Contents

This file was last modified on 8/12/99

About This Book

This book was last updated 8/12/99.

This book discusses how to enable and install Java servlets and JavaServerPages (JSP) in Enterprise Server 4.0.

This book has the following chapters and appendices:

- Chapter 1, “Using Servlets and JavaServerPages.”

  This chapter discusses how to enable and install servlets and JavaServerPages in Enterprise Server 4.0. It explains how to specify settings for servlets and for the JRE, JDK and JVM by using the Server Manger interface or by editing configuration files.

- Chapter 2, “Servlet and JSP Examples.”

  This chapter discusses example servlets and JSP.

- Appendix A, “Session Managers.”

  This appendix discusses the session managers provided with Enterprise Server and gives an overview of a sample session manager that you can use to define your own session managers.

- Appendix B, “Servlet Settings in obj.conf.”

  This appendix discusses how the configuration file `obj.conf` changes depending on the settings for servlets and JSP.

- Appendix C, “servlets.properties and rules.properties.”

  This appendix discusses the `servlets.properties` file, which contains configuration information for servlets, and the `rules.properties` file which defines virtual paths for servlets.

- Appendix D, “JVM Configuration.”

  This appendix discusses which configuration files to edit if you want to manually specify JVM configuration information.

- Appendix E, “Remote Servlet Debugging.”

  This appendix discusses how to enable remote debugging for servlets.

- Appendix F, “Remote Servlet Profiling.”

  This appendix discusses how to enable remote profiling for servlets.

- Appendix G, “API Clarifications.”

  This chapter discusses methods in the Servlets API that behave marginally differently in Enterprise Server than specified in the Sun Microsystems’ Servlets API documentation or where the behavior documented by Sun Microsystems is ambiguous.

Chapter

1

Using Servlets and JavaServerPages

Enterprise Server 4.0 supports servlets and JavaServer Pages (JSP). This chapter gives a brief overview of servlets and JavaServer Pages and discusses how to enable and configure them in Enterprise 4.0.

The sections in this chapter are:

- Servlets
- JavaServer Pages
- What Does the Server Need to Run Servlets and JSP?
- Serving Servlets and JSP
- Using the Server Manager Interface to Specify Servlet Behavior and Attributes
- Activating Servlets and JSP
- Configuring General Servlet Properties
- Registering Servlet Directories
- Registering Individual Servlets
- Specifying Servlet Virtual Paths
- Configuring JRE/JDK Paths
- Deleting Version Files
- Configuring JVM

Servlets

Java servlets are server-side Java programs that web servers can run to generate content in response to a client request in much the same way as CGI programs do. Servlets can be thought of as applets that run on the server side without an interface. Servlets are invoked through URL invocation

Netscape Enterprise Server 4.0 includes support for JavaSoft's Servlet API at the level of the Java Servlet Development Kit (JSDK) 2.1.

To develop servlets, use Sun Microsystems' Java Servlet API. For information about using the Java Servlet API, see the documentation provided by Sun Microsystems at:

`http://www.javasoft.com/products/servlet/index.html`

Netscape Enterprise Server 4.0 includes all the files necessary for developing Java Servlets. The `servlets.jar` file is in the ES4.0 installation directory at:

`/bin/https/jar`

When compiling servlets, make sure the `servlets.jar` file is accessible variable to your Java compiler.

JavaServer Pages

Enterprise Server 4.0 supports JavaServerPages (JSP) to the level of JSP API 0.92 compliance.

A JavaServerPage (JSP) is a page much like an HTML page, that can be viewed in a web browser. However, as well as containing HTML tags, it can include a set of JSP tags that extend the ability of the web page designer to incorporate dynamic content in a page. These tags provide functionality such as displaying property values and using simple conditionals.

One of the main benefits of JavaServer Pages is that, like HTML pages, they do not need to be compiled. The web page designer simply writes a page that uses HTML and JSP tags, and puts it on their web server. The web page designer does not need to learn how to define Java classes or use Java compilers.

JSP pages can access full Java functionality in the following ways:

- by embedding Java code directly in scriptlets in the page

- by accessing Java beans
- by using server-side tags that include Java servlets

Both beans and servlets are Java classes that need to be compiled, but they can be defined and compiled by a Java programmer who then publishes the interface to the bean or the servlet. The web page designer can access a pre-compiled bean or servlet from a JSP page without having to do any compiling themselves.

For information about creating JavaServer Pages, see Sun Microsystem's JavaServer Pages 0.92 spec in the build in the manuals/https/servlets/jsp092 subdirectory .

For information about Java Beans, see Sun Microsystem's JavaBeans web page at:

`http://www.javasoft.com/beans/index.html`

What Does the Server Need to Run Servlets and JSP?

Enterprise Server 4.0 includes the Java Runtime Environment (JRE) but not the Java Development Kit (JDK). The server can run servlets using the JRE, but it needs the JDK to run JSP. If you want to run JSP, you must tell the Enterprise Server to use a custom JDK.

Enterprise Server 4.0+ requires you to use official versions of JDK, with different platforms requiring different versions. For example, Sun Solaris requires JDK1.2 or higher; HP-UX requires JDK 1.1 (C.01.17.01 or any higher 1.1 version); and Windows NT requires a JDK of 1.2.2 or higher. Check the *Installation and Migration Guide* and the latest release notes for updates on required JDK versions.

Note On Sun Solaris, the JRE included is the JRE 1.2.2 reference implementation from JavaSoft. For performance, it is recommended to use the latest SunSoft production release of JDK, currently 1.2.1_03.

JDK 1.2 (and other JDK versions) are available from Sun Microsystems at:

`http://www.javasoft.com/products/jdk/1.2/`

You can specify the path to the JDK in either of the following ways:

- You can specify the path during the server installation process.

  When you install Enterprise Server 4.0, one of the dialog boxes in the installation process asks if you want to use a custom Java Development Kit (JDK), and if so, you can specify the path to it.

- You can specify it after the server is installed.

  To specify the path to the JDK, use the “Configure JRE/JDK Paths” page in the Servlets tab of the Server Manager, as discussed in the section "Configuring JRE/JDK Paths."

Whether you specify the path to the JDK during installation or later, the path is the folder in which you installed the JDK.

Serving Servlets and JSP

Enterprise Server 4.0 includes an appropriate version of the Java runtime environment (JRE) for running servlets. For the server to be able to serve JSP, you must specify a path to a Java Development Kit (JDK) as discussed in the section "What Does the Server Need to Run Servlets and JSP?."

For the server to serve servlets and JSP, servlet activation must be enabled. (See the section "Activating Servlets and JSP" for details.)

When the servlet engine is activated, you have a choice of two ways to make a servlet accessible to clients:

- Put the servlet class file in a directory that has been registered with the Enterprise Server as a servlet directory. For more information, see "Registering Servlet Directories."

- Define a servlet virtual path for the servlet. In this case, the servlet class can be located anywhere in the file system or even reside on a remote machine. For more information, see "Specifying Servlet Virtual Paths."

No special steps are needed to enable JSP pages other than making sure that JSP is activated on the Enterprise Server. So long as JSP activation is enabled, the Enterprise Server treats all files with a `.jsp` extension as JavaServer Pages. (Do not put JSP files in a registered servlets directory, since the Enterprise Server expects all files in a registered servlet directory to be servlets.)

In detail, to enable the Netscape Enterprise Server to serve servlets and JSP pages, do the following:

1. Activating Servlets and JSP. (This is the only step needed to enable JSP.)
2. Configuring General Servlet Properties
3. Registering Servlet Directories
4. Registering Individual Servlets if Needed
5. Specifying Servlet Virtual Paths if Desired
6. Configuring JVM if Necessary

Using the Server Manager Interface to Specify Servlet Behavior and Attributes

In the Enterprise Server 4.0 Server Manager interface, you can use the Servlets tab to specify settings for servlets. For information about using the interface for working with servlets, see the following subsections in the "Servlets Tab" section of Appendix F, "The Enterprise Server User Interface" in the "Enterprise Server Administration Guide:"

- The Enable/Disable Servlets Page
- Configure JRE/JDK Page
- The Servlet Directory Page
- The Configure Global Attributes for Servlets Page
- The Configure Servlet Attributes Page
- The Configure Servlet Virtual Path Translation Page
- The Configure JVM Attributes Page
- The Delete Version Files Page

Activating Servlets and JSP

To enable and disabled servlets and JSP in Enterprise Server 4.0, use the Servlets>Enable/Disable Servlets page in the Server Manager interface.

If servlets are enabled, JSP can be enabled or disabled. However, if you disable servlets, JSP is automatically also disabled. In this case, if you enable servlets later, you will need to re-enable JSP also if desired.

To enable servlets programmatically, add the following lines to `obj.conf`. These directives first load the shared library containing the servlet engine, which is in `NSServletPlugin.dll` on Windows NT or `NSServletPlugin.so` on Unix. Then they initialize the servlet engine.

```
Init shlib="server_root/bin/https/bin/NSServletPlugin.dll/so"
funcs="NSServletEarlyInit,NSServletLateInit,NSServletNameTrans,NSServle
tService" shlib_flags="(global|now)" fn="load-modules"

Init EarlyInit="yes" fn="NSServletEarlyInit"

Init LateInit="yes" fn="NSServletLateInit"
```

In the default object in `obj.conf`, add the following `NameTrans` directive:

```
NameTrans fn="NSServletNameTrans" name="servlet"
```

By default, regardless of whether servlets are enabled or disabled, the file `obj.conf` contains additional objects with names such as `servlet`, `jsp`, and `ServletByExt`. Do not delete these objects. If you delete them, you will no longer be able to activate servlets through the Server Manager.

Configuring General Servlet Properties

You can specify the following servlet properties:

- Startup Servlets -- servlets to load when the Enterprise Server starts up.
- Session Manager -- the session manager for servlets, if applicable. For more information about the session manager, see Appendix A, “Session Managers.”
- Reload Interval -- the time period that the server waits before re-loading servlets and JSPs if they have changed on the server. The default value is 5.

You can set these attributes interactively in the Servlets>Configure General Servlet Properties page in the Server Manager interface. Alternatively, you can edit the configuration file `servlet.properties` in the server’s `config` directory.

The following code shows an example of the settings in `servlet.properties`:

```
# General properties:
servlets.startup=hello
servlets.config.reloadInterval=5
servlets.config.docRoot=C:/Netscape/Server4/docs
servlets.sessionmgr=com.netscape.server.http.session.SimpleSessionManager
```

Registering Servlet Directories

One of the ways to make a servlet accessible to clients is to put it into a directory is registered with the Enterprise Server as a servlet directory. Servlets in registered servlet directories are dynamically loaded when needed. The server monitors the servlet files and automatically reloads them on the fly as they change.

For example, if the `SimpleServlet.class` servlet is in the `servlet` subdirectory of the server's document root directory, you can invoke the servlet by pointing the web browser to:

`http://`*`your_server`*`/servlet/SimpleServlet`

You can register any number of servlet directories for the Enterprise Server. Initially, the Enterprise Server has a single servlet directory, which is *`server_root`*`/docs/servlet/` (For example, `d:/netscape/server4/docs/servlet`.)

The Enterprise Server expects all files in a registered servlet directory to be servlets. The server treats any files, including applets, in that directory that have the `.class` extension as servlets. The Enterprise Server does not correctly serve other files, such as HTML files or JSPs, that reside in that directory.

The server can have multiple servlet directories. You can map servlet directories to virtual directories if desired. For example, you could specify that `http://poppy.my_domain.com/products/` invokes servlets in the directory *`server_root`*`/docs/servlet/january/products/servlets/`.

To register servlet directories and to specify their URL prefixes, use the Servlets>Servlet Directory page in the interface.

Alternatively, you can register servlet directories by adding appropriate `NameTrans` directives to the default object in the file `obj.conf`, such as:

```
NameTrans fn="pfx2dir" from="/servlets"
dir="d:/netscape/server4/docs/servlet/january/products/servlets/"
name="ServletByExt"
```

Registering Individual Servlets

The Enterprise Server treats any file in a registered servlet directory as a servlet. There is no need to register individual servlets that reside in these directories unless either of the following criteria apply:

- The servlet takes input parameters that are not passed through the request URL.
- You want to set up virtual URLs for the servlet.

If either of these conditions is true, register the individual servlet by using the Servlets>Configure Servlet Attributes page in the Server Manager interface. Alternatively you can edit the file `servlet.properties` to add an entry for the servlet.

When registering an individual servlet, specify the following attributes:

- Servlet Name -- The Enterprise Server uses this value as a servlet identifier to internally identify the servlet. (This identifier is not part of the URL that is used to invoke the servlet, unless by coincidence the identifier is the same as the class code name.)

- Servlet Code (class name) -- the name of the class file. You do not need to specify the `.class` extension.

- Servlet Classpath -- This is the absolute pathname or URL to the directory or zip/jar file containing the servlet. The classpath can point anywhere in the file system. The servlet classpath may contain a directory, a `.jar` or `.zip` file, or a URL to a directory.(You cannot specify a URL as a classpath for a zip or jar file.)

  If the servlet classpath is not a registered servlet directory, you must additionally provide a servlet virtual path for it (as discussed in" Specifying Servlet Virtual Paths") to make the servlet accessible to clients.

  Enterprise Server supports the specification of multiple directories, jars, zips, and URLs in the servlet classpath.

- Servlet Args -- a comma delimited list of additional arguments for the servlet if required.

For example, in Figure 1.1, the Servlets>Configure Servlet Attributes page of the Server Manager interface shows configuration information for a servlet whose class file `buynow1A` resides in the directory `D:/Netscape/server4/docs/`

`servlet/buy/`. (Note that the final `/` is omitted in the interface.) This servlet is configured under the name `BuyNowServlet`. It takes additional arguments of `arg1=45`, `arg2=online`, `arg3="quick shopping"`.

Figure 1.1 Configuring attributes for an individual servlet

The following code shows an example of the configuration information for the same servlet in `servlet.properties`:

```
servlet.BuyNowServlet.classpath=D:/Netscape/server4/docs/servlet/buy
servlet.BuyNowServlet.code=BuyNow1A
servlet.BuyNowServlet.initArgs=arg1=45,arg2=online,arg3="quick shopping"
```

Note that you can specify multiple values as the servlet classpath if needed.

Specifying Servlet Virtual Paths

One way of making servlets available to clients is to put them in registered servlet directories. Another way is to define servlet virtual paths for individual servlets. For example, you could specify that the URL

`http://poppy.my_domain.com/plans/plan1`

invokes the servlet defined in the directory

`server_root``/docs/servlets/plans/releaseA/planP2Version1A.class`

You can set up servlet virtual paths for servlets that reside anywhere, be it on a local or remote file system, and be it in or out of a registered servlet directory.

To specify a servlet virtual path, use the Servlets>Configure Servlet Virtual Path Translation page in the Server Manager interface. In this page, specify the virtual path name and the servlet name. You can alternatively manually edit the `rules.properties` configuration file to add a servlet virtual path. Only servlets for which a virtual path has been set up can use initial arguments (See "GenericServlet.getInitParameter() and getInitParameterNames()" for iinformation about initial arguments.)

Before using a servlet virtual path, a servlet identifier must be added for the servlet in the Servlets>Configure Servlet Attributes page of the interface (or in the `servlets.properties` configuration file).

Virtual Servlet Path Example

This example discusses how to specify that the logical URL:

`http://poppy.my_domain.com/plans/plan1`

invokes the servlet defined in

`server_root``/docs/servlet/plans/releaseA/planP2Version1A.class.`

1. Specify the servlet identifier, class file, and class path.

In the Servlets>Configure Servlet Attributes page in the interface, do the following:

- in the Servlet Name field, enter an identifier for the servlet, such as `plan1A`. (Notice that this is not necessarily the same as the class file name).
- in the Servlet Code field, enter the name of the class file, which is `planP2Version1A`. Don't specify any directories. The `.class` extension is not required.
- in the Servlet Class Path field, enter the absolute path name for the directory, jar or zip file where the servlet class file resides, or enter a URL for a directory. In this example, you would enter *`server_root`*`/docs/servlet/plans/releaseA`. (For example: `D:/netscape/server4/docs/servlet/plans/releaseA`.)
- in the Servlet Args field, enter the additional arguments that the servlet needs, if any. (This example does not use extra arguments.)

Figure 1.2 shows the settings in the interface.

Save the changes.

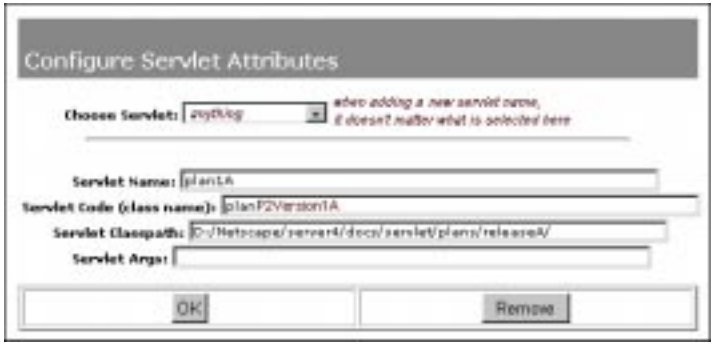

Figure 1.2 Specifying the servlet name, code, and class path

To make this change programmatically, add the following lines to the configuration file `servlet.properties`:

```
servlet.plan1A.classpath=D:/Netscape/server4/docs/servlet/plans/
releaseA/

servlet.plan1A.code=planP2Version1A
```

2. Specify the virtual path for the servlet.

In the Servlets>Configure Servlet Virtual Path Translations page, do the following:

- In the Virtual Path field, enter the virtual path name. Note that the server name is implied as a prefix, so in this case you would only need to enter `/plans/plan1` to specify the virtual path `http://poppy.mcom.com/plans/plan1`.
- In the Servlet field, enter the identifier for the servlet that is invoked by this virtual path. This is the servlet identifier that you specified in the Configure Servlet Attributes page, which in this case is `plan1A`.

Save the changes.

Figure 1.3 shows the settings in the interface.

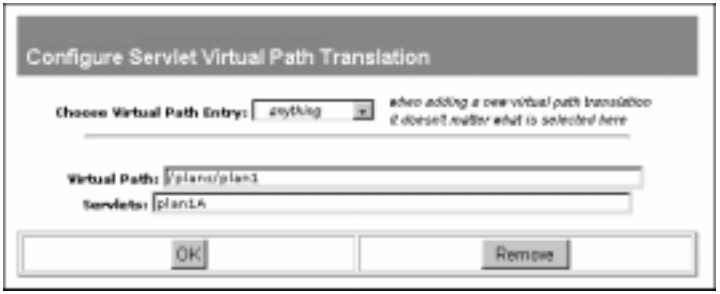

Figure 1.3 Adding a virtual path

To do this programmatically, add the following line to `rules.properties`:

```
/plans/plan1=plan1A
```

After this virtual servlet path has been established, if a client sends a request to the server for the URL `http://poppy.my_domain.com/plans/plan1`, the server sends back the results of invoking the servlet in *`server_root`*`/docs/servlet/plans/releaseA/plan2PVersion1A.class`.

Configuring JRE/JDK Paths

When you install Enterprise Server 4.0, you can choose to install the Java Runtime Environment (JRE) or you can specify a path to the Java Development Kit (JDK).

The server can run servlets using the JRE, but it needs the JDK to run JSP. The JDK is not bundled with the Enterprise Server, but you can download it for free from Sun Microsystems at:

`http://www.javasoft.com/products/jdk/1.2/`

Enterprise Server 4.0+ requires you to use an official version of JDK1.2 on Solaris and NT. On HP, AIX and IRIX use JDK 1.1.

Regardless of whether you choose to install the JRE or specify a path to the JDK during installation, you can tell the Enterprise Server to switch to using either the JRE or JDK at any time, by using the “Configure JRE/JDK Paths” page in the Servlets tab. You can also change the path to the JDK in this page.

This page has the following fields:

- Change JRE or JDK

  Select either the JRE or JDK radio button as desired.

- Path

  Enter the path for the JRE or JDK. This is the directory where you installed the JRE or JDK.

- Classpath

  The class path includes the paths to the directories and jar files needed to run the servlet engine, the servlet examples, and any other paths needed by servlets that you add. Values are separated by semicolons. You can add new values to the existing class path, but don't delete the existing value since it includes paths that are essential for servlet operation.

It is easiest to use the Server Manager interface to switch between the JRE and the JDK, but you can also make the change programmatically, as follows:

- On Unix:

  Edit the file *server_root*/https-admserv/start.jre.

  If the server is currently using the JRE, this file has a variable NSES_JRE. To enable the server to use a JDK, add the variable NSES_JDK whose value is the JDK directory. You'll also need to change the value of the NSES_JRE variable.

  If you're using JDK 1.2 or greater, NSES_JDK should point to the installation directory for the JDK, while NSES_JRE should point to the JRE directory in the installation directory for JDK (that is, *jdk_dir*/jre). For JDK 1.1.x, NSES_JDK and NSES_JRE should both point to the installation directory for the JDK.

- On Windows NT:

  Add the path to the Java libraries to the extrapath setting in magnus.conf.

  Edit the NSES_JDK and NSES_JRE variables in the registry HKEY_LOCAL_MACHINE/SOFTWARE/Netscape/Enterprise/4.0/. If the server is enabled to use the JDK, both these variables are needed. If the server is to use the JRE, only the NSES_JRE variable should be set. If you're using JDK 1.2 or greater, NSES_JDK should point to the installation directory for the JDK, while NSES_JRE should point to the JRE directory in the

installation directory for JDK (that is, *jdk_dir*/jre). For JDK 1.1.x, NSES_JDK and NSES_JRE should both point to the installation directory for the JDK.

Deleting Version Files

The server uses two directories to cache information for JavaServerPages (JSP) and servlets:

- ClassCache

  When the server serves a JSP page, it creates a .java and a .class file associated with the JSP and stores them in the JSP class cache, in a directory structure under the ClassCache directory.

- SessionData

  If the server uses the MMapSessionManager session manager, it stores persistent session information in the SessionData directory. (For more information about session managers, see Appendix A, "Session Managers.")

Each cache has a version file containing a version number that the server uses to determine the structure of the directories and files in the caches. You can clean out the caches by simply deleting the version file.

When the server starts up, if it does not find the version files, it deletes the directory structures for the corresponding caches and re-creates the version files. Next time the server serves a JSP page, it recreates the JSP class cache. The next time the server serves a JSP page or servlet while using MMapSessionManager session manager, it recreates the session data cache.

If a future upgrade of the server uses a different format for the caches, the server will check the number in the version file and clean up the caches if the version number is not correct.

You can delete the version files simply by deleting them from the ClassCache or SessionData directories as you would normally delete a file or you can use the Servlets>Delete Version Files page in the Server Manager to delete them. After deleting one or both version files, be sure to restart the Enterprise Server to force it to clean up the appropriate caches and to recreate the version files before the server serves any servlets or JSPs.

Configuring JVM

If necessary, you can configure parameters for JVM either by using the Servlets>Configure JVM Attributes page in the Server Manager interface, or by editing `JVM.conf` (or `JVM11.conf` or `JVM12.conf`, depending on which version of JVM is being used).

The default settings in Enterprise Server for JVM are suitable for running servlets. However, there may be times when you want change the settings. For example, if a servlet or bean file uses a JAR file, add the JAR location to the Classpath variable. To enable the use of a remote profiler, set the OPTITDIR and Profiler variables.

The JVM parameters you can set are:

- Option -- You can set any options allowed by the vendor's JVM.
- Profiler -- If you are using the Optimizeit! 3.0 profiler from Intuitive Systems, enter the value `optimizeit`. For more information about this optimizer, see the section Appendix F, “Remote Servlet Profiling.”
- OPTITDIR -- If you are using the Optimizeit! 3.0 profiler from Intuitive Systems, enter the pathname for the directory where Optimizeit! resides, for example, `D:/App/IntuitiveSystems/OptimizeIt30D`. For more information about this optimizer, see the section "Appendix F, “Remote Servlet Profiling.”."
- Minimum Heap Size -- determines the minimum heap size allocated for Java.
- Maximum Heap Size -- determines the maximum heap size allocated to Java.
- Compiler -- You can specify options to turn on and off JIT (just-in-time compiler). See your JVM documentation for details.
- Classpath -- Enter additional classpath values as needed. For example, if a JSP uses a bean that is packaged in a JAR, add the JAR path to the classpath.

  The classpath must not include backslashes in directory names. If you use backslashes in the directory path in the interface, the system automatically converts the backslashes to forward slashes. However, if you edit the `jmv.conf` (or `jvm11.conf` or `jvm12.conf`) file, do not use backslashes in directory names.

- Enable Class GC -- Specifies whether or not to enable class garbage collection. The default is yes.
- Verbose Mode -- Determines whether the JVM logs a commentary on what it is doing, such as loading classes. The commentary appears in the error log.
- Enable Debug -- You can enable or disable remote debugging. The default is disabled. For more information about remote debugging, see the section Appendix E, “Remote Servlet Debugging.”

Chapter

2

Servlet and JSP Examples

This chapter discusses some Servlet and JSP examples. It has the following sections:

- Examples Shipped with Enterprise Server 4.0
- Servlet Examples
- JSP Examples

Examples Shipped with Enterprise Server 4.0

Enterprise Server4.0 comes with a set of example servlets and JSP files. You can find them at the following location:

server_root/plugins/samples/servlets

This directory contains the following directories:

- `beans` -- Contains example Java Bean files.
- `bookstore` -- Contains files for an online bookstore example. This example contains both servlets and JSPs.
- `edemo` -- Contains files for a general online store front. This example contains both servlets and JSPs.

- `jsp` -- Contains subdirectories that each contain example JavaServer Page examples.
- `make` -- Contains example make files for servlets. These are common makefiles containing rules that are included by all other makefiles.
- `servlets` -- Contains subdirectories that each contain example Java and makefiles for servlet examples.
- `sessions` -- Contains code for `SimpleSessionManager.java`, which is the default servlet session manager when the Enterprise Server runs in single process mode. This directory also contains the code for `SimpleSession.java`, which defines session objects, which are the sessions managed by `SimpleSessionManager`. The source code for `SimpleSessionManager` and `SimpleSession` are provided for you to use as the starting point for defining your own session managers if desired. For more information about sessions and session managers, see Appendix A, "Session Managers."

Servlet Examples

This section discusses two servlet examples as follows:

- A Simple Servlet Example -- generates a very simple page to be displayed in a web browser.
- Example of a Servlet that Parses Input Parameters -- this servlet is used as a form action.

You can find additional examples in the directory *`server_root`*`/plugins/samples/servlets/servlets`.

These examples are simple, introductory examples. For information about using the Java Servlet API, see the documentation provided by Sun Microsystems at:

`http://www.javasoft.com/products/servlet/index.html`

A Simple Servlet Example

The following example code defines a very simple servlet. This example is the `SimpleServlet` example in the *`server_root`*`/plugins/samples/servlets/Simple1` directory.

This servlet generates an HTML page that says "`This is output from the servlet.`" as shown in Figure 2.1

Figure 2.1 Output from SimpleServlet.class

This example defines the main servlet class as a subclass of `HttpServlet` and implements the `doGet` method. The code is shown below:

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

/**
* This is a simple example of an HTTP Servlet that outputs
   a very simple web page.
*/

public class SimpleServlet extends HttpServlet
{
/**
* Handle the GET method by building a simple web page.
*/

public void doGet (
   HttpServletRequest request,
   HttpServletResponse response)
   throws ServletException, IOException

{
   PrintWriter out;
   String title = "Simple Servlet Output";

   // Set content type and other response header fields first
   response.setContentType("text/html");

   // Then write the data of the response
   out = response.getWriter ();

   out.println("<HTML><HEAD><TITLE>");
   out.println(title);
   out.println("</TITLE></HEAD><BODY>");
   out.println("<H1>" + title + "</H1>");
   out.println("<P>This is output from SimpleServlet.");
   out.println(getServletInfo());
```

```
    out.println("</BODY></HTML>");
    out.close();
    }
}
```

Example of a Servlet that Parses Input Parameters

This example demonstrates how to use a servlet as a form action. This example involves the following components:

- `servform.htm` - a web page containing a form as shown in Figure 2.2.
- `servlet1` -- a servlet that responds to the form.

servform.htm

This web page contains a form with the following elements:

- a text field named `companyname`
- three checkboxes named `hosting`, `design` and `javadev`
- a radio button named `numberofpeople`. The four possible values are `oneplus`, `tenplus`, `hundredplus` and `thousandplus`
- a submit button

Figure 2.2 shows an example of the form in a web page:

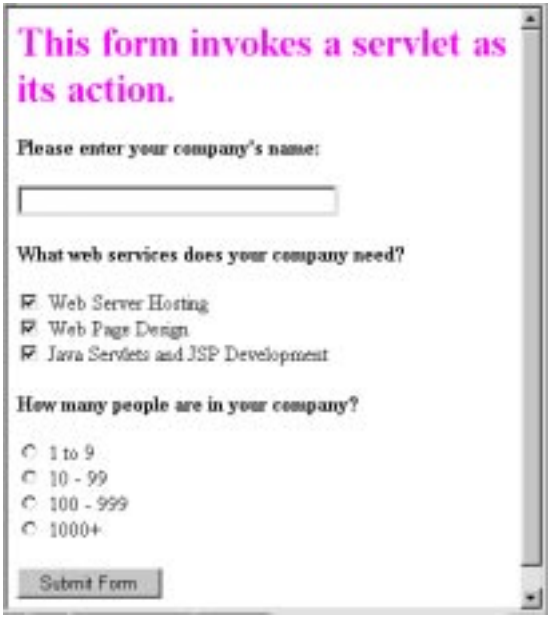

Figure 2.2 This form invokes a servlet as its action

The form's method is `GET` and the action is `servlet1.class`.

```
<FORM METHOD=GET ACTION="servlet/servlet1.class">
```

Click the following link to see the form: (View the source to see the source code).

`servform.htm`

servlet1

This servlet parses the input parameters received from the form. It displays the query string that invoked it, and then parses and displays the input parameters received from the form. Finally it constructs a message that is customized according to the input parameters received. An example is shown in Figure 2.3.

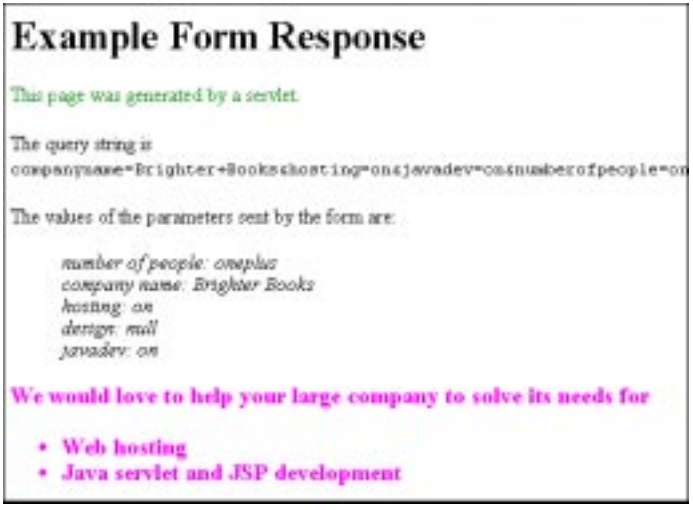

Figure 2.3 An example response from the servlet

This class implements the `doGet()` method, since it will be invoked by a form that uses the `GET` method. If the form used the `POST` method, the class would need to implement the `doPost()` method.

The source code is shown below.You can also access it through this link: `servlet1.java`.

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

/**
* This is a simple example of an HTTP Servlet that responds to input
* parameters such as form input.
*/

public class servlet1 extends HttpServlet
{
/**
* The doGet method parses the input parameters and constructs
* an output page based on the information received from the form.
*/

public void doGet (
   HttpServletRequest      request,
   HttpServletResponse     response
```

```
) throws ServletException, IOException

{
   PrintWriter out;
   String title = "Example Form Response";

   // Set content type and other response header fields first.
   response.setContentType("text/html");

   // Get an output stream.
   out = new java.io.PrintWriter(response.getOutputStream ());

   // Print the HTML, HEAD, and TITLE tags.
   out.println("<HTML><HEAD><TITLE>");
   out.println(title);
   out.println("</TITLE></HEAD><BODY>");
   out.println("<H1>" + title + "</H1>");
   out.println("<P><FONT color=green>This page was generated by " +
   "a servlet.</FONT></P>");

   // Print the query string just for informational purposes.
   String queryString = request.getQueryString();
   out.println("<P>The query string is <CODE>" + queryString +
   "</CODE>");

   // Extract the values of the parameters sent by the form.
   // If a parameter does not exist, getParameter() returns null.

   String numberofpeople = request.getParameter("numberofpeople");
   String companyname = request.getParameter("companyname");
   String companysize = "";
   String design = request.getParameter("design");
   String javadev = request.getParameter("javadev");

   // Print out the input parameters

   out.println("<P>The values of the parameters sent by the form are:");
   out.println("<BLOCKQUOTE><I>number of people: " + hosting);
   out.println("<BR>company name: " + hosting);
   out.println("<BR>hosting: " + hosting);
   out.println("<BR>design: " + design);
   out.println("<BR>javadev: " + javadev + "</I></BLOCKQUOTE>");

   // Construct a customized message encouraging
   // the viewer to use our company for its web needs.

   // First test if any skills are needed.
   // If not, construct a generic message.
   // If skills are needed, create an unordered list with
   // a bullet item for each skill selected.

   String skillsneeded = "";
   if ((hosting == null) && (design == null) && (javadev == null))
   {skillsneeded = " all your web server requirements.";
   }
```

```
    else
     { skillsneeded = "<UL>";
       if ((hosting != null) && (hosting.equals("on")))
          skillsneeded += "<LI>Web hosting";
       if ((design != null) && (design.equals("on")))
          skillsneeded +=  "<LI>Web page design";
       if ((javadev != null) && (javadev.equals("on")))
          skillsneeded += "<LI>Java servlet and JSP development ";
       skillsneeded = skillsneeded + "</UL>";
    }

    // Figure out what size company sent the form.
    // Choices are small, small to medium-size, medium-size and large.
    if (numberofpeople == null)
       numberofpeople = "size unknown";
    else if (numberofpeople == "oneplus")
       companysize = "small ";
    else if (numberofpeople == "tenplus")
       companysize = "small to medium-size ";
    else if (numberofpeople == "hundredplus")
       companysize = "medium-size ";
    else companysize = "large";

    // Print a message tailored to the company that sent the form.

    out.println("<H3><FONT color=magenta>" +
    "We would love to help your " + companysize + " company" +
    " to solve its needs for " + skillsneeded + "</FONT></H3>");

    // Print the closing tags in the HTML page
    // and close the output stream.
    out.println("</BODY></HTML>");
    out.close();

// end the method
}

// end the class
}
```

Running the Example

To run this example, save `servform.htm` to the directory of your choice in or under the server's document root. Create a subdirectory called `servlet` in the directory where you save `servform.htm`. Save `servform.class` to the new `servlet` directory. Register the new `servlet` directory as a servlets directory, as discussed in the section "Registering Servlet Directories."

To view the form, open `servform.htm` using an http URL. For example, if you put `servform.htm` in the document root directory, open it with the URL `http://`*`your_server`*`/servform.htm`.

JSP Examples

This section presents the following JSP examples:

- JSP that Accesses the Request Object. This example is self-contained -- it uses no external beans or Java classes.
- JSP that Responds to a Form and Uses Java Beans.

You can find additional examples in the directory *`server_root`*`/plugins/samples/servlets/jsp`.

These examples are simple, introductory examples. For information about creating JavaServer Pages, see Sun Microsystem's JavaServer Pages web page at:

`http://www.javasoft.com/products/jsp/index.html`

JSP that Accesses the Request Object

JavaServer Pages contain both standard HTML tags and JSP tags. One of the JSP tags is `<DISPLAY>` which displays information contained in bean objects. The `<DISPLAY>` tag has the following format:

```
<DISPLAY property=object:property>
```

where `property` can include nested properties, for example:

```
property:object1:property2:property3
```

Examples of the `<DISPLAY>` tag are:

```
<DISPLAY property=request:method>
<DISPLAY property=product:version:modificationDate>
```

All JSP pages can implicitly access the `request` object, which contains information about the request that invoked the page, such as the requested URI, the query string, the content type and so on. The `request` object has properties such as `requestURI`, `queryString`, and `contentType`.

This example displays information about the current request. It gets all its data from the `request` object, which is automatically passed to the JSP. This example is the `snoop.jsp` example in the *`server_root`*`/plugins/samples/servlets/jsp/snoop` directory.

Figure 2.4 shows an example of the output page generated by this JSP.

Figure 2.4 Output page generated by snoop.jsp

The source code for `snoop.jsp` is:

```
<HTML>

<BODY>
<H1> Request Information </H1>

JSP Request Method: <DISPLAY property=request:method><BR>
Request URI: <DISPLAY property=request:requestURI><BR>
Request Protocol: <DISPLAY property=request:protocol><BR>
Servlet path: <DISPLAY property=request:servletPath><BR>
Path info: <DISPLAY property=request:pathInfo><BR>
Path translated: <DISPLAY property=request:pathTranslated><BR>
Query string: <DISPLAY property=request:queryString><BR>
Content length: <DISPLAY property=request:contentLength><BR>
Content type: <DISPLAY property=request:contentType><BR>
Server name: <DISPLAY property=request:serverName><BR>
Server port: <DISPLAY property=request:serverPort><BR>
Remote user: <DISPLAY property=request:remoteUser><BR>
Remote address: <DISPLAY property=request:remoteAddr><BR>
```

```
Remote host: <DISPLAY property=request:remoteHost><BR>
Authorization scheme: <DISPLAY property=request:authType> <BR>

The input parameter value is <DISPLAY property=request:params:input1
placeholder="NoValueGiven">

</BODY></HTML>
```

JSP that Responds to a Form and Uses Java Beans

This example discusses a JSP that accesses data on Java beans to respond to a form.

This example presents a web page, `shoeform.htm`, that displays a form asking the user to select the kinds of shoes they want to know more about. The action of the form is `shoes.jsp`. This JSP file gets information about the relevant kinds of shoes from a set of Java beans.(Note that Java beans were originally designed for use with visual tool builders, and they have some overhead that can make them slow when used to retrieve data to display in web pages.)

The discussion of this example has the following sections:

- The Form
- The Output Page Generated by the JSP File
- Accessing Input Parameters
- Testing for Parameter Values
- Using Externally Defined Java Beans
- Source Code for shoes.jsp
- The Java Beans Used in this Example
- Running the Example

The Form

The form in the page has the following elements:

- a text field called `userName`.
- three checkboxes named `Sandals`, `HikingBoots` and `WalkingShoes`
- a submit button

The form's method is `POST` and the action is `shoes.jsp`. (It also works if the form's method is `GET`.)

```
<FORM METHOD=POST ACTION="shoes.jsp">
```

Figure 2.5 shows an example of the form.

Figure 2.5 This form invokes a JSP as its action

You can view this form live at `shoeform.htm`.

The Output Page Generated by the JSP File

The JSP file `shoes.jsp` responds to the form. It uses the `request:params` property to access the parameters received from the form.

The output page generated by `shoes.jsp` prints a welcome message that includes the user's name, as entered in the `userName` text field in the form. It then displays information about each kind of shoe that was selected. The JSP file gets information about the shoes from Java Beans.

This JSP file demonstrates the following features:

- Accessing Input Parameters
- Testing for Parameter Values
- Using Externally Defined Java Beans

Figure 2.6 shows an example of the output from `shoes.jsp`:

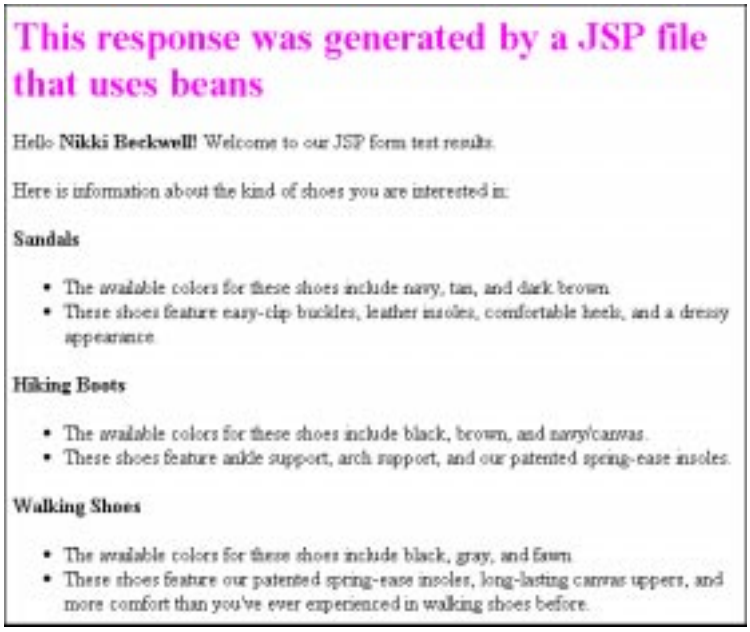

Figure 2.6 A JSP page generated in response to a form submission

You can view the JSP file at:

`shoes.jsp`

Accessing Input Parameters

JSP pages can extract input parameters when invoked by a URL with a query string, such as when they are invoked as a form action for a form that uses the GET method. The implicit `request` object has a property `params` whose value is an object that has attributes for each parameter in the query string.

For example, if the following URL is used to invoke a JSP:

```
http://poppy.my_domain.com/products/
shoes.jsp?userName=Jocelyn&Sandals=on&WalkingShoes=on
```

The `request:params` object has properties `userName`, `Sandals`, and `WalkingShoes`.

Testing for Parameter Values

This example uses the <INCLUDEIF> JSP tag, which includes a block of JSP and/or HTML code if a given parameter has a specified property.

For example:

```
<INCLUDEIF PROPERTY="request:params:Sandals" VALUE="on">
<H4>Sandals</H4>
</INCLUDEIF>
```

This code says if, and only if, the input parameters include a parameter named Sandals whose value is "on", then print <H4>Sandals</H4> to the output page.

For more information about the <INCLUDEIF> tag, and the corresponding <EXCLUDEIF> tag, see the JSP API documentation from Sun Microsystems at:

```
http://www.javasoft.com/products/jsp/index.html
```

Using Externally Defined Java Beans

Some bean objects including the request object, are always available implicitly to a JSP page. Other objects, such as user-defined objects, are not automatically available to the page, in which case you have to include a <USEBEAN> tag to tell the page which object to use. S

The JSP tag <USEBEAN> creates an instance of an externally defined Java Bean for use within the JSP page. For example, the following code creates an instance of the Sandals bean which is in com/shoes/beans.

```
<USEBEAN name="sandalA" type=com.jocelyn.beans.sandals lifespan=page>
</USEBEAN>
```

In this case, the Sandals bean instance exists for the duration of the page.

The following code retrieves the value of the colors property of the Sandals bean.

```
The available colors for these shoes include
<DISPLAY property=sandalA:colors>
```

Source Code for shoes.jsp

Here is the source code for the JSP file shoes.jsp:

```
<HTML>
```

```
<HEAD><TITLE>When the shoe fits, wear it</TITLE></HEAD>

<H1><FONT color="#FF00FF">
This response was generated by a JSP file that uses beans
</FONT></H1>

<!-- Get the person who sent the form from the userName parameter -->
<P>Hello <B><DISPLAY PROPERTY="request:params:userName">!</B>

Welcome to our JSP form test results.
</B></P>

<!-- Display a bullet item for each kind of shoe selected -->
<P>Here is information about the kind of shoes you are interested in:

<INCLUDEIF PROPERTY="request:params:Sandals" VALUE="on">
   <USEBEAN name="sandalS" type=com.shoes.beans.sandals lifespan=page>
   </USEBEAN>
   <H4>Sandals</H4>
   <UL>
      <LI>The available colors for these shoes include
         <DISPLAY property=sandalS:colors>.
      <LI>These shoes feature <DISPLAY property=sandalS:features>.
   </UL>
</INCLUDEIF>

<INCLUDEIF PROPERTY="request:params:HikingBoots" VALUE="on">
   <USEBEAN name="hikingH" type=com.shoes.beans.hikingBoots
      lifespan=page>
   </USEBEAN>
   <H4>Hiking Boots</H4>
   <UL>
      <LI>The available colors for these shoes include
         <DISPLAY property=hikingH:colors>.
      <LI>These shoes feature  <DISPLAY property=hikingH:features>.
   </UL>
</INCLUDEIF>

<INCLUDEIF PROPERTY="request:params:WalkingShoes" VALUE="on">
   <USEBEAN name="walkingW" type=com.shoes.beans.walkingShoes
      lifespan=page>
   </USEBEAN>
   <H4>Walking Shoes</H4>
   <UL>
      <LI>The available colors for these shoes include
         <DISPLAY property=walkingW:colors>.
      <LI>These shoes feature  <DISPLAY property=walkingW:features>.
   </UL>
</INCLUDEIF>

</BODY>
</HTML>
```

The Java Beans Used in this Example

The `shoes.jsp` file accesses three Java Bean objects, `sandals`, `hikingBoots`, and `walkingShoes`.

These classes all inherit from a superclass `Shoes`, which defines setter and getter methods for the variables `prodName`, `colors`, and `features`. The subclasses `Sandals`, `HikingBoots`, and `WalkingShoes` simply define their own values for these variables.

The source code for these classes is available through the following links:

- `shoes.java`
- `sandals.java`
- `hikingBoots.java`
- `walkingShoes.java`

A jar file containing all the class files is available at:

- `shoes.jar`

For more information about defining Java Beans, see:

`http://www.javasoft.com/beans/index.html`

Running the Example

To run this example, get the jar file `shoes.jar` (or create it by compiling `shoes.java`, `sandals.java`, `hikingShoes.java`, and `walkingShoes.java` and then packaging the compiled class files into a JAR file). Put `shoes.jar` in a subdirectory of your choosing in your Enterprise Server root directory. If you already have a directory where you put JAR files, you can put it in there.

Add the full pathname of the `shoes.jar` location to the JVM class path, which you can do in the Servlets>Configure JVM Attributes page of the Server Manager interface.

Put `shoeform.htm` and `shoes.jsp` together in a subdirectory in or below the Enterprise Server's document root directory. (JSP files do not go in a registered servlet directory.)

Note Note that the link above to `shoes.jsp` opens a file of the name `shoes.jsp.txt` -- save this file from the browser without the `.txt` extension to save it as a JSP file rather than a plain text file.

To view the form, open `shoesform.htm` using an http URL. For example, if you put `shoesform.htm` in the document root directory, open it with the URL `http://`*`your_server`*`/shoesform.htm`.

Appendix

A

Session Managers

Session objects maintain state and user identity across multiple page requests over the normally stateless HTTP protocol. A session persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times. The server can maintain a session either by using cookies or by rewriting URLs. Servlets can access the session objects to retrieve state information about the session.

This appendix has the following sections:

- Session Overview
- Specifying a Session Manager
- SimpleSessionManager
- MMapSessionManager
- How Do Servlets Access Session Data?

Session Overview

An HTTP session represents the server's view of the session. The server considers a session new under these conditions:

- The client does not yet know about the session
- The session has not yet begun

A session manager automatically creates new session objects whenever a new session starts. In some circumstances, clients will not join the session, for example, if the session manager uses cookies and the client does not accept cookies.

Enterprise Server 4.0 comes with two session managers for creating and managing sessions:

- `SimpleSessionManager` -- the default session manager when the server runs in single process mode.
- `MMapSessionManager` -- the default session manager when the server runs in multi-process mode.

Enterprise Server 4.0 also allows you to develop your own session managers and load them into the server. The build includes the source code for `SimpleSessionManager` and the session objects it manages, `SimpleSession`. The source code for these classes are provided as a starting point for you to define your own session managers if desired. These Java files are in the directory *`server-root`*`/plugins/samples/servlets/sessions/SimpleSession`.

Specifying a Session Manager

By default, if the Enterprise Server starts in single process mode, it uses `SimpleSessionManager` as the session manager for servlets. If it starts in multi-process mode, it uses `MMapSessionManager`. For more information about single process mode versus multi processes mode, see Chapter 7, “Configuring Server Preferences” in the Enterprise Server 4.0 Administrator's Guide.

You can change the session manager in either of the following ways:

- Use the Servlets>Configure Global Servlet Attributes page in the Server Manager interface.

  In the Session Manager field, specify the session manager. (You cannot specify initial parameter values in the interface).

- Edit the file `servlet.properties` in the directory *`server-id`*`/config`.

  Add a line specifying a value for `servlets.sessionmgr` and, if appropriate, also add a line specifying the parameters for the session manager. For example:

```
servlets.sessionmgr=com.netscape.server.http.session.YourSessionManager
servlets.sessionmgr.initArgs=maxSessions=20,timeOut=300,reapInterval=150
```

SimpleSessionManager

The `SimpleSessionManager` works only in single process mode. It is loaded by default if the Enterprise Server starts in single-process mode when a SessionManager is not specified in the `servlets.properties` configuration file. These sessions are not persistent, that is, all sessions are lost when the server is stopped.

Parameters

The `SimpleSessionManager` class takes the following parameters:

- `maxSessions` - the maximum number of sessions maintained by the session manager at any given time. The session manager refuses to create any more new sessions if there are already `maxSessions` number of sessions present at that time.
- `timeOut` - the amount of time in seconds after a session is accessed by the client before the session manager destroys it. Those sessions that haven't been accessed for at least `timeOut` seconds will be destroyed by the `reaper()` method.
- `reapInterval` - the amount of time in seconds that the SessionReaper thread sleeps before calling the `reaper()` method again.

Enabling SimpleSessionManager

To enable the Enterprise Server to use `SimpleSessionManager` do either of the following:

- Use the Servlets>Configure Global Servlet Attributes page in the Server Manager interface.

  In the Session Manager field specify:

  ```
  com.netscape.server.http.session.SimpleSessionManager
  ```

- Edit the file `servlet.properties` in the directory *`server-id`*`/config`.

  Add a line specifying a value for `servlets.sessionmgr` and a line specifying the parameters for the session manager:

```
servlets.sessionmgr=com.netscape.server.http.session.SimpleSessionManager
servlets.sessionmgr.initArgs=maxSessions=20,timeOut=300,reapInterval=150
```

Source Code for SimpleSessionManager

The `SimpleSessionManager` creates a `SimpleSession` object for each session. The source files for `SimpleSessionManager.java` and `SimpleSession.java` are in the directory *`server-root`*`/plugins/samples/servlets/sessions/SimpleSession`.

The source code for `SimpleSessionManager.java` and `SimpleSession.java` are provided so you can use them as the starting point for defining your own session managers and session objects. These files are very well commented.

`SimpleSessionManager` extends `NSHttpSessionManager`. The class file for `NSHttpSessionManager` is in the JAR file `NSServletLayer.jar` in the directory *`server_root`*`/plugins/jar`. `SimpleSessionManager` implements all the methods in `NSHttpSessionManager` that need to be implemented, so you can use `SimpleSessionManager` as an example of how to extend `NSHttpSessionManager`. When compiling your subclass of `SimpleSessionManager` or `NSHttpSessionManager`, be sure that the JAR file `NSServletLayer.jar` is in your compiler's class path.

MMapSessionManager

This is a persistent memory map file-based session manager that works in both single process as well as multi-process mode. It can be used for inter-process communication. It is loaded by default if the Enterprise Server starts in multi-process mode when a session manager is not specified in the `servlets.properties` configuration file.

Parameters

`MMapSessionManager` takes the following parameters:

- `maxSessions` - the maximum number of sessions maintained by the session manager at any given time. The session manager refuses to create any more new sessions if there are already `maxSessions` number of sessions present at that time.
- `maxValuesPerSession` - maximum number of values a session can hold.
- `maxValuesSize` - maximum size of the object that can be stored in the session.
- `timeOut` - the amount of time in seconds after a session is accessed by the client before the session manager destroys it. Those sessions that haven't been accessed for at least `timeOut` seconds will be destroyed by the `reaper()` method.
- `reapInterval` - the amount of time in seconds that the SessionReaper thread sleeps before calling the `reaper()` method again.

Enabling MMapSessionManager

To enable Enterprise Server to use `MMapSessionManager` do either of the following:

- Use the Servlets>Configure Global Servlet Attributes page in the Server Manager interface.

  In the Session Manager field specify:

  ```
  com.netscape.server.http.session.MMapSessionManager
  ```

- Edit the file `servlet.properties` in the directory *`server-id`*`/config`.

  Add a line specifying a value for `servlets.sessionmgr` and a line specifying the parameters for the session manager:

```
servlets.sessionmgr=com.netscape.server.http.session.MMapSessionManager
SessionManager
servlets.sessionmgr.initArgs=maxSessions=20,maxValueSize=1024,timeOut=3
00,reapInterval=150
```

This session manager can only store objects that implement `java.io.Serializable`.

How Do Servlets Access Session Data?

To access the state information stored in a session object, your servlet can create a new session as follows:

```
// request is an HttpServletRequest that is passed to the servlet
SessionClass session = request.getSession(true);
```

The servlet can call any of the public methods in `javax.servlet.http.HttpSession` on the session object. These methods include (amongst others):

```
getCreationTime
getId
getLastAccessedTime
getMaxInactiveInterval
getValue
```

For more information about the classes `HttpServletRequest` and `HttpSession`, see Sun Microsystem's API Servlets Documentation at:

`http://www.javasoft.com/products/servlet/2.1/html/api-reference.fm.html`

Appendix

B

Servlet Settings in obj.conf

Netscape Enterprise Server 4.0 automatically modifies the file `obj.conf` in the `config` directory to load the servlet engine if servlets are enabled. Whenever you make changes to servlet settings by using the Server Manager interface, the system automatically updates `obj.conf` appropriately.

However, in case you are interested in the settings that affect servlets, this appendix describes the directives in `obj.conf` and value settings `mime.types` that are relevant to servlets.

Directives for Enabling Servlets

The following directives in the `init` section of `obj.conf` load and initialize the servlet engine to enable servlets:

```
Init fn="load-modules" shlib="server_root/bin/https/bin/
NSServletPlugin.dll/so"
funcs="NSServletEarlyInit,NSServletLateInit,NSServletNameTrans,
NSServletService" shlib_flags="(global|now)"

Init fn="NSServletEarlyInit" EarlyInit=yes

Init fn="NSServletLateInit" LateInit=yes
```

`NSServletEarlyInit` takes an optional parameter `cache_dir` that specifies the location of a temporary cache directory for JSP classes. By default the directory is named `"ClassCache"` and goes under your server root directory.

`NSServletLateInit` takes an optional parameter `CatchSignals` that specifies whether or not Java thread dumps are logged. The value is `yes` or `no`.

When servlets are enabled, the following directive appears in the default object:

```
NameTrans fn="NSServletNameTrans" name="servlet"
```

This directive is used for servlet virtual path translations and for the URI cache. Do not delete this line when servlets are enabled.

Also, `obj.conf` always has the following objects, which you should not delete:

```
<Object name="servlet">
Service fn="NSServletService"
</Object>


<Object name="jsp">
Service fn="NSServletService"
</Object>
```

If you delete these objects, you will no longer be able to use the Server Manager interface to enable servlets and modify servlet settings.

Directives for Registered Servlet Directories

For each registered servlet directory, the default object in `obj.conf` has a `NameTrans` directive that assigns the name `ServletByExt` to all requests to access that directory. For example:

```
NameTrans fn="pfx2dir" from="/servlet" dir="D:/Netscape/Server4/docs/
servlet" name="ServletByExt"
```

A separate object named `ServletByExt` has instructions for processing requests for servlets:

```
<Object name="ServletByExt">
ObjectType fn="force-type" type="magnus-internal/servlet"
Service type="magnus-internal/servlet" fn="NSServletService"
</Object>
```

Do not delete this object, even if no servlet directories are currently registered. If this object is deleted, you will no longer be able t use the Server Manager interface to register servlet directories.

JSP

The following line in `mime.types` sets the type for files with the extension `.jsp`:

```
type=magnus-internal/jsp exts=jsp
```

When JSP is enabled, the following directive in `obj.conf` handles the processing of requests for files of type `magnus-internal/jsp` (that is, JSP files)

```
Service fn="NSServletService" type="magnus-internal/jsp"
```

Appendix

C

servlets.properties and rules.properties

This appendix discusses the purpose and use of the files `servlet.properties` and `rules.properties`, which reside in the directory *`server_id`*`/config`.

servlet.properties

The `servlet.properties` file defines global servlet settings and the list of servlets in the system.

The `servlet.properties` file specifies global settings for servlets, such as a servlet to run when the Enterprise server starts up, the reload interval for servlets, and so on. It also specifies configuration information for individual servlets. Configuration information includes the class name, the class path and any input arguments required by the servlet.

If you want to specify a virtual path translation for a servlet, the servlet must be configured in the `servlet.properties` file.

You can specify configuration information for servlets either by using the Servlets>Configure Servlet Attributes page in the Server Manager interface or by editing `servlets.properties` directly. Whenever you make a change in the Servlets>Configure Servlet Attributes page in the Server Manager interface, the system automatically updates `servlets.properties`.

When specifying attributes for a servlet, you specify a `name` parameter for the servlet. This name is not the name of the class file for the servlet but is instead an internal identifier for the servlet. You specify the name of the class file as the value of the `code` parameter.

Here is a sample `servlet.properties` file:

`servlet.properties`

```
# Servlets Properties
# servlets to be loaded at startup
servlets.startup= hello
# the reload interval for dynamically-loaded servlets and JSPs
# (default is 10 seconds)
servlets.config.reloadInterval=5
# the default document root,
# needed so ServletContext.getRealPath will work
servlets.config.docRoot=E:/Netscape/Server4/docs
# the session manager
servlets.sessionmgr=com.netscape.server.http.session.SimpleSessionManager
# tracker servlet
servlet.tracker.code=MyTrackerServlet
servlet.tracker.classpath=D:/Netscape/Server4/docs/servlet
# demo1 servlet
servlet.demo1.code=Demo1Servlet
servlet.demo1.classpath=D:/Netscape/Server4/docs/demos
servlet.demo1.initArgs=a1=0,b1=3456
```

rules.properties

The `rules.properties` files defines servlet virtual path translations. For example, you could set up a mapping so that the URL pointing to `/index.html` URL invokes the servlet `/servlet/runintro.class`. You can specify virtual paths for your servlets either by setting parameters in the Servlets>Configure Servlet Virtual Path Translation page of the Server Manager interface or by specifying the paths in the `rules.properties` file.

Note that the “name” associated with the servlet in `servlets.properties` is used in the file `rules.properties` -- the class name of the servlet does not show up in `rules.properties`. For example, the following lines in `servlet.properties` associate the servlet name `demo1` with the servlet class file `Demo1Servlet.class` in the directory `D:/Netscape/Server4/docs/demos`.

```
# in servlets.properties
# demo1 servlet
```

```
servlet.demo1.code=Demo1Servlet
servlet.demo1.classpath=D:/Netscape/Server4/docs/demos
```

The following line in `rules.properties` defines a servlet virtual path translation such that the URL `http://`*`server-name`*`/mytest2` invokes the servlet at `D:/Netscape/Server4/docs/demos/Demo1Servlet.class`.

```
/mytest2=demo1
```

Here is an example of `rules.properties`.

`rules.properties` (defines URL name space for each of the servlets):

```
# Servlet rules properties
# This file specifies the translation rules for invoking servlets.
# The syntax is:
# /virtual-path=servlet-name
# where virtual-path is the virtual path used to invoke the servlet,
# and servlet-name is the name of the servlet as specified in
# servlets.properties.
# Surrounding white space is ignored.
# The ordering of the rules is not important, as the longest
# match is always used first.
/mytest1=tracker
/mytest2=demo1
```

Appendix

D

JVM Configuration

The Java Virtual Machine (JVM) works by default without any additional configuration if properly set up.

However, if you need to specify settings for the JVM, such as additional classpath information, you can configure the JVM properties for Enterprise Server via the Administrator interface. You can add as many other properties as you want to (up to 64).

You can also configure JVM parameters by editing the `jvm11.conf` or `jvm12.conf` configuration files, (depending on which version of the JDK is being used) which reside under the server's `config` directory.

Here is an example `jvm.conf` file:

`jvm.conf` (example for JDK1.1):

```
[JVMConfig]
#jvm.nativeStackSize=131072
#jvm.javaStackSize=409600
#jvm.minHeapSize=1048576
#jvm.maxHeapSize=16777216
#jvm.verifyMode=0
#jvm.enableClassGC=1
#jvm.enableVerboseGC=0
#jvm.disableAsyncGC=0
#jvm.verboseMode=1
jvm.enableDebug=1
jvm.debugPort=2525
```

```
jvm.classpath=/server_root/bin/https/jre/lib/classes.zip;
ANY_OTHER_JAVA_SPECIFIC_PROPERTY
```

For example, to disable JIT you can add the following line to `jvm.conf`:

```
java.compiler=DISABLED
```

`jvm12.conf` (example for JDK1.2)

```
[JVMConfig]
#jvm.minHeapSize=1048576
#jvm.maxHeapSize=16777216
jvm.enableClassGC=0
#jvm.verboseMode=1
#jvm.enableDebug=1
jvm.option=-Xrunoii
jvm.profiler=optimizeit
java.compiler=NONE
OPTITDIR=D:/App/IntuitiveSystems/OptimizeIt30D
```

The configuration file for JDK1.2 is similar to the one for JDK1.1. Generally you should use plain property options (like `name=value`) for the JDK1.2 configuration and `jvm.option=options` for JVM-vendor dependant configurations. There could be multiple occurrences of `jvm.option` parameters.

In Enterprise Server 4.0, jvm.conf files support a configuration parameter called `jvm.stickyAttach`. Setting this parameter to 1 causes threads to remember that they are attached to the JVM, thus speeding up request processing by eliminating calls to `AttachCurrentThread` and `DetachCurrentThread`. It can, however, have side effect as recycled threads that may be doing other processing can be suspended from the garbage collection pool arbitrarily.

For information about JVM, see *The Java Virtual Machine Specification* from Sun at

`http://www.javasoft.com/docs/books/vmspec/html/VMSpecTOC.doc.html`

Appendix

E

Remote Servlet Debugging

Enterprise Server 4.0 ships with the Java Runtime Environment (JRE), not the Java Development Kit (JDK). However, during installation you can select an option that tells the server to use the JDK if it is installed elsewhere on your system.

If the server has been instructed to use a JDK, you can do remote servlet debugging. If the server is using the JRE, you need to switch it to using the JDK before you can do remote debugging. For information on instructing the server to use the JDK or the JRE, see the section "Configuring JRE/JDK Paths" in Chapter 1, "Using Servlets and JavaServerPages."

Assuming that the server is using the JDK, you can enable remote debugging by following these steps:

- Make sure that the server is running in single-process mode. Single-process mode is the default, but you can check in the file `magnus.conf` to make sure that the `MaxProcs` parameter is not set to a value greater than 1. If you do not see a setting for `MaxProcs` in `magnus.conf` then the default value of 1 is enabled for it.

  For more information about single process mode versus multi processes mode, see Chapter 7, "Configuring Server Preferences" in the Enterprise Server 4.0 Administrator's Guide.

- Set the following parameters in `jvm11.conf` or `jvm12.conf` as appropriate:

```
jvm.enableDebug=1
java.compiler=DISABLED
```

- Change your `obj.conf` to use the debuggable version of the JVM (this doesn't apply for JDK1.2):

```
Init fn="load-modules" shlib="server_root/bin/https/bin/
NSServletPlugin_g.dll"
funcs="NSServletEarlyInit,NSServletLateInit,NSServletNameTrans,NSServle
tService" shlib_flags="(global|now)"
```

- Start the server manually and record the password for remote debugging (this will be displayed on the console)

- Start the Java debugger: `jdb -host your_host -password the_password`

You should be able to debug your Java classes now.

Appendix

F

Remote Servlet Profiling

You can use Optimizeit! 3.0 from Intuitive Systems to perform remote profiling on the Enterprise Server to discover bottlenecks in server-side performance.

You can purchase Optimizeit! from Intuitve Systems at:

`http://www.optimizeit.com/index.html`

Once Optimizeit! is installed using the following instructions it becomes integrated into Enterprise Server 4.0.

To enable remote profiling, make the following modifications in the `jvm11.conf` or `jvm12.conf` files as appropriate:

`jvm[12].conf` example:

```
jvm.enableClassGC=0
jvm.option=-Xrunoii # this is only required for JDK1.2
jvm.profiler=optimizeit
java.compiler=NONE
OPTITDIR=<optimizeit_root_dir>/OptimizeIt30D
```

When the server starts up with this configuration, you can attach the profiler (for further details see the Optimizeit! documentation).

Also, update the `PATH` and `NSES_CLASSPATH` system variables to include the profiler's own jar files and dlls.

Note: If any of the configuration options are missing or incorrect the profiler may experience problems that affect the performance of the Enterprise Server.

Appendix

G

API Clarifications

This appendix clarifies the way some of the standard Servlet API work in Enterprise Server 4.0. For the official documentation for the API discussed here (and for all servlets API) see the Servlets API Class Reference published by Sun Microsystems at:

`http://www.javasoft.com/products/servlet/2.1/html/api-reference.fm.html`

This appendix provides clarifications for using the following API with Enterprise Server 4.0:

- HttpUtils.getRequestURL()
- HttpSession.setMaxInactiveInterval()
- GenericServlet.getInitParameter() and getInitParameterNames()
- ServletContext.getContext()
- RequestDispatcher.forward() and include()
- Request.getInputStream() and getReader()

HttpUtils.getRequestURL()

```
public static StringBuffer getRequestURL(HttpServletRequest request);
```

This method reconstructs the URL used by the client to make the given request on the server. This method accounts for difference in scheme (such as http, https) and ports, but does not attempt to include query parameters.

This method returns a StringBuffer instead of a String so that the URL can be modified efficiently by the servlet

Clarification

To determine the server name part of the requested URL, Enterprise Server first tries the to use the "Host" header and then looks at the value of `ServerName` in `magnus.conf`. The server name by default is the machine name. But this value is editable while installing Enterprise Server 4.0. If the server name has been changed, `HttpUtils.getRequestURL` might not return the host name that is needed to reconstruct the request.

For example, suppose the request is `http://abc/index.html`. However, the server name has been changed to `xyz`. In this case, `HttpUtils.getRequestURL()` might return `http://xyz/index.html`, which is not the original URL that was requested.

HttpSession.setMaxInactiveInterval()

```
public int setMaxInactiveInterval(int interval);
```

Sets the amount of time that a session can be inactive before the servlet engine is allowed to expire it.

Clarification

The returned `int` is the previous value.

It is not possible to set the maximum inactive interval so that the session never times out. The session will always have a timeout value.

If you pass a negative or zero value, the session expires immediately.

GenericServlet.getInitParameter() and getInitParameterNames()

```
public String getInitParameter(String name);
```

```
public String getInitParameter(String name);
```

This method returns a String containing the value of the servlet's named initialization parameter, or null if this parameter does not exist.

```
public Enumeration getInitParameterNames();
```

This method returns an enumeration of String objects containing the names of the initialization parameters for the calling servlet. If the calling servlet has no initialization parameters, `getInitParameterNames` returns an empty enumeration.

Clarification

For servlets running on Enterprise Server 4.0, the methods `getInitParameter` and `getInitParameterNames` for the class `ServletConfig` only work for servlets that are invoked through virtual path translations. The same restriction applies to the convenience methods of the same names in the class `GenericServlet`, which invoke the corresponding methods on `ServletConfig`.

For information about setting virtual path translations, see the section Specifying Servlet Virtual Paths in Chapter 1, "Using Servlets and JavaServerPages."

These methods do not work if the servlet is invoked by a client request that specifies a servlet in a registered servlet directory rather than using a virtual path translation to access the servlet.

ServletContext.getContext()

```
public ServletContext getContext(String uripath);
```

Returns the servlet context object that contains servlets and resources for a particular URI path, or null if a context cannot be provided for the path.

Clarification

This method only works if both the following conditions are true:

- the servlet whose context is being obtained (that is, the servlet pointed to by `uripath`) has been configured either through the Servlets>Configure Servlet attributes property of the Server Manager interface or by editing `servlets.properties`.
- the servlet whose context is being obtained has been loaded.

  Enterprise Server 4.0 does not load a servlet specified by a URI when `getContext()` is called from another servlet to get the context of an unloaded servlet.

RequestDispatcher.forward() and include()

```
public void forward(ServletRequest request, ServletResponse response)
throws ServletException, IOException;
```

Used for forwarding a request from this servlet to another resource on the web server. This method is useful when one servlet does preliminary processing of a request and wants to let another object generate the response.

The request object passed to the target object will have its request URL path and other path parameters adjusted to reflect the target URL path of the target object.

You cannot use this method if a `ServletOutputStream` object or `PrintWriter` object has been obtained from the response. In that case, the method throws an `IllegalStateException`.

```
public void include(ServletRequest request, ServletResponse response)
throws ServletException, IOException
```

Used for including the content generated by another server resource in the body of a response. In essence, this method enables programmatic server-side includes. The request object passed to the target object will reflect the request URL path and path info of the calling request. The response object only has access to the calling servlet's `ServletOutputStream` object or `PrintWriter` object.

An included servlet cannot set headers. If the included servlet calls a method that needs to set headers (such as cookies), the method is not guaranteed to work. As a servlet developer, you must ensure that any methods that might

need direct access to headers are properly resolved. To ensure that a session works correctly, start the session outside the included servlet, even if you use session tracking.

Clarification

In Enterprise Server 4.0, the `dispatcher.forward()` method may or may not throw an `IllegalStateException` when either `Writer` or `OutputStream` have been obtained. This behavior follows the 2.2 draft and is needed for JSP error page handling. It throws the exception only if the actual data has been flushed out and sent to the client. Otherwise, the data pending in the buffer is simply discarded.

In the case of servlets in registered servlet directories and JSP, `include()` flushes the output and headers before doing the include, which effectively causes any further calls to `setHeader()` to have no effect. The same behavior occurs when forwarding to non-servlet URIs (like cgis or static files). In the case of statically-defined uri mapping rules `setHeader()` might work until it exceeds the buffer.

The `forward()` and `include()` methods may throw a `ServletException` if the target URI is identified as an unsafe URI (that is, it includes insecure path characters such as `//`, `/./`, `/../` and`/.`, `/..` (and also `./` for NT) at the end of the URI.

Request.getInputStream() and getReader()

There are two ways for a servlet to read the raw data posted by a client:

- by obtaining the InputStream through the `request.InputStream()` method, an older method.
- by obtaining a BufferedRead through the `request.getReader()` method, a method in use since 2.0.

Clarification

A servlet will hang if it attempts to use an InputStream to read more data than is physically available. (To find how much data is available, use `request.getContentLength()`.) However, if the servlet reads data using a BufferedReader returned from a call to `getReader ()`, the allowed content length is automatically taken into the account.

Index

F

G

H

I

J

M

N

O

P

R

S

T

U

V

Programmer's Guide to Servlets in Enterprise Server 4.0