax of the JSP directive is:

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

where the valid directive names are:

language
The scripting language used in the file. At this time, the only valid value and the default value is `java`, for the Java programming language. The scope of this directive spans the entire file. When used more than once, only the first occurrence of the directive is significant. An example:

```
<%@ language ="java" %>
```

method
The name of the method generated by the embedded Java code (scriptlet). The generated code becomes the body of the specified method name. The default method is `service`. When used more than once, only the first occurrence of the directive is significant. An example:

```
<%@ method ="doPost" %>
```

import
A comma-separated list of Java language package names or class names that the servlet imports. This directive can be specified multiple times within a JSP file to import different packages. An example:

```
<%@ import ="java.io.*,java.util.Hashtable" %>
```

content_type
The MIME type of the generated response. The default value is text/html. When used more than once, only the first occurrence of this directive is significant. An example:

```
<%@ content_type ="image/gif" %>
```

implements
A comma-separated list of Java language interfaces that the generated servlet implements. You can use this directive more than once within a JSP file to implement different interfaces.

extends
The name of the Java language class that the servlet extends. The class must be a valid class and does not have to be a servlet class. The scope of this directive spans the entire JSP file. When used more than once, only the first occurrence of the directive is significant. An example:

```
<%@ extends ="javax.servlet.http.HttpServlet" %>
```

Specifying class-wide variables and methods

Use the <SCRIPT> and </SCRIPT> tags to declare class-wide variables and class-wide methods for the servlet class. The general syntax is:

```
<script runat=server>

// code for class-wide variables and methods

</script>
```

The attribute `runat=server` is required and indicates that the tag is for server-side processing. An example of specifying class-wide variables and methods:

```
<script runat=server>

// class-wide variables
init i = 0;
String foo = "Hello";

// class-wide methods
private void foo() {
  // code for the method

}
</script>
```

Embedding inline Java code (scriptlets)

You can embed any valid Java language code inline within a JSP file between the <% and %> tags. Such embedded code is called a *scriptlet*. An advantage of embedding Java coding for servlets inline in JSP files is that the servlet does not have to be compiled in advance, and placed on the server. This makes it easier to quickly test servlet coding.

If you do not specify the method directive, the generated code becomes the body of the service method.

The scriptlet for servlets can use a set of predefined variables that correspond to essential servlet, output, and input classes:

request
The servlet request class defined by javax.servlet.http.HttpServletRequest

response
The servlet response class defined by javax.servlet.http.HttpServletResponse

out The output writer class defined by java.io.PrintWriter

in The input reader class defined by java.io.BufferedReader

An example:

```
<%
foo = request.getParameter("Name");
out.println(foo);
%>
```

Specifying variable text using Java

To specify a Java language expression that is resolved when the JSP file is processed, use the JSP expression tags <%= and %>. The expression is evaluated, converted into a string, and displayed. Primitive types, such as int and float, are automatically converted to string representation. In this example, foo is the class-wide variable declared in the <SCRIPT> example in "Specifying class-wide variables and methods" on page 61:

```
<p>Translate the greeting <%= foo %>.</p>
```

When the Web page is served, the text reads: `Translate the greeting Hello.`

Note: Another method for variable substitution is the <INSERT> tag. See "Developing HTML templates for variable data" on page 69 for an explanation.

Accessing JavaBeans

JSP support for JavaBeans enables you to reuse components across your Web site. The JavaBeans can be class files, serialized Beans, or dynamically generated by a servlet.

A JavaBean can even be a servlet (that is, provide a service). If a servlet generates dynamic content and stores it in a Bean, the Bean can then be passed to a JSP file for use within the Web page defined by the file. See "Using the JSP APIs" on page 64 for additional information about the supporting interfaces.

Use the <BEAN> tag to create an instance of a Bean that will be accessed elsewhere within the JSP file. Use JSP syntax and HTML template syntax to access the Bean.

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

introspect
When the value is `yes`, the JSP processor examines all request properties and calls the set property methods (passed in the BeanInfo) that match the request properties. The default value of this attribute is `yes`.

create When the value is `yes`, the JSP processor creates an instance of the Bean if the processor does not find the Bean within the specified scope. The default value is `yes`.

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

- request - The Bean is set as a context in the request by a servlet that invokes the JSP file using the APIs described in "Using the JSP APIs" on page 64. 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.

beanName
The name of the Bean .class file, the Bean package name, or the serialized file (.ser file) that contains the Bean. (This name is given to the Bean instantiator.) 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.

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.

In addition to using the <param> attribute to set Bean properties, there are three other methods:

- Specifying query parameters when requesting the URL of the Web page (JSP file) that contains the Bean. The introspect attribute must be set to yes. An example:

  ```
  http://www.myserver.com/signon.jsp?name=jones&password=d13x
  ```

  where the Bean property name will be set to `jones`.
- Specifying the properties as parameters submitted through an HTML <FORM> tag. The method attribute must be set to `post`. The action attribute is set the URL of the JSP file that invokes the Bean. The introspect attribute must be set to yes. An example:

```
<form action="http://www.myserver.com/SearchSite" method="post">
  <input type="text" name="Search for: ">
  <input type="submit">
</form>
```

- Using JSP syntax to set the Bean property

You cannot call a method of a Bean from within the <BEAN> tag. Use JSP syntax to call a method of a Bean that is instantiated within the JSP file.

After specifying the <BEAN> tag, you can access the Bean at any point within the JSP file. There are three methods for accessing Bean properties:

- Using a JSP scriptlet
- Using a JSP expression
- Using the <INSERT> tag (as described in "Developing HTML templates for variable data" on page 69)

An example:

```
<!-- The Bean declaration -->

<bean name="foobar" type="FooClass" scope="request" >
    <param fooProperty="fooValue" barProperty="1">
</bean>

<!-- Later in the file, some HTML content that includes JSP syntax that calls a
 method of the Bean -->

<p>The name of the row is <%= foobar.getRowName() %>.</p>
```

See the DisplayData.jsp file in "A sample JSP file" on page 66 for an example of each of the three ways to access Beans.

Using the JSP APIs

Two interfaces support the JSP technology. These APIs provide a way to separate content generation (business logic) from the presentation of the content (HTML formatting). This separation enables servlets to generate content and store the content (for example, in a Bean) in the request context. The servlet that generated the context generates a response by passing the request context to a JSP file that contains the HTML formatting. The <BEAN> tag provides access to the business logic.

The interfaces that support JSP are:

- com.sun.server.http.HttpServiceRequest

  This class implements the javax.servlet.http.HttpServletRequest interface and a setAttribute() method to set attributes defined by name.
- com.sun.server.http.HttpServiceResponse

  This class implements the javax.servlet.http.HttpServletResponse interface and adds a callPage() method enabling servlets to call JSP files and optionally pass a context.

When the Web server receives a request for a JSP file, the server sends the request to the Application Server. The Application Server parses the JSP file and generates Java source, which is compiled and executed as a servlet. The generation and compilation of the Java source occurs only on the first invocation of the servlet, unless the original JSP file has been updated. In such a case, the Application Server

detects the change, and regenerates and compiles the servlet before executing it. Figure 2 illustrates this access model for JSP.

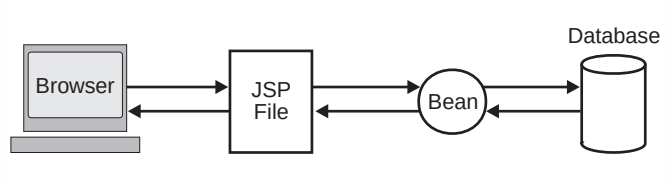

Figure 2. Request sent to a JSP file

The second model for JSP access facilitates the separation of content generation from content display. The Application Server supplies a set of new methods in the HttpServiceRequest object and the HttpServiceResponse object. These methods allow an invoked servlet to place an object (usually a Bean) into a request object and pass that request to another page (usually a JSP file) for display. The invoked page retrieves the Bean from the request object and uses JSP to generate the client-side HTML. Figure 3 illustrates this access model.

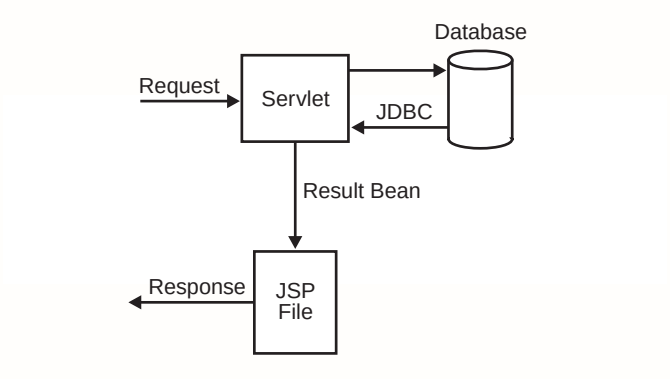

Figure 3. Request sent to a servlet

Using the callPage() method

Use the callPage() method to serve a page from within a servlet. The served page (an HTML or a JSP file) is returned as the response to the browser. If the served page is a JSP file, the calling servlet can also pass some context via the request object. You should code the header of the served page to include a directive to tell the browser to not cache the file. See “Preventing caching of dynamic content” on page 88.

The syntax of the callPage() method is:

```
public void callPage(String fileName,
                     HttpServletRequest req) throws ServletException, IOException
```

where:

fileName

The name of the URL that identifies the file that will be used to generate the output and present the content. If the filename begins with slash (/), the file location is assumed to be relative to the document root. If the filename does not begin with slash, the location is assumed to be relative to the URL with which the current request was invoked.

req The HttpServletRequest object of the servlet that invoked this method. Most often, the content is passed as a Bean in the context of the request object.

To use the callPage() method, you must cast the response object to a special Sun object: com.sun.server.http.HttpServiceResponse. See PopulateBeanServlet "A sample JSP file" for an example.

Using the setAttribute() method

Use the setAttribute() method to store an attribute in the request context. The syntax is:

```
public void setAttribute(String key, Object o)
```

where:

key The name of the attribute to be stored.

o The context object stored with the *key*.

To use the setAttribute() method, you must cast the request object to a special Sun object: com.sun.server.http.HttpServiceRequest. See the PopulateBeanServlet in "A sample JSP file" for an example of the syntax.

A sample JSP file

The sample servlet and JSP file in this section illustrate the JSP access model that enables content separation. The sample includes a servlet (PopulateBeanServlet) developed by the business logic programmer on the Web team. The servlet is invoked from a Web page (not shown) developed by the HTML author on the team.

The servlet:

1. Creates an instance of the DataBean (named dataBean)
2. Sets the Bean parameters
3. Sets the Bean instance (dataBean) as an attribute in the current request object
4. Calls a JSP file (DisplayData.jsp)

The HTML author developed the DisplayData.jsp file. The JSP file gets the dataBean that was passed in the request object and displays the Bean properties.

The PopulateBeanServlet.java sample:

```
import java.io.*;
import java.beans.Beans;

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

import DataBean;

/*****************************************************************
* PopulateBeanServlet - This servlet creates an instance of a Bean
* (DataBean), sets several of its parameters, sets the Bean instance
* as an attribute in the request object, and invokes a JSP file to
* format and display the Bean data.
*****************************************************************/

public class PopulateBeanServlet extends HttpServlet
```

```
{

   public void Service(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
  {
      DataBean dataBean;

      // Create an instance of DataBean

      try
     {
         dataBean = (DataBean) Beans.instantiate(this.getClass().getClassLoader(), "DataBean");
      }
      catch (Exception ex)
      {
         throw new ServletException("Can't create BEAN of class DataBean: "
                                    + ex.getMessage());
      }

      // Set some Bean properties (content generation)
      dataBean.setProp1("Value1");
      dataBean.setProp2("Value2");
      dataBean.setProp3("Value3");

      // To send the Bean to a JSP file for content formatting and display
      // 1) Set the Bean as an attribute in the current request object
      ((com.sun.server.http.HttpServiceRequest) req).setAttribute("dataBean", dataBean);

      // 2) Use callPage to invoke the JSP file and pass the current request object
      ((com.sun.server.http.HttpServiceResponse) res).callPage("/DisplayData.jsp", req);

   }

} /* end of class PopulateBeanServlet */
```

The DisplayData.jsp sample:

```
<!-- This JSP file gets a Bean passed in a request object
      and displays the Bean properties -->

<html>
<head>
<title>Bean Data Display</title>
</head>

<!-- Get the Bean using the BEAN tag   -->
<bean name="dataBean" type="DataBean" introspect="no" create="no" scope="request">
</bean>
<body>
<!-- There are three ways to access Bean properties -->
<!--    Using a JSP scriptlet  -->
<% out.println("The value of Bean property 1 is " + dataBeans.getProp1());
%>

<!--    Using a JSP expression -->
<p>The value of Bean property 2 is
<%= dataBean.getProp2()  %> </p>

<--     Using the INSERT tag   -->
<p>The value of Bean property 3 is
<insert bean=dataBean property=prop3 default="No property value" >
</insert></p>

</body>
</html>
```

Coding relative URLs in invoked JSP files

If a servlet uses the callPage() method to invoke a JSP file and the JSP file contains a relative URL, the browser will resolve the URL as relative to the URL of the Web page from which the servlet was invoked, which might not be the correct base URL.

To ensure that relative URLs in the invoked JSP file are resolved correctly, use the HTML <BASE> tag to specify the base URL. Place the <BASE> tag between the <HEAD> and </HEAD> tags in the JSP file. Add the <HEAD> tags if they aren't present. The syntax of the tag is:

```
<base href="base_URL">
```

As an example, suppose all of the URLs referenced in the JSP file are at http://www.mycompany.com/web and the JSP file contains a link to http://www.mycompany.com/web/store/catalog, your JSP file would include:

```
<html>
<head>
<title>Displaying Data</title>
<base href="http://www.mycompany.com/web">
</head>
<body>
<!-- Some HTML tagging and that includes a link:  -->
<p>Check out our <a href="/store/catalog">catalog</a>.</p>
</body>
</html>
```

Troubleshooting errors in JSP files

If a JSP file contains a JSP syntax error, the browser will report the error to the user. Some of the possible error conditions include:

- The value of an attribute is not enclosed within quotes (″ ″). The message indicates the name of the attribute.
- The JSP file refers to a Bean that does not exist or is not accessible. The message indicates the values of the name and type attributes for the <BEAN> tag.
- Required attributes were not specified on the <BEAN> tag. The message indicates the missing attributes.
- The <BEAN> tag specifies attributes that are not valid. The message indicates the unknown attributes.
- The JSP file refers to class-wide variables that were not declared in the <SCRIPT> tag. The message indicates names of the unknown variables.
- The JSP file contains Java code that is not valid. The message indicates the line number of the compilation error.

Bean introspection and security

By default, a client has access to all of a Bean's properties. The following steps are the steps you can take to prevent client access to sensitive Bean properties:

- Use a *shadow* Bean (also known as a *proxy* Bean) to set non-sensitive Bean properties. The shadow Bean then passes the parameters to the real Bean, which contains the full set of properties. This is the only fully secure way to hide properties.
- Mark sensitive Bean attributes as hidden in BeanInfo PropertyDescriptors.

When the JSP processor performs introspection on the Bean, it respects hidden attributes by refusing to set them from the PARAM attribute on the <BEAN> tag or request parameters.

- Avoid revealing the name of sensitive properties by not using Bean method names, such as setFoo and getFoo.

Multiple browsers on the same machine

When session tracking is enabled, multiple browsers on the same machine will share the same session ID because the session ID is assigned by machine. This fact has implications for Beans accessed by a <BEAN> tag with the scope attribute set to session and the introspect attribute set to yes.

As an example, suppose the Bean is a shopping cart accessed from the shopping cart Web page (a JSP file). The Bean scope is session and introspection is turned on. An instance of the Bean is created whenever a visitor requests to add an item to his shopping cart. Suppose Joe accesses the cart from browser A and submits a request to add a radio to the cart. Then, he accesses the shopping cart Web page from browser B and adds a video to the cart. He returns to browser A and sends a request for a list of the shopping cart contents. The list includes the radio and the video, because only one instance of the Bean was created for the shared session.

Developing HTML templates for variable data

The Application Server HTML template syntax enables you to put variable fields on your HTML page and have your servlets and JavaBeans dynamically replace the variables with values from a database when the page is returned to the browser. This capability is an IBM extension of JSP to make it easier to reference variable data. The syntax can only be used in JSP files.

The HTML template syntax consists of three tags:

- <INSERT> tags for embedding variables in an HTML page
- <REPEAT> tags for repeating a block of HTML tagging that contains the <INSERT> tags and the HTML tags for formatting content
- <REPEATGROUP> for repeating a block of HTML tagging for data that is already logically grouped in the database

These tags are designed to pass intact through HTML authoring tools. Each tag has a corresponding end tag. Each tag is case-insensitive, but some of their attributes are case-sensitive, as explained later in this section.

The basic HTML template syntax

The <INSERT> tag is the base tag for specifying variable fields. The general syntax is:

```
<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. See "Developing JavaServer Pages (JSP)" on page 59 for an explanation of the <BEAN> tag. 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.

Some examples of the basic syntax are:

```
<insert bean=userProfile property=username></insert>
<insert requestparm=company default="IBM Corporation"></insert>
<insert requestattr=ceo default="Company CEO"></insert>
<insert bean=userProfile property=lastconnectiondate.month></insert>
```

In most cases, the value of the property attribute will be just the property name. However, you access a property of a property (sub-property) by specifying the full form of the property attribute. The full form also gives you the option to specify an index for indexed properties. The optional index can be a constant (such as 2) or an index like the one described in "Repeating HTML tagging" on page 71. Some examples of using the full form of the property tag:

```
<insert bean=staffQuery property=address(currentAddressIndex)></insert>
<insert bean=shoppingCart property=items(4).price></insert>
<insert bean=fooBean property=foo(2).bat(3).boo.far></insert>
```

The alternate HTML template syntax

The HTML standard does not permit embedding HTML tags within HTML tags. Consquently, you cannot embed the <INSERT> tag within another HTML tag, for example, the anchor tag (<A>). Instead, use the HTML template alternate syntax.

To use the alternate syntax:

1. Use the <INSERT> and </INSERT> to enclose the HTML tag in which substitution is to take place.
2. Specify the Bean and property attributes:
   - To specify the Bean and property attributes, use the form:
     `$(bean=`*b* `property=`*p* `default=`*d*`)`

     where *b*, *p*, and *d* are values as described for the basic syntax.

- To specify the requestparm attribute, use the form
  `$(requestparm=r default=d)`

  where *r* and *d* are values as described for the basic syntax.
- To specify the requestattr attribute, use the form
  `$(requestattr=r default=d)`

  where *r* and *d* are values as described for the basic syntax.

Some examples of the altertnate HTML template syntax:

```
<insert>
  <img src=$(bean=productAds property=sale default=default.gif)>
</insert>

<insert>
  <a href="http://www.myserver.com/map/showmap.cgi?country=$(requestparms=country default=usa)
  &city$(requestparm=city default="Research Triangle Park")&email=
     $(bean=userInfo property=email)>Show map of city</a>
</insert>
```

Repeating HTML tagging

Use the <REPEAT> and <REPEATGROUP> tags to repeat the HTML template syntax and the associated HTML formatting in a section of your Web page (JSP file). These tags are especially useful for formatting tables and lists.

<REPEAT> tag

The syntax of the <REPEAT> tag is:

```
<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.

Examples 1, 2, and 3 show how to use the <REPEAT> tag. The examples produce the same output if all indexed properties have 300 or fewer elements. If there are more than 300 elements, Examples 1 and 2 will display all elements, while Example 3 will show only the first 300 elements.

Example 1 shows implicit indexing with the default start and default end index:

```
<table>
<repeat>
  <tr><td><insert bean=serviceLocationsQuery property=city></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery property=address></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery property=telephone></insert></tr></td>
</repeat>
</table>
```

Example 2 shows indexing, starting index, and ending index:

```
<table>
<repeat index=myIndex start=0 end=2147483647>
  <tr><td><insert bean=serviceLocationsQuery property=city(myIndex)></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery property=address(myIndex)></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery property=telephone(myIndex)></insert></tr></td>
</repeat>
</table>
```

Example 3 shows explicit indexing and ending index with implicit starting index. Although the index attribute is specified, the indexed property city can still be implicitly indexed because the (i) is not required.

```
<table>
<repeat index=myIndex end=299>
  <tr><td><insert bean=serviceLocationsQuery property=city></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery property=address(i)></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery property=telephone(i)></insert></tr></td>
</repeat>
</table>
```

You can nest <REPEAT> blocks. Each block is separately indexed. This capability is useful for interleaving properties on two beans, or properties that have sub-properties. In the example, two <REPEAT> blocks are nested to display the list of songs on each compact disc in the user's shopping cart.

```
<repeat index=cdindex>
  <h1><insert bean=shoppingCart property=cds.title></insert></h1>
  <table>
  <repeat>
    <tr><td><insert bean=shoppingCart property=cds(cdindex).playlist></insert>
    </td></tr>
  </table>
</repeat>
```

<REPEATGROUP> tag

The <REPEATGROUP> tag enables you to create repeating HTML blocks to display data that is already grouped in the database. The tag syntax is:

```
<repeatgroup bean=name property=property_name
   start=starting_index end=ending_index>
</repeatgroup>
```

where:

bean The name of the JavaBean that will access the grouped data. The Bean is declared by a <BEAN> tag within the JSP file. See "Developing JavaServer Pages (JSP)" on page 59 for an explanation of the <BEAN> tag. Do not specify this attribute on the inner-most block of nested <REPEATGROUP> blocks. The value is case-sensitive.

property
The property of the Bean to access for grouping. 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 attribute. Do not specify this attribute on the inner-most block of nested <REPEATGROUP> blocks. The value is case-sensitive.

start An optional starting index value for this repeatgroup block. The default value is 0. Specify this attribute only for the outer-most nested block. The attribute is ignored on all inner nested blocks.

end An optional ending index value for this repeatgroup 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. Specify this attribute only for the outer-most nested block. The attribute is ignored on all inner nested blocks.

The <REPEATGROUP> tag is useful when combined with a database query that groups the results into categories. The data is assumed to be grouped. Otherwise, the result will not be the desired output.

The same index is used throughout all <REPEATGROUP> blocks. Each index of the Bean's property is substituted into the output HTML only once. A block is repeated when the specified value changes.

In this example, the database contains data grouped by country and state:

country	state	city	address	telephone
Canada	Ontario	Ottawa	8 Leaf Lane	000-555-0000
Canada	Ontario	Toronto	9 Main Street	000-555-1111
USA	GA	Atlanta	5 Support Plaza	111-555-0123
USA	NC	Charlotte	3 Main Street	222-555-3456
USA	NC	Raleigh	2 Main Street	333-555-6789

The HTML author creates a Web page (JSP file) that uses nested <REPEATGROUP> tags to output an <h1> heading specifying the country name, an <h2> heading specifying the state or province within the country, and a table that lists the available service centers by city. The inner-most block repeats until there are no more cities in the database for that state. The state block repeats until there are no more states in the database for the country. The outer-most block repeats until there are no more countries in the database. The HTML coding is:

```
<repeatgroup bean=serviceCenterQuery property=country>
  <h1><insert bean=serviceCenterQuery property=country></insert></h1>
  <hr>
  <repeatgroup bean=serviceCenterQuery property=state>
    <h2><insert bean=serviceCenterQuery property=state></insert></h2>
    <table>
       <tr>
       <th>City</th>
       <th>Address</th>
       <th>Phone</th>
       </tr>
       <repeatgroup>
          <tr>
          <td><insert bean=serviceCenterQuery property=city></insert></td>
          <tr><insert bean=serviceCenterQuery property=address></insert></td>
          <tr><insert bean=serviceCenterQuery property=telephone></insert></td>
          </tr>
       </repeatgroup>
    </table>
  </repeatgroup>
</repeatgroup>
```

The output for the nested <REPEATGROUP> tags is:

```
Canada
-----------------------------------------------
Ontario
```

City	Address	Telephone
Ottawa	8 Leaf Lane	000-555-0000
Toronto	9 Main Street	000-555-1111

USA

GA

City	Address	Telephone
Atlanta	5 Support Plaza	111-555-0123

NC

City	Address	Telephone
Charlotte	3 Main Street	222-555-3456
Raleigh	2 Main Street	333-555-6789

Posting site-wide bulletins

The com.ibm.servlet.servlets.personalization.util package has SetVariableText and GetVariableText servlets that let you add bulletins or news flashes to your Web site on a page-by-page basis. This can be useful to instantly broadcast site-wide or application-specific information. For example, store managers can use it to advertise a special sale, or site administrators can use it to remind users of upcoming deadlines.

- SetVariableText

  This servlet adds a form to your Web page that allows the user to send a text bulletin to whomever is currently at all or some of the pages in the Web site. The form has two input fields: an identifying queue name of the bulletin and the content of the bulletin. When the form is returned, the text is put into the Session object.

  The queue name lets you categorize and identify the bulletins, allowing you to target the bulletins from a specific queue to the appropriate Web pages. These names must be decided beforehand and made known to the people sending the bulletins and to those creating the pages that display them.

  For example, if your server hosts a sports Web site and a horticulture Web site, you can use *sports* and *plants* for the queue names and specify them when you send the bulletins. Then, on the sports site's pages, you can embed an instance of the GetVariableText servlet that checks for bulletins with the queue name of *plants*. On the horticulture site's pages, embed the one that checks for *plants*. Likewise, you can define a queue for *all* and add the corresponding servlet to every page on the server.

- GetVariableText

  This servlet checks the Session object for a bulletin, identified by the bulletin's queue name, and returns it. Specify the queue name as an initialization parameter on the <SERVLET> tag. For example, use <SERVLET NAME=PlantGetVarText queueName=plants>.

To enable the site-wide bulletin function, embed either of these servlets in your HTML pages according to the function to provide. For example, to allow a Web site administrator to send bulletins, embed an instance of the SetVariableText servlet in a page the administrator accesses to set the bulletin. To propagate the

bulletins, embed an instance of the GetVariableText servlet in all the pages on which you want visitors to be able to see the bulletin.

Enabling your Web visitors to exchange messages

The com.ibm.servlet.servlets.personalization.util package has SendMessage, CheckMessage, and GetMessage servlets that let you add person-to-person messages to your Web pages. You can embed instances of these servlets in your HTML according to the function to provide on that page. For example, to send and receive messages from the same HTML page, embed instances of both the SendMessage and GetMessage servlets in the page.

- SendMessage

  This servlet adds a simple form button to your Web page that, when clicked, displays a dialog allowing the person using the page to send a text message to someone else who is currently at the Web site. The dialog has two input fields:

  – A field for selecting the recipient of the message

  – A text entry field for the content of the message

  The servlet finds all the current users on the site (meaning all those with a Session object) and builds the selection list with their user names. When the form is returned, the message is put into the recipient's Session object.

- CheckMessage

  This servlet checks to see if the person on this Web page has messages in their SessionContextImpl object and renders one of two images on the page. If there is a message, it renders the MessageWaiting image. Otherwise, it renders the NoMessage image. The MessageWaiting image is a hot link that, when clicked, calls the GetMessage servlet.

  The Application Server provides a GIF image depicting a telephone for the NoMessage image, and an animated GIF image depicting a ringing telephone for the MessageWaiting image. You can replace these with your own images.

- GetMessage

  This servlet is called from the MessageWaiting image. It checks the Web page user's SessionContextImpl for a message. If it finds one, it dynamically creates an HTML fragment with the message text, returns the HTML, and clears the message queue. To replace the HTML fragment with a customized message page that includes the message text fragment, embed the GetMessage servlet in the replacement page and change the MessageWaiting image so that it links to the customized page instead of the servlet.

To enable the message function:

1. Add an instance of the SendMessage servlet to every page on which you want to enable visitors to send messages to other people at the Web site.
2. Add an instance of the CheckMessage servlet to every page on which you want to enable visitors to receive messages from other people at the Web site.

Chapter 7. Placing servlets on your Application Server

This section contains detailed instructions for compiling servlets, placing servlets and related files on the Application Server, and configuring the servlet.

In this section, the following conventions apply:

- The root directory for your Web server is called *server_root*. Refer to your Web server configuration to determine the specific location for your installation.
- The servlet root directory is *applicationserver_root*\servlets, where *applicationserver_root* is the Application Server root directory.
- The HTML document root directory is *server_root**HTML_directory*, where *HTML_directory* is the directory from which HTML files are served. Refer to your Web server configuration to determine the specific location for your installation.
- *JAVA_HOME* is the Java Development Kit (JDK) root directory.
- Backslash (\ ) is used as the separator in directory path names. Depending on your server platform, you have either \ or / as a file separator.

Step 1: Compiling servlets

To compile a servlet:

1. Ensure that the CLASSPATH environment variable includes the Application Server JAR files and the JDK classes.zip file:
   - For Windows NT, type the following on one line:

     ```
     set CLASSPATH=.;JAVA_HOME\lib\classes.zip;
     applicationserver_root\lib\ibmwebas.jar;
     applicationserver_root\lib\jst.jar;applicationserver_root\lib\jsdk.jar;
     applicationserver_root\lib\x509v1.jar;applicationserver_root\lib;
     applicationserver_root\web\classes\ibmjbrt.jar;
     applicationserver_root\lib\databeans.jar;%CLASSPATH%
     ```

   - For C-shell on Solaris and AIX, type the following on one line:

     ```
     setenv CLASSPATH .:JAVA_HOME/lib/classes.zip:
     applicationserver_root/lib/ibmwebas.jar:
     applicationserver_root/lib/jst.jar:applicationserver_root/lib/jsdk.jar:
     applicationserver_root/lib/x509v1.jar:applicationserver_root/lib:
     applicationserver_root/web/classes/ibmjbrt.jar:
     applicationserver_root/lib/databeans.jar:$CLASSPATH
     ```

   - For Korn-shell on Solaris and AIX, and OS/390 OpenEdition shell, type the following on one line:

     ```
     export CLASSPATH=.:JAVA_HOME/lib/classes.zip:
     applicationserver_root/lib/ibmwebas.jar:
     applicationserver_root/lib/jst.jar:applicationserver_root/lib/jsdk.jar:
     applicationserver_root/lib/x509v1.jar:applicationserver_root/lib:
     applicationserver_root/web/classes/ibmjbrt.jar
     applicationserver_root/lib/databeans.jar::$CLASSPATH
     ```

2. Set the PATH environment variable to include the java/bin directory by using the commands:
   - For Windows NT:

     ```
     set PATH=JAVA_HOME\bin;%PATH%
     ```

   - For C-shell on Solaris and AIX:

     ```
     setenv PATH "JAVA_HOME"/bin:"$PATH"
     ```

   - For Korn-shell on Solaris and AIX, and OS/390 OpenEdition shell:

```
export PATH="JAVA_HOME"/bin:"$PATH"
```

3. Test that the appropriate Java Development Kit (JDK) is in your path by issuing the command:

   ```
   java -version
   ```

   This command should return a message stating what the JDK version is. If the command is not found, you will see an error message.

4. Compile the servlet by issuing the command:

   ```
   javac filename.java
   ```

Step 2: Placing servlet class files on the Application Server

By default, the Application Server looks for servlet class files in the servlet root directory, *applicationserver_root*\servlets. Copy your compiled servlet class files to that directory. If the servlets are part of a package, be sure to copy them to the correct subdirectory under *applicationserver_root*\servlets.

To load servlets from a local directory other than *applicationserver_root*\servlets, see "Migrating servlets" on page 12 for instructions on setting the servlets.classpath property. You can also load a servlet from a remote system. To remotely load a servlet, specify the remote system when you configure the servlet using the Application Server Manager.

If your servlets import non-servlet classes that you developed, it is recommended that you copy those classes to *applicationserver_root*\servlets.

Stdout for all servlets goes to *applicationserver_root*\logs\ncf.log or to the Java console window, depending on settings in the jvm.properties file. See "Configuring setup parameters" on page 10 for instructions on enabling Java standard out logging.

Step 3: Placing the HTML files on the Application Server

Copy the HTML, JSP, and SHMTL files for the servlet to the Web server's HTML document root directory, *server_root**HTML_directory*. This directory is determined by your specific server configuration (the settings for pass, alias, and virtual hosting rules). You should code your servlets to use the method that returns the location of the servlet's HTML file:

```
ServletRequest.getRealPath("/my.html")
```

where *my.html* is the name of the servlet's HTML file.

The Web server document root directory is accessible from your Web browser by opening the URL:

```
http://your.server.name/
```

You can place HTML files in subdirectories relative to the document root directory, in *HTML_directory*/morefiles for example. In such a case, to open the HTML file from a browser, specify the relative directory in the URL. For example, if you added the morefiles subdirectory under the document root directory and place an HTML file there, open the following URL to view the file:

```
http://your.server.name/morefiles/myhtml.html
```

Step 4: Configuring servlets

If you want to load a servlet from a JAR or SER file on a remote system, or set servlet initialization parameters, use the Application Server Manager to configure the servlet:

1. Open the Application Server Manager at http://*your.server.name*:9090/ and log in.
2. In the list of Services, double-click the instance of the Application Server to manage.
3. On the next page, select the Servlets button.
4. If the servlet is not already listed in the Servlets tree-view, select the Add option. Follow the online Help to complete the page.
5. After the servlet is added to the tree-view, select the servlet entry in the tree and then select the Configure option.

   Note: If your Application Server is running on Netscape Enterprise or Netscape FastTrack Server under Sun Solaris and you specify a JAR file for the remote load, be sure that the JAR file was created with the **no** compression flag (-0) set. If compression was specified when the JAR file was created, the browser will return an error 500 when you try to invoke the servlet.

Chapter 8. Invoking servlets

This section describes how to invoke a servlet you have placed on and defined to the server. You can:

- Invoke the servlet by its URL
- Specify the servlet on the ACTION attribute of the <FORM> tag in an HTML file
- Specify the servlet within the <SERVLET> tag in an SHTML file
- Embed the Java coding for a servlet or use the <BEAN> tag to embed a JavaBean in a JSP file

Invoking a servlet by its URL

There are two methods for invoking a servlet from a browser using its URL:

- Specifying the servlet name:

  When you use the Application Server Manager to add (register) a servlet instance to the server configuration, you must specify a value for the Servlet Name parameter.

  For example, you might specify hi as the Servlet Name for HelloWorldServlet. To invoke the servlet, you would open http://*your.server.name*/servlet/hi.

  You can specify a servlet name that is the same as the class name (HelloWorldServlet). In such a case, the instance of the servlet would be invoked by http://*your.server.name*/servlet/HelloWorldServlet.
- Specifying the servlet alias:

  Use the Application Server Manager to configure a servlet alias, which is a shortcut URL for invoking the servlet. The shortcut URL does not include the servlet name.

Specifying a servlet within the <FORM> tag

You can invoke a servlet within the <FORM> tag. An HTML form enables a user to enter data on a Web page (from a browser) and submit the data to a servlet.

If the information entered by the user is to be submitted to the servlet by a GET method, the servlet must override the doGet() method. To invoke the servlet, use:

```
<FORM METHOD="GET" ACTION="/servlet/myservlet">
(Tags to place text entry areas, buttons, and other prompts go here)
</FORM>
```

If the information entered by the user is to be submitted to the servlet by a POST method, the servlet must override the doPost() method. To invoke the servlet, use:

```
<FORM METHOD="POST" ACTION="/servlet/myservlet"
(Tags to place text entry areas, buttons, and other prompts go here.)
</FORM>
```

ACTION attribute

The ACTION attribute indicates the URL used to invoke the servlet. See “Invoking a servlet by its URL”.

METHOD attribute

With the GET method, the user-supplied information is URL-encoded in a query string. You do not have to do the URL encoding, because it is done by the form. The URL-encoded query string is then appended to the servlet URL and the entire URL is submitted.

The URL-encoded query string pairs the name of the visual component with the value selected by the user, based on user interaction with the visual component. For example, consider the following HTML fragment, which displays buttons, labeled AM and FM.

```
<OL>
<INPUT TYPE="radio" NAME="broadcast" VALUE="am">AM<BR>
<INPUT TYPE="radio" NAME="broadcast" VALUE="fm">FM<BR>
</OL>
```

If the user selects the FM button, the query string will contain the *name=value* pair:

```
broadcast=fm
```

Because the servlet in this case is intended to respond to an HTTP request, the servlet should be based on the HttpServlet class. The servlet should override the doGet() or doPost() method, depending on whether the user information in the query string is submitted to the servlet using the GET or POST method.

As discussed earlier, the getParameterNames(), getParameter(), and getParameterValues() methods are available for extracting the form parameter names and values. The extraction process also decodes the names and values.

Often, the final action of the servlet is to dynamically create an HTML response (based on parameter input from the form) and pass it back to the user through the server. Methods of the HttpServletResponse object are used to send the response, which is sent back to the client as a complete HTML page.

Specifying a servlet within the <SERVLET> tag

When you use the <SERVLET> tag to invoke the servlet, you do not create an entire HTML page, as you do when using the <FORM> tag. Instead, the output of the servlet is only part of an HTML page, and is dynamically embedded within the otherwise static text of the original HTML page. All this takes place on the server (a process called *server-side include*) and only the resulting HTML page is sent to the user. The recommended approach is to use the <SERVLET> tag in Java Server Pages (JSP). See “Invoking a servlet within a JSP file” on page 84 for more information.

The original HTML page contains the <SERVLET> and </SERVLET> tags. The servlet is invoked within the tags, and everything between and including the two tags is overlaid with the servlet’s response. If the user’s browser can look at the HTML source, the user will not see the <SERVLET> and </SERVLET> tags.

To use this method of invoking servlets on a Domino Go Webserver, enable the server-side include function on the server. Part of that enablement involves adding a special file type, SHTML.

When the Web server receives a request for a Web page with the extension SHTML, it searches for the <SERVLET> and </SERVLET> tags. For all of the supported Web servers, the Application Server processes the information between the SERVLET tags.

The following HTML fragment shows how to use this technique.

```
<SERVLET NAME="myservlet" CODE="myservlet.class" CODEBASE="url" initparm1="value">
<PARAM NAME="parm1" VALUE="value">
</SERVLET>
```

NAME, CODE, and CODEBASE attributes

Using the NAME and CODE attributes provides flexibility. Either or both can be used. The NAME attribute specifies a servlet name (configured using the Application Server Manager), or the servlet class name without the .class extension. The CODE attribute specifies the servlet class name.

With the Application Server, it is recommended that you specify both NAME and CODE, or only NAME if the NAME specifies the servlet name. If only CODE is specified, an instance of the servlet with NAME=CODE is created.

The loaded servlet will assume a servlet name matching the name specified in the NAME attribute. Other SHTML files can then successfully use the NAME attribute specifying the servlet name, and invoke the already loaded servlet. The NAME value can be used directly in the URL to invoke the servlet.

If both NAME and CODE are present, and NAME specifies an existing servlet, the servlet specified in NAME is always used. Because the servlet creates part of an HTML file, you will probably use a subclass of HttpServlet when creating the servlet and override the doGet() method, because GET is the default method for providing information to the servlet. Another option is to override the service() method.

CODEBASE is optional and specifies a URL on a remote system from which the servlet is to be loaded. Use the Application Server Manager to configure remote loading of a servlet from a JAR file.

Parameter attributes

In the previous tagging example, *initparm1* is the name of an initialization parameter and *value* is the value of the parameter. You can specify more than one set of name-value pairs. Use the getInitParameterNames() and getInitParameter() methods of the ServletConfig object (which is passed into the servlet's init() method) to find a string array of parameter names and values.

In the example, *parm1* is the name of a parameter that is set to *value* after the servlet is initialized. Because the parameters set using the <PARAM> tag are available only through methods of a Request object, the server must have invoked the servlet service() method, passing a request from a user. To get information about the user's request, use the getParameterNames(), getParameter() and getParameterValues() methods.

Initialization parameters are persistent. Assume that a client invokes the servlet by invoking an SHTML file containing some initialization parameters. Next, assume that a second client invokes the same servlet through a second SHTML file, which

does not specify any initialization parameters. The initialization parameters set by the first invocation of the servlet remain available and unchanged through all successive invocations of the servlet through any other SHTML file. You cannot reset the initialization parameters until after the destroy() method has been called on the servlet. For example, if another SHTML file specifies a different value for an initialization parameter but the servlet is already loaded, the value is ignored.

In contrast, the parameters set within the <PARAM> attribute are for a specific invocation of the servlet. If a second SHTML file invokes the same servlet with no <PARAM> parameters, the <PARAM> parameters set by the first invocation of the servlet are not available to the second invocation of the servlet.

Invoking a servlet within a JSP file

You can invoke servlets from within JavaServer Pages (JSP) files. See “Developing JavaServer Pages (JSP)” on page 59 for a complete description of the JSP syntax.

To invoke a JSP file:

- Use a Web browser to open the JSP file
- Invoke a servlet that invokes the JSP file

Chapter 9. Tips

This section contains tips for developing servlets.

Migrating servlets from the beta Java Servlet API

This section provides tips for migrating servlets that were developed using the beta Java Servlet API.

- The beta version of the Java Servlet API lets you overload an init() method that has no function arguments. The Java Servlet API 1.0 does not allow an init() method with no function arguments. If you overload the Servlet.init method, the init() method must have the following arguments to enable your overloaded function to be invoked when the servlet is initialized:
  - For beta version (no longer valid):

    ```
    public void init()
    ```

  - For 1.0 version:

    ```
    public void init(ServletConfig config)

    and you must call superinit(config)
    ```

- Servlets should overload the doGet() and doPost() methods instead of the service() method. During servlet execution, the appropriate method will be called according to the clients request method.
- Servlets must now import javax.servlet.* instead of java.servlet.*.

Migrating servlets from ServletExpress betas

If you want to use servlets that were developed using beta releases of IBM ServletExpress, you might need to modify your servlets to account for:

- The Application Server does not support the <!META> tag. Use the <INSERT> tag for variable substitution. See “Developing HTML templates for variable data” on page 69 for details.
- The Application Server and ServletExpress include personalization classes that are extensions and additions to the Java Servlet API. However, the package names for those classes differ between the products. See “Servlet packages” on page 17 for a list of the new package names.

Overriding Servlet API class methods

You can override any method in the Servlet API. However, when you override the init(ServletConfig) method, call super.init(ServletConfig). The default init() method contains actions that will no longer occur if the method is overridden. In particular, no ServletConfig object will be returned for use in initializing the servlet, so it will not be initialized on the server. By calling the method in the superclass, those default behaviors will still be performed, as well as the code in the overridden method. The init() method now requires parameters, unlike the init() method supported in the beta-level Servlet API.

To debug your servlet code, put a try{}catch(){} block around all of the code in your overridden service(), doGet(), or doPost() method after the point where you initialize your Response output stream. Use the output stream to write the stack trace to the client browser page. This will show any errors or exceptions that are occurring during runtime.

Sample code:

```
public void service(ServletRequest req, ServletResponse res)
{
   variable declaration block ....

   res.setContentType("text/plain");
   PrintWriter pw = res.getWriter();

   try
   {
      rest of service() code .....
   }
   catch(Throwable e)
   {
      pw.println(e.toString());
      PrintStream out = new PrintStream(res.getOutputStream());
      e.printStackTrace(out);
      out.close();
      pw.close();
   } // end of catch(){}
} // end of service()
```

To ensure that the servlet will handle the double byte character set (DBCS) in the output stream, wrap the response stream in an outputStreamWriter class:

```
outputStreamWriter out = new OutputSreamWriter(res.getOutputStream());
```

Making servlets threadsafe

By design, servlets have multiple concurrent calls to the service() method, one for each client request. Each call to service(), doGet(), or doPost() obtains a thread from the servlet thread pool. When that call returns, the thread is returned to the pool and is available for another service() method call instance to use. Also, at any point in the servlet code, other threads or thread groups can be spawned for processing concurrent tasks.

Ensure that servlets are threadsafe. This means that critical sections of code must be protected in some manner. Critical sections are code segments that may be executed concurrently by separate threads and segments for which simultaneous access to the same data will result in incorrect behavior.

To maintain good object-oriented programming, practice critical sections should be entire methods, rather than just lines within a method. Critical sections should be made as small as possible so that protection mechanisms do not degrade performance more than necessary. Critical sections should be written to avoid conditions in which a thread never gets a needed resource (starvation) or each of two threads is waiting for something from the other (deadlock).

Critical section protection mechanisms include the use of the Java synchronized keyword. When a thread is executing declared, synchronized code, the thread blocks access to that code by any other thread until execution is completed. Java objects that have a synchronized method are associated with a unique monitor for each instance of that object. The thread that called the method gets control of the

monitor and blocks other threads from calling that synchronized method until the monitor is released by the thread that has control. This getting and releasing of monitor control is done automatically by the Java runtime environment (JRE).

Variables local to the service(), doGet(), or doPost() method and any methods called from within these methods differ from variables global to the servlet. Scoping rules say that each service() method call thread gets its own copy of those local variables. Therefore, threadsafe programming is only necessary for threads spawned from within service() method. Those spawned threads can concurrently access the variables local to the service() method, but global to the spawned threads. Variables that are global to the servlet whose values are being changed must be declared synchronized, because there is only one instance of those variables and all service() method call threads have access to that one instance.

For further information, read the sections on threading in *The Java Tutorial* at the URL: http://www.javasoft.com/docs/books/tutorial/index.html. There are other excellent books about concurrent programming, both Java-specific and for programming in general. Consult some of these for in-depth information on techniques for making code threadsafe, such as monitors and thread priorities, and to learn how to write code to detect and prevent starvation and deadlock conditions.

Using common methods

The **init()** method is called, automatically by the server when the servlet is first invoked. This method will not be called again. The init() method sets the servlet's initialization parameters and startup configuration using its ServletConfig object parameter, so any servlet that overrides the init() method should call super.init() to ensure these tasks are still performed. The init() method is guaranteed to complete before the service() method is called.

The **service()** method is called after initialization is complete to receive a single client request contained in a Request object parameter. It uses the response object parameter to return information to the client. The underlying network transport mechanisms are used by the request object input stream and response object output stream. Synchronize concurrent thread access to resources. Because the HttpServlet.service() method checks whether the request method is HEAD, POST, or GET and calls the appropriate handler method of doPost() or doGet(), it is not necessary to override the service() method.

The **doGet()** method is called from HttpServlet.service() if the request method is GET. If the client sends only the GET request, you can override the doGet() method. If the request is to change stored data or to purchase something online, do not use the GET method, because of possible security breaches. Write all doGet() overrides so that they are both safe and safely repeated.

The **doPost()** method is called from HttpServlet.service() if the request method is POST. If the client sends only the POST request, you can override the doPost() method.

The **destroy()** method is called only once, automatically by the server, when the servlet is unloaded, to clean up any resources held by the servlet. When the server unloads a servlet, the destroy() method is called after all service() method calls complete or after a specified time interval. Where threads have been spawned from within service() method and the threads have long-running operations, those

threads may be outstanding when destroy() method is called. Because this is undersirable, make sure those threads are ended or completed when destroy() method is called.

Preventing caching of dynamic content

You can prevent Web browsers from caching dynamically generated Web pages (meaning dynamic output that results from processing JSP files, JHTML files, SHTML files, and servlets). Set the following fields in the header of the HTTP Response:

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

Tuning performance

You can improve Application Server performance in environments that are not sensitive to security. Specifically, you can improve access control (ACL) by setting Application Server properties:

1. Use a text editor to open the server.properties file in the path *applicationserver_install_root*\properties\server\servlet
2. Set the property server.security=false.
3. Use a text editor to open the httpd.properties file in the path *applicationserver_install_root*\properties\server\servlet\servletservice
4. Set the property enable.acls=false.

Enabling the Debug stage processor

Application Server includes a Debug stage processor that provides request and header information. To enable the processor:

1. Open the Application Server httpd.properties file in the path *applicationserver_install_root*\properties\server\servlet\servletservice
2. Set the property pipeline.stages=debug, resolver, runner.
3. Add the property pipeline.stage.debug.class=com.sun.server.http.stages.Debug after the similar property for the Runner.

The output of Debug goes to the Java console window or to the Application Server ncf.log file, depending on settings in the Application Server jvm.properties file. See “Configuring setup parameters” on page 10 for instructions on setting Java standard out logging.

Generating responses from multiple threads

If your Web server does not support generating responses from multiple threads, you want to use remote loading of servlets on a Domino Go Webserver, or you want to use servlet filtering on a Domino Go Webserver, perform the following procedure:

1. Open the Application Server httpd.properties file in the path *applicationserver_install_root*\properties\server\servlet\servletservice
2. Set the property plugin.multithreaded=false.

Using the attributes for access to SSL information

The Java Servlet API specifies three attributes that return SSL information:

- SSL peer certificates (javax.net.ssl.peer_certificates)
- The name of the SSL cipher suite in use (javax.net.ssl.cipher_suite)
- The SSL session object (javax.net.ssl.sslsession)

At the time of this publication, most of the attributes are not supported by the Web servers on which the Application Server can be installed. Only javax.net.ssl.peer_certificates is supported by the Netscape servers and Microsoft IIS.

Use javax.net.ssl.peer_certificates to gain access to an SSL peer certificate on a given request extracted by the Web server. You could use that information to handle your own user authentication. A code example:

```
...

X509Cert certChain [] = (X509Cert [])req.getAttribute ("javax.net.ssl.peer_certificates");
if (certChain != null)
{
  //Now in SSL and can use the certificate information to query
  //the user and authenticate
  out.println("<h1>HTTPS Information:<h1>");
  out.println("<pre>");

  for (int i = 0;i < certChain.length;i++)
  {
    X500Name issuer = certChain[i].getIssuerName().getName();
    doSomethingWith(issure);
  }
...
}
...
```

Solving problem with legacy SHTML files on Apache server

If you have problems using legacy SHTML files on your Apache server after the Application Server is installed, try this:

1. Open the Application Server rules.properties file in the path: *applicationserver_install_root*\properties\server\servlet\servletservice
2. Comment out the following pass rule: .shtml=pageCompile
3. Save the file.

Using servlets in Active Server Pages (ASP)

If you have legacy ASP files on your Microsoft Internet Information Server (IIS), you can use the ASP files to invoke servlets, if you are not able to migrate the ASP files to JSP files. The ASP support in Application Server consists of an ActiveX control for embedding servlets.

Before using this function, become familiar with the ASP programming model and with servlets.

AspToServlet reference

When Application Server is installed, the ASP support is installed and registered on your Windows NT system. This section describes the methods and properties for the ActiveX control AspToServlet.AspToServlet.

The methods are:

- String ExecServletToString(String servletName)

  Execute servletName and return its output in a String
- ExecServlet(String servletName)

  Execute servletName and send its output to the HTML page directly
- String VarValue(String varName)

  Get a pre-set (additional form) variable value
- VarValue(String varName, String newVal)

  Set a variable value. The total size occupied by your variable should be less than 0.5 Kbyte. Use these variables only for configuration.

The properties are:

- Boolean WriteHeaders

  If this property is true, servlet provided headers are written to the user. The default value is false.
- Boolean OnTest

  If this property is true, the server will log messages into the generated HTML page. The default value is false.

Sample ASP Script

This section contains a sample ASP file. The page is written in Microsoft Visual Basic Scripting (VBScript). The AspToServlet code is in bold.

```
<%
    ' Small sample asp file to show the capabilities of the servlets
    ' and the ASP GateWay ...
%>
<H1> Starting the ASP->Java Servlet demo</H1>
<%
    ' Create a Servlet gateway object and initialize it ...
    Set javaasp = Server.CreateObject("AspToServlet.AspToServlet")
    ' Setting these properties is only for the sake of demo.
    ' These are the default values ...
    javaasp.OnTest = False
    javaasp.WriteHeaders = False
    ' Add several variables ...
    javaasp.VarValue("gal") = "lag"
    javaasp.VarValue("pico")= "ocip"
    javaasp.VarValue("tal") = "lat"
    javaasp.VarValue("paz") = "zap"
    javaasp.VarValue("variable name with spaces") = "variable value with spaces"
%>
<BR>
Lets check the variables
<%
    Response.Write("variable gal = ")
    Response.Write(javaasp.VarValue("gal"))
%>
<BR>
<%
    Response.Write("variable pico = " & javaasp.VarValue("pico"))
```

```
%>

<BR>
<HR>
<%
    galout = javaasp.ExecServletToString("SnoopServlet")
    If javaasp.WriteHeaders = True Then
%>
Headers were written <%
    Else
%>
Headers were not written <%
    End If
    Response.Write(galout)
%>
<H1> The End ...</H1>
```

Chapter 10. Running sample servlets and applications

The Application Server includes sample servlets and Web site applications. These demonstrate both simple techniques for request and response processing and more advanced techniques for database access, maintaining state information, authenticating users, posting site bulletins, and enabling Web visitors to exchange messages.

Viewing the samples

Compiled copies of the samples are on your system and ready for use. You do not need to recompile the samples unless you want to make changes. Before you use the samples that demonstrate database access, follow the instructions for preparing the database, as described in "Running the database samples" on page 94.

For quick access to the samples, use your browser to open the samples index page:

`http://your.server.name/IBMWebAS/samples/`

where *your.server.name* is the hostname of your Web server.

The source code (.java files) and HTML files for a given sample is in the directory:

applicationserver_root`\samples\`*sample_name*

where *sample_name* is the name of the sample, such as `ReqInfoServlet`.

Running the non-database samples

This section describes the samples that do not require a database.

ReqInfoServlet

This servlet extracts information about the servlet request and returns it to the client. This servlet does not have HTML pages. To run the servlet, use either of the following methods:

- Use your browser to open the samples index page and select the servlet:

  `http://your.server.name/IBMWebAS/samples/`
- Use your browser to open the URL of the servlet:

  `http://your.server.name/servlet/ReqInfoServlet`

FormDisplayServlet

This servlet reads input from an HTML form and returns it in a Web page. To run the servlet, use either of the following methods:

- Use your browser to open the samples index page and select the servlet:

  `http://your.server.name/IBMWebAS/samples/`
- Use your browser to open the URL of the servlet form:

  `http://your.server.name/IBMWebAS/samples/FormDisplayServlet/FormDisplayServletForm.html`

FormProcessingServlet

This servlet reads and processes registration data from an HTML form, returns a customized page, and writes registration data to a file. To run the servlet:

1. Copy the seminar.txt file from the *applicationserver_root*\samples\FormProcessingServlet directory to the HTML document root directory on your Web server.
2. To run the sample, use either of the following methods:
   - Use your browser to open the samples index page and select the servlet:

     `http://your.server.name/IBMWebAS/samples/`
   - Use your browser to open the URL of the servlet form:

     `http://your.server.name/IBMWebAS/samples/FormProcessingServlet/FormProcessingServletForm.html`

XtremeTravel

This Web application is a travel service. The application determines the page (called the *referrer*) that you were on before coming to the Web site. It uses this information to determine your area of interest and links you to the appropriate travel packages. A Web site administrator can post bulletins to specific groups of pages and all the visitors on this site can exchange messages.

To run this application, use either of the following methods:

- Use your browser to open the samples index page and select the Web application:

  `http://your.server.name/IBMWebAS/samples/`
- Use your browser to open the URL of the servlet form:

  `http://your.server.name/IBMWebAS/samples/XtremeTravel/index.html`

Running the database samples

Five of the samples demonstrate database access. Before you can use these samples, install IBM DB2 and populate the appropriate database for the sample. The following sections contain step-by-step instructions for each sample.

JDBCServlet

The JDBCServlet reads input from an HTML form to build an SQL query, makes a JDBC connection to the IBM DB2 sample database, processes the query result set, and returns it to the Web page. To run the sample:

1. Install IBM DB2. Follow the instructions in the DB2 documentation.
2. Create the sample database that comes with DB2.
3. If your Application Server is running under AIX or Sun Solaris, refer to the DB2 documentation on running Java programs for instructions on setting the DB2INSTANCE environment variable and related settings.
4. Add the DB2 JDBC driver to the Application Server Java classpath:
   a. Open Application Server Manager at http://*your.server.name*:9090/.
   b. Select Setup Basic.
   c. On the Basic tab, ensure that the DB2 sqllib\java\db2java.zip is in the Java Classpath field.

5. Edit the login.properties file to give JDBCServlet access to DB2:
   a. Use a text editor to open *applicationserver_root*\samples\login.properties.
   b. Set JDBCServlet.dbOwner to the owner of your DB2 sample database.
   c. If the user ID and password for DB2 are different from the user ID and password that your Web server runs under:
      1) Set JDBCServlet.dbUserid to the user ID for DB2. Depending on your setup, the dbUserid may differ from the dbOwner.
      2) Set JDBCServlet.dbPassword to the password for DB2.
   d. Save the login.properties file.
   e. Copy the login.properties file to the *applicationserver_root*\servlets directory.
6. To view the sample, use your browser to open the URL of the servlet form:
   `http://your.server.name/IBMWebAS/samples/JDBCServlet/JDBCServletForm.html`

IBMConnMgrTest

The IBMConnMgrTest servlet shows how to use the connection manager to manage JDBC connections to the IBM DB2 sample database. See "Chapter 4. Using the connection manager" on page 29 for a detailed discussion of the servlet.

To run the sample:

1. Install IBM DB2. Follow the instructions in the DB2 documentation.
2. Create the sample database that comes with DB2.
3. If your Application Server is running under AIX or Sun Solaris, refer to the DB2 documentation on running Java programs for instructions on setting the DB2INSTANCE environment variable and related settings.
4. Add the DB2 JDBC driver to the Application Server Java classpath:
   a. Open Application Server Manager at http://*your.server.name*:9090/.
   b. Select Setup Basic.
   c. On the Basic tab, ensure that the DB2 sqllib\java\db2java.zip is in the Java Classpath field.
5. Edit the login.properties file to give IBMConnMgrTest access to DB2:
   a. Use a text editor to open *applicationserver_root*\samples\login.properties.
   b. Set JDBCServlet.dbOwner to the owner of your DB2 sample database. This sample and the JDBCServlet use the same properties.
   c. If the user ID and password for DB2 are different from the user ID and password that your Web server runs under:
      1) Set JDBCServlet.dbUserid to the user ID for DB2 sample database. Depending on your setup, the dbUserid may differ from the dbOwner.
      2) Set JDBCServlet.dbPassword to the password for DB2 sample database.
   d. Save the login.properties file.
   e. Copy the login.properties file to the *applicationserver_root*\servlets directory.
6. To view the sample, use your browser to open the URL of the servlet form:
   `http://your.server.name/servlet/IBMConnMgrTest`

IBMDataAccessTest

The IBMDataAccessTest servlet shows how to use the data access JavaBeans for enhanced SQL access to the IBM DB2 sample database. See "Chapter 5. Using data access JavaBeans" on page 49 for a detailed discussion of the servlet.

To run the sample:

1. Install IBM DB2. Follow the instructions in the DB2 documentation.
2. Create the sample database that comes with DB2.
3. If your Application Server is running under AIX or Sun Solaris, refer to the DB2 documentation on running Java programs for instructions on setting the DB2INSTANCE environment variable and related settings.
4. Add the DB2 JDBC driver to the Application Server Java classpath:
   a. Open Application Server Manager at http://*your.server.name*:9090/.
   b. Select Setup Basic.
   c. On the Basic tab, ensure that the DB2 sqllib\java\db2java.zip is in the Java Classpath field.
5. Edit the login.properties file to give IBMDataAccessTest access to DB2:
   a. Use a text editor to open *applicationserver_root*\samples\login.properties.
   b. Set JDBCServlet.dbOwner to the owner of your DB2 sample database. This sample and JDBCSerlet use the same properties
   c. If the user ID and password for DB2 are different from the user ID and password that your Web server runs under:
      1) Set JDBCServlet.dbUserid to the user ID for DB2. Depending on your setup, the dbUserid may differ from the dbOwner.
      2) Set JDBCServlet.dbPassword to the password for DB2.
   d. Save the login.properties file.
   e. Copy the login.properties file to the *applicationserver_root*\servlets directory.
6. To view the sample, use your browser to open the URL of the servlet form:
   `http://your.server.name/servlet/IBMDataAccessTest`

World of Money Bank (WOMBank)

This on-line banking service uses a JDBC connection to a DB2 database to maintain customer IDs, passwords, and account information. It does its own user authentication, allowing customers to register or log in, open accounts, deposit, withdraw or transfer funds, and view a log of their transactions. WOMBank uses UserProfile and session tracking to personalize the Web pages and track the application state.

To run the sample:

1. Install IBM DB2. Follow the instructions in the DB2 documentation.
2. If your Application Server is running under AIX or Sun Solaris, refer to the DB2 documentation on running Java programs for instructions on setting the DB2INSTANCE environment variable and related settings.
3. Add the DB2 JDBC driver to the Application Server Java classpath:
   a. Open Application Server Manager at http://*your.server.name*:9090/.
   b. Select Setup Basic.
   c. On the Basic tab, ensure that the DB2 sqllib\java\db2java.zip is in the Java Classpath field.
4. Edit the login.properties file to give WOMBank access to DB2:
   a. Use a text editor to open *applicationserver_root*\samples\login.properties.
   b. Set WomBank.dbOwner to the owner of the WOMBank database (womdb) that you will create in a later step.

c. If the user ID and password for DB2 from the user ID and password that your Web server runs under:
   1) Set WomBank.dbUserid to the user ID for DB2. Depending on your setup, the dbUserid may differ from the dbOwner.
   2) Set WomBank.dbPassword to the password for DB2.
d. Save the login.properties file.
e. Copy the login.properties file to the *applicationserver_root*\servlets directory.

5. In the DB2 Command Line Processor (CLP). To populate the database, type:

```
quit
cd applicationserver_root\samples\WomBank
womdb.bat
```

6. Use the Application Server Manager at http://*your.server.name*:9090/ to configure User Profile. Refer to the online Help for instructions on how to complete the page.
7. To view the sample, use your browser to open the URL of the servlet form:

```
http://your.server.name/IBMWebAS/samples/WomBank/enter.html
```

TicketCentral

This ticket service uses a JDBC connection to a DB2 database. It searches several pre-loaded database tables to find ticket information, and reads and writes other tables to authenticate customers and maintain their data. Registered customers can log in, specify ticket search information, select tickets to purchase, and use a credit card to pay for them. TicketCentral uses UserProfile and session tracking to personalize the Web pages and track the application state.

To run the sample:

1. Install IBM DB2. Follow the instructions in the DB2 documentation.
2. If your Application Server is running under AIX or Sun Solaris, refer to the DB2 documentation on running Java programs for instructions on setting the DB2INSTANCE environment variable and related settings.
3. Add the DB2 JDBC driver to the Application Server Java classpath:
   a. Open Application Server Manager at http://*your.server.name*:9090/.
   b. Select Setup Basic.
   c. On the Basic tab, ensure that the DB2 sqllib\java\db2java.zip is in the Java Classpath field.
4. Edit your system CLASSPATH environment variable:
   - Add the DB2 sqllib\java\db2java.zip, if it is not already specified.
   - Add *applicationserver_root*\servlets.
5. Edit the login.properties file to give TicketCentral access to DB2:
   a. Use a text editor to open *applicationserver_root*\samples\login.properties.
   b. Set TicketCentral.dbOwner to the owner of the TicketCentral database (tcdb) that you will create in a later step.
   c. If the user ID and password for DB2 are different from the user ID and password that your Web server runs under:
      1) Set TicketCentral.dbUserid to the user ID for DB2. Depending on your setup, the dbUserid may differ from the dbOwner.
      2) Set TicketCentral.dbPassword to the password for DB2.
   d. Save the login.properties file.
   e. Copy the login.properties file to the *applicationserver_root*\servlets directory.

6. In the DB2 Command Line Processor (CLP). To populate the database, type:

   ```
   quit
   cd applicationserver_root\servlets
   tcdb.bat
   ```

7. Use the Application Server Manager at http://*your.server.name*:9090/ to configure User Profile. Refer to the online Help for instructions on how to complete the page.
8. To view the sample, use your browser to open the URL of the servlet form:

   ```
   http://your.server.name/IBMWebAS/samples/TicketCentral/enter.html
   ```

Using the Site Activity Monitor

The Site Activity Monitor is an applet for viewing real-time activity at any of the registered Web site applications. When an application is registered, you can use the Site Activity Monitor to see how many visitors are currently on each page in that application. You can also view any specific information about each visitor that the application might be tracking.

By default, the sample Web site applications are already registered in the AppPackage class. After you compile and run the sample Web site applications on your server, you will be able to view their activity locally with this applet. You can also register your own Web applications and then view their activity. The sections that follow describe how to register applications and invoke the Site Activity Monitor.

Registering with the Site Activity Monitor

By default, the sample Web site applications are already registered in the AppPackage class. Register your own applications using the methods provided in the AppPackage class in com.ibm.servlet.personalization.sam.

To register an application:

1. Name the application package with the setPackageName() method.
2. Specify the URI directory path of the HTML pages with the setPackageSubPath() method.
3. Identify each of the pages in that directory with the addLocatable() method.

For example, if your Human Resources department has a Web site with four HTML pages for employee benefit information in the HR/benefits/ directory, you might add this code to the first servlet the application executes:

```
AppPackage ap = new AppPackage();
ap.setPackageName(new String("EmpBenefits"));
ap.setPackageSubPath(new String("/HR/benefits/"));
ap.addLocatable(new String("main.html"));
ap.addLocatable(new String("health.html"));
ap.addLocatable(new String("leave.html"))
ap.addLocatable(new String("retire.html"));;
int i = AppPackage.addAppPackage(ap);
```

Invoking the Site Activity Monitor

There are two versions of the Site Activity Monitor: one uses the Java Development Kit Version 1.02 and the other uses the Java Development Kit Version 1.1.x. To

invoke the Site Activity Monitor applet on your own Web site, open the appropriate main page with your browser. For example, use the browser to open the URL:

`http://`*`your.server.name`*`/IBMWebAS/sam/sam11.html`

or

`http://`*`your.server.name`*`/IBMWebAS/sam/sam102.html`

where *your.server.name* is the hostname of your Web server. These main pages provide instructions for navigating Web sites, displaying session and user information, controlling the display, and sending messages to Web site visitors.

Appendix A. Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make them 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 that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter described 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:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA

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:

Site Counsel
IBM Corporation
P.O. Box 12195
3039 Cornwallis Road
Research Triangle Park, NC 27709-2195
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement.

This document is not intended for production use and is furnished as is without any warranty of any kind, and all warranties are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of Application Server.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

- AIX
- Domino Go Webserver
- IBM
- Lotus
- OS/390
- OpenEdition
- WebSphere

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Java and JavaBeans are trademarks of Sun Microsystems, Inc.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Appendix B. Supported NCSA tags

For the echo command, the Application Server supports the standard server-side include (SSI) environment variables and Common Gateway Interface (CGI) environment variables.

The SSI environment variables are:

- DOCUMENT_NAME

  The current filename
- DOCUMENT_URI

  The path to the document (such as, /docs/tutorials/index.shtml)
- QUERY_STRING_UNESCAPED

  The unescaped version of any search query the client sent, with all shell special characters escaped with the \ character
- DATE_LOCAL

  The current date and local time zone
- DATE_GMT

  The current date and local time zone in Greenwich mean time (GMT)
- LAST_MODIFIED

  The last date the current document was changed

The CGI environment variables are:

- SERVER_SOFTWARE

  The name and version of the server answering the request
- SERVER_NAME

  The server's IP address, hostname, or Domain Name Server (DNS) alias
- GATEWAY_INTERFACE

  The revision level of the CGI specification to which the server complies
- SERVER_PROTOCOL

  The name and revision level of the protocol used to format this request
- SERVER_PORT

  The port number to which the request was sent
- REQUEST_METHOD

  The method with which this request was made. For HTTP, GET, HEAD, POST, and so on are the methods.
- PATH_INFO

  The extra path information given by the client in this request. The extra information follows the virtual pathname of the CGI script.
- PATH_TRANSLATED

  The server provides a translated version of PATH_INFO, which takes the path and performs any virtual-to-physical mapping.
- SCRIPT_NAME

  The virtual path to the script being run. This variable is used for self-referencing URLs.
- QUERY_STRING

The information that follows the ? symbol in the URL request for a script

- REMOTE_HOST

  The hostname of the remote host sending the request. If the server does not have this information, the server should set REMOTE_ADDR and leave REMOTE_HOST unset.

- REMOTE_ADDR

  The IP address of the remote host sending the request.

- AUTH_TYPE

  The protocol-specific authentication method used to validate the user, if the server supports user authentication and the script is protected

- REMOTE_USER

  The username used for authentication, if the server supports user authentication and the script is protected

- REMOTE_IDENT

  If the HTTP server supports RFC 931 identification, the remote username retrieved from the server

- CONTENT_TYPE

  The content type of the data for queries that have information attached (such as HTTP POST and PUT)

- CONTENT_LENGTH

  The length of the content, as specified by the remote host

Glossary

For more information on terms used in this book, access the online IBM Software Glossary at URL:

`http://www.networking.ibm.com/nsg/nsgmain.htm`

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

M

N

O

P

Q

R

S

T

U

V

W

X

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-4767-00