SYBEX® Sample Chapter

Java™ Developer's Guide to Servlets and JSP

by Bill Brogden

ISBN #0-7821-2809-2

Debugging

CHAPTER 5

Debugging

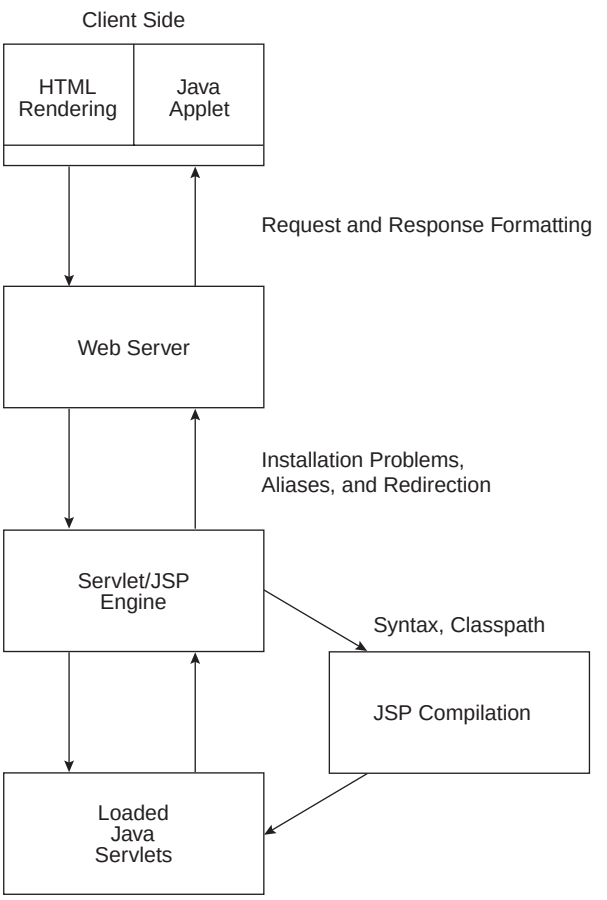
- Debugging Hints for Browser Settings
- Snooping on Browser Requests and Server Responses
- Common Server Installation Problems
- Hints for Debugging JSP
- Designing for Ease in Debugging
- Making the Most of Exceptions
- Monitoring Memory Leaks

For many programmers, working with servlets and JavaServer Pages will be their first contact with multi-tier applications. Because servlets and JSP applications involve so many cooperating parts, there are many places for bugs to reside. Further complicating things is the fact that working applications are usually multithreaded and have many simultaneous activities. Debugging these complex applications involves techniques that must be adapted especially for this complex environment.

Anything That Can Go Wrong Will

The first rule of debugging is to be methodical, so I am going to start with a diagram (see Figure 5.1) of things that can go wrong between the client and the servlet engine. Later sections tackle debugging inside servlets.

FIGURE 5.1
Some bug locations between client and servlet



Let's consider things that can go wrong at the very top of the diagram—the typical Web browser. There are several things to watch out for at the browser level:

Web Page Caching Many programmer hours are wasted because browsers use cached pages instead of loading your newly modified page. When debugging, make sure your browser is doing minimal caching and checks for newer versions of a page with every visit. It is helpful to have your JSP or servlet write a version number as part of the page output. (Of course, you do, however, have to remember to change the version number with every source code change.)

Cookie Security Naturally, if your application needs to use cookies or sessions, you should make sure your browser allows them.

Proxy Caching If your browser communicates with the Web server through a proxy, make sure the proxy is not caching pages or applet classes.

Proxy Blocking A proxy or firewall may block certain kinds of content, causing strange behavior in your Web application.

Applet Class Reloading If your application involves an applet on the client side, beware of browsers that do not reload applets when the Web page is reloaded. Try closing all instances of the browser and restarting if you get strange behavior by applets. I like to incorporate a statement writing the version number to `System.out` in an applet. That way you can look at the browser's Java console or javalog.txt file to verify that an old version has not been cached.

Requests to a Web server may be formulated by the browser interface or by an applet. With an applet, you can print the request line to `System.out` to examine it, but with forms on HTML pages and with JavaScript methods, it is hard to determine exactly what is being sent. I have written a Java application to spy on both requests and responses; that application is discussed in the following sections.

Responses from a Web server can be very uninformative and frustrating, as anybody who has gotten a "404 File Not Found" message is aware. The HTTP standard defines a number of standard error numbers for responses that are summarized in Appendix A. Your servlet or JSP code can send an error message with the `sendError` method in the `HttpServletResponse` class. Unfortunately, when your browser gets one of these codes, you frequently can't tell if it is coming from the Web server or from a servlet.

Spying on Requests

Because communication between a client and a Web server is conducted with streams and sockets, it is possible to create a Java utility that can intercept and monitor both request and response. In this section, I describe one simple utility that captures the client request and server responses (as long as they are in text form).

The UtilSnoop Application

The `UtilSnoop` program takes advantage of the fact that browser requests are directed at both a specific host and a specific port. In the example I am using, the Tomcat server listens on port 8080. The utility listens for browser requests on port 9000 and redirects lines of text to Tomcat while copying the lines to a display area. Responses coming to the utility are both sent to the client on port 9000 and copied to another display area. Figure 5.2 summarizes this data flow.

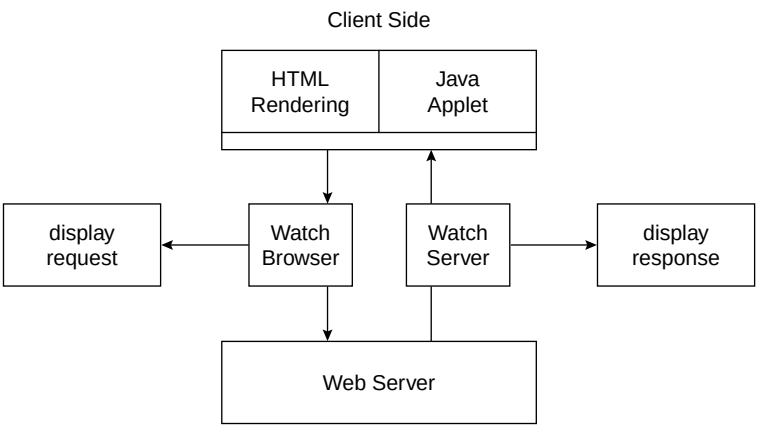
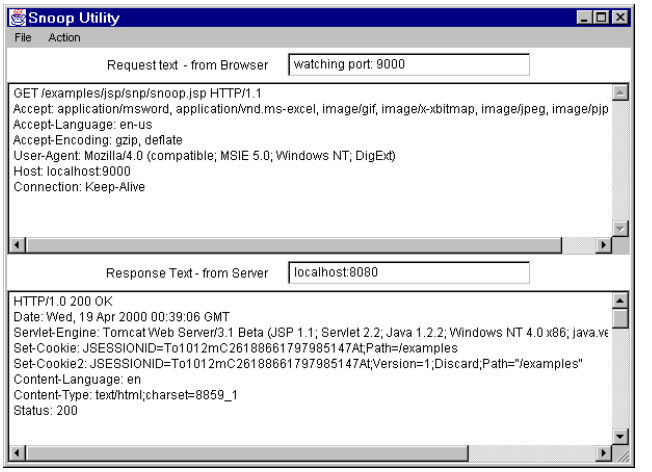


FIGURE 5.2 The Snoop utility sits between the client and Web server.

The program is implemented as a Java AWT graphic application. Figure 5.3 shows it in action: The top text area is the request from the browser, and the bottom shows the start of the server response. This particular request was for the JSP page named snoop.jsp that is one of the standard examples usually shipped with JSP engines.

FIGURE 5.3
Snoop utility display

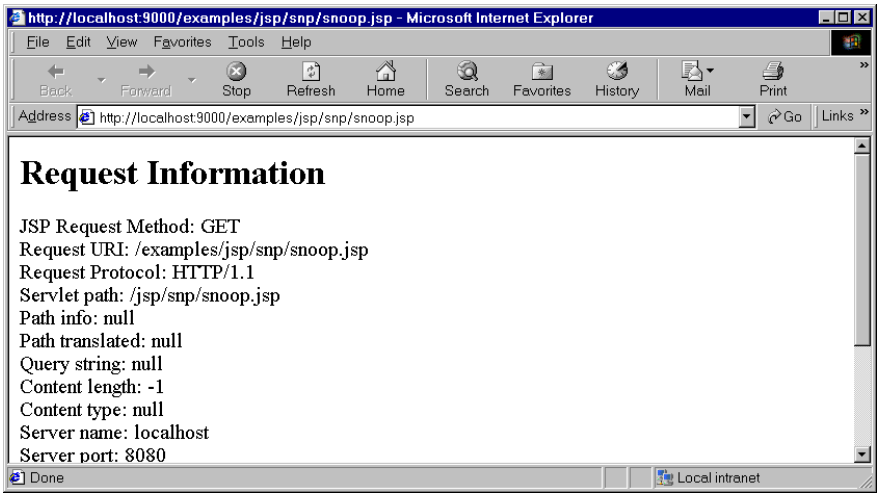


Many interesting points about interactions between a browser and a server are revealed by snooping on the headers, as shown in Figure 5.3. For example, you can see that the server has sent both "Set-Cookie" and "Set-Cookie2" headers to set a session ID. Note that the path for these cookies has been set to the "/examples" directory; this means that these cookies will only be sent with future requests to pages based at that directory.

You can copy the text in the utility displays to the Windows Clipboard by highlighting the text with the mouse and pressing Ctrl+C. However, a better solution would be to provide an option for saving the text in a file, but this is left as an exercise for the student.

Figure 5.4 shows the browser's display for this request. The snoop.jsp page formats request parameters in an easy-to-read list. Directing your browser to this page is a good way to verify that a JSP engine is working correctly.

FIGURE 5.4
The browser results



How UtilSnoop Works

`UtilSnoop` is a Java graphical application that uses the AWT graphical user interface (GUI) classes. The following menu options are provided:

File—Load Properties Brings up a file dialog box so that the default settings can be changed via a properties file.

File—Exit Simply exits immediately.

Action—Clear Clears the text areas.

Action—Start Opens sockets and starts threads, ready for the next request and response.

Action—Stop Sets a flag that will eventually stop the threads.

Listing 5.1 shows the start of the `UtilSnoop` class source code. Note that there are instance variables with default values for the host name and the ports involved. These can be replaced in the `setProp` method by values read in from a properties file.

Listing 5.1: Start of the UtilSnoop class listing

```
import java.awt.*;
import java.io.* ;
import java.util.* ;
import java.net.* ;
```

```
public class UtilSnoop extends java.awt.Frame implements java.lang.Runnable {
  String host = "localhost" ;
  int browserPort = 9000 ;
  int hostPort = 80 ;

  Vector fromBrowser, fromServer ;
  boolean running = false ;

  PrintWriter toBrowser ;
  Socket browserSocket ;
  ServerSocket ssok ;

  PrintWriter toServer ;
  Socket serverSocket ;

  private void setWatchTF(){
    requestTF.setText("watching port: " + browserPort );
    responseTF.setText( host + ":" + hostPort );
  }

  // set from properties file
  private void setProp( Properties p ){
    try {
      String tmp = p.getProperty("host") ;
      if( tmp != null ) host = tmp ;
      tmp = p.getProperty("hostport");
      if( tmp != null ){
        hostPort = Integer.parseInt( tmp );
      }
      tmp = p.getProperty("browserport");
      if( tmp != null ){
         browserPort = Integer.parseInt( tmp );
      }
    }catch( Exception e){
       System.out.println("Setting Propeties " + e );
    }
  }
```

When the Start menu item is selected, the method shown in Listing 5.2 is executed. This method creates new `Vector` objects to store the `String` objects received from browser and server, and then it starts two `Thread` objects attached to anonymous inner classes, one to execute the `watchBrowser` method and one for the `watchServer` method. Finally, it starts the `Thread` that executes the `run` method in the `UtilSnoop` class itself.

Listing 5.2: The method that starts snooping

```
void startMI_ActionPerformed(java.awt.event.ActionEvent event){
  // note: every start creates new Vectors and Threads
  fromBrowser = new Vector();
  fromServer = new Vector();
  running = true ;
  Thread t1 = new Thread( new Runnable() {
    public void run(){
      while( running ){
        System.out.println("watchBrowser start");
        try{   watchBrowser();
           System.out.println("watchBrowser returns");
        }catch(Exception e1){
           System.out.println("watchBrowser " + e1 );
        }
      }
      System.out.println("exit watchBrowser Thread");
    } // end run method
  });
  t1.setPriority( Thread.MIN_PRIORITY );
  Thread t2 = new Thread( new Runnable() {
    public void run(){
      while( running ){
        System.out.println("watchServer start");
        try { watchServer();
           System.out.println("watchServer returns");
        }catch(Exception e2){
          System.out.println("watchServer " + e2 );
        }
      }
      System.out.println("exit watchServer Thread");
    }
  });
  t2.setPriority( Thread.MIN_PRIORITY );
  t1.start() ;
  t2.start() ;
  new Thread( this ).start();
}

void stopMI_ActionPerformed( java.awt.event.ActionEvent event){
  running = false ;
  System.out.println("running stop");
}
```

The `run` method belonging to the `UtilSnoop` class (see Listing 5.3) watches both `Vector` objects for `String` objects received from the browser or server and sends them to the correct destination. It also adds them to their respective `TextArea` displays. Recall that the adding and removing methods in the `Vector` class are synchronized, so there can be no conflict between the different threads.

Listing 5.3: The UtilSnoop class run method

```
 // this Thread handles received data from both sides
public void run() {
  String tmp = null ;
  int errCt = 0 ;
  System.out.println("main run method starts");
  while( running ){
   try {
     while( fromBrowser.size() > 0 ){
       tmp = (String)fromBrowser.firstElement();
       toServer.print( tmp ); toServer.print("\r\n");
       toServer.flush();
       requestTA.append( tmp ); requestTA.append("\n");
       fromBrowser.removeElementAt(0);
     }
     while( fromServer.size() > 0 ){
       tmp = (String)fromServer.firstElement();
       toBrowser.print( tmp ); toBrowser.print("\r\n");
       toBrowser.flush();
       responseTA.append( tmp ); responseTA.append("\n");
       fromServer.removeElementAt(0);
     }
     Thread.sleep( 80 );
   }catch(Exception e){
      e.printStackTrace( System.out );
      if( ++errCt > 10 ) break ; ;
   }
  }
  System.out.println("main run stop");
}
```

Recall that the `watchBrowser` and `watchServer` methods (see Listing 5.4) are executed by two separate `Thread` objects using two anonymous inner classes. Although the `watchBrowser` method normally runs without interruption, the `watchServer` method throws an `IOException` every time the server closes the socket. Obviously, a busy Web server can't afford to keep sockets open for very long, even if the client has requested a "Keep-Alive" connection. The `watchServer` method gets a `null` value from reading the `DataInputStream` when the server socket closes. The inner class `run` method simply restarts the `watchServer` method so it is ready for the next response. Any `IOException` that might be thrown is treated the same way.

Listing 5.4: The watchBrowser and watchServer methods

```
void watchBrowser() throws IOException {
  if( ssok == null ){
    ssok = new ServerSocket( browserPort );
    requestTF.setText("watching port: " + browserPort );
  }
```

```
    browserSocket = ssok.accept();
    System.out.println("watchBrowser got socket");
    OutputStream os = browserSocket.getOutputStream();
    InputStream is = browserSocket.getInputStream();
    toBrowser = new PrintWriter( os );
    BufferedReader br = new BufferedReader( new InputStreamReader( is ));
    String tmp = br.readLine();
    while( running && tmp != null ){
      fromBrowser.addElement( tmp );
      tmp = br.readLine();
    }
  }

  void watchServer() throws IOException {
    serverSocket = new Socket( host, hostPort );
    responseTF.setText( host + ":" + hostPort );
    OutputStream os = serverSocket.getOutputStream();
    toServer = new PrintWriter( os );
    InputStream is = serverSocket.getInputStream();
    DataInputStream dis = new DataInputStream( is );
    // an alternative would be to use a BufferedReader here
    // instead of DataInputStream
    System.out.println("watchServer got socket");
    String tmp = dis.readLine();
    while( running && tmp != null ){
      fromServer.addElement( tmp );
      tmp = dis.readLine();
    }
  }
```

Note that the compiler objects to the deprecated `DataInputStream` class `readLine` method, so you could substitute a `BufferedReader` in `watchServer`. The advantage of the `DataInputStream` class is that `readLine` doesn't perform any Unicode translations; thus, you could switch from reading character lines to reading binary data.

Listing 5.5 shows the constructor for `UtilSnoop`. Because this utility was put together using Visual Cafe, there are some extra comment lines with special formats that the IDE uses to locate certain chunks of code. Essentially, the application shows two panels, one for data from the browser on top and the other for data from the server. Each panel has a `TextArea` object for the message text and a `TextField` used to display the port information. Because `TextArea` objects in Windows systems are limited to about 32kb of character data, this limits the amount of text that can be shown.

Listing 5.5: The UtilSnoop constructor

```
public UtilSnoop()  {
  //{{INIT_CONTROLS
  setLayout(new GridLayout(2,1,0,0));
  setSize(600,400);
```

```
setVisible(false);
reqPanel.setLayout(new BorderLayout(0,0));
add(reqPanel);
reqPanel.setBounds(0,0,20,40);
requestTA.setEditable(false);
reqPanel.add("Center",requestTA);
requestTA.setBounds(0,0,405,305);
reqTopP.setLayout(new FlowLayout(FlowLayout.CENTER,5,5));
reqPanel.add("North",reqTopP);
reqTopP.setBounds(0,0,20,40);
label1.setText("Request text  - from Browser");
reqTopP.add(label1);
label1.setBounds(0,0,100,40);
requestTF.setEditable(false);
reqTopP.add(requestTF);
requestTF.setBounds(0,0,100,40);
respPanel.setLayout(new BorderLayout(0,0));
add(respPanel);
respPanel.setBounds(0,0,20,40);
respTopP.setLayout(new FlowLayout(FlowLayout.CENTER,5,5));
respPanel.add("North",respTopP);
respTopP.setBounds(0,0,20,40);
label2.setText("Response Text - from Server");
respTopP.add(label2);
label2.setBounds(0,0,100,40);
responseTF.setEditable(false);
respTopP.add(responseTF);
responseTF.setBounds(0,0,100,40);
respPanel.add("Center",responseTA);
responseTA.setBounds(0,0,100,40);
setTitle("Snoop Utility");
//}}

//{{INIT_MENUS
menu1.setLabel("File");
menu1.add(loadPropMI);
loadPropMI.setLabel("Load Properties");
menu1.add(separatorMenuItem);
separatorMenuItem.setLabel("-");
menu1.add(exitMI);
exitMI.setLabel("Exit");
mainMenuBar.add(menu1);
menu2.setLabel("Action");
menu2.add(clearMI);
clearMI.setLabel("Clear");
menu2.add(startMI);
startMI.setLabel("Start");
menu2.add(menuItem1);
menu2.add(stopMI);
stopMI.setLabel("STOP");
mainMenuBar.add(menu2);
```

```
  //$$ mainMenuBar.move(0,312);
  setMenuBar(mainMenuBar);
  //}}

  //{{REGISTER_LISTENERS
  SymWindow aSymWindow = new SymWindow();
  this.addWindowListener(aSymWindow);
  SymAction lSymAction = new SymAction();
  exitMI.addActionListener(lSymAction);
  clearMI.addActionListener(lSymAction);
  loadPropMI.addActionListener(lSymAction);
  startMI.addActionListener(lSymAction);
  stopMI.addActionListener(lSymAction);
  //}}
}

public UtilSnoop(String title)  {
  this();
  setTitle(title);
}

public void setVisible(boolean b)  {
  if(b){
    setLocation(50, 50);
  }
  super.setVisible(b);
}
```

The listings continue with the `main` method and some of the utility methods used to initialize the application. Listing 5.6 also includes the declarations of the GUI components in the Visual Cafe format.

Listing 5.6: The UtilSnoop main method

```
static public void main(String args[])  {
  try  {
      (new UtilSnoop()).setVisible(true);
  }  catch (Throwable t)  {
    System.err.println(t);
    t.printStackTrace( System.err );
    //Ensure the application exits with an error condition.
    System.exit(1);
  }
}

public void addNotify()  {
  // Record the size of the window prior to calling parents addNotify.
  Dimension d = getSize();
  super.addNotify();
  if (fComponentsAdjusted)return;
```

```
  // Adjust components according to the insets
  setSize(getInsets().left + getInsets().right + d.width,
       getInsets().top + getInsets().bottom + d.height);
  Component components[] = getComponents();
  for (int i = 0; i < components.length; i++)
  {
    Point p = components[i].getLocation();
    p.translate(getInsets().left, getInsets().top);
    components[i].setLocation(p);
  }
  fComponentsAdjusted = true;
}

// Used for addNotify check.
boolean fComponentsAdjusted = false;

//{{DECLARE_CONTROLS
java.awt.Panel reqPanel = new java.awt.Panel();
java.awt.TextArea requestTA = new java.awt.TextArea();
java.awt.Panel reqTopP = new java.awt.Panel();
java.awt.Label label1 = new java.awt.Label();
java.awt.TextField requestTF = new java.awt.TextField(30);
java.awt.Panel respPanel = new java.awt.Panel();
java.awt.Panel respTopP = new java.awt.Panel();
java.awt.Label label2 = new java.awt.Label();
java.awt.TextField responseTF = new java.awt.TextField(30);
java.awt.TextArea responseTA = new java.awt.TextArea();
//}}

//{{DECLARE_MENUS
java.awt.MenuBar mainMenuBar = new java.awt.MenuBar();
java.awt.Menu menu1 = new java.awt.Menu();
java.awt.MenuItem loadPropMI = new java.awt.MenuItem();
java.awt.MenuItem separatorMenuItem = new java.awt.MenuItem();
java.awt.MenuItem exitMI = new java.awt.MenuItem();
java.awt.Menu menu2 = new java.awt.Menu();
java.awt.MenuItem clearMI = new java.awt.MenuItem();
java.awt.MenuItem startMI = new java.awt.MenuItem();
java.awt.MenuItem menuItem1 = new java.awt.MenuItem("-");
java.awt.MenuItem stopMI = new java.awt.MenuItem();
//}}

class SymWindow extends java.awt.event.WindowAdapter
{
  public void windowClosing(java.awt.event.WindowEvent event)
  {
    Object object = event.getSource();
    if (object == UtilSnoop.this)
      System.exit(0);
  }
}
```

Menu selections are handled by the method shown in Listing 5.7.

Listing 5.7: Menu event handling in UtilSnoop

```
class SymAction implements java.awt.event.ActionListener
{
  public void actionPerformed(java.awt.event.ActionEvent event)
  { Object obj = event.getSource();
    if (obj == exitMI)
      exitMI_ActionPerformed(event);
    if (obj == clearMI)
      clearMI_ActionPerformed(event);
    if (obj == loadPropMI)
      loadPropMI_ActionPerformed(event);
    if (obj == startMI)
      startMI_ActionPerformed(event);
    if (obj == stopMI)
      stopMI_ActionPerformed(event);
    }
}

void exitMI_ActionPerformed(java.awt.event.ActionEvent event)  {
  System.exit(0);
}

void clearMI_ActionPerformed(java.awt.event.ActionEvent event)
{ requestTA.setText("");
  responseTA.setText("");
}
```

Finally, we come to the method used to show a file dialog box and open a properties file (see Listing 5.8). Having a properties file to set the host name and port values makes it easy to use the utility for many different configurations.

Listing 5.8: The method that reads a property file

```
void loadPropMI_ActionPerformed( java.awt.event.ActionEvent event)
{
  FileDialog fd = new FileDialog(this,"Load Properties",
      FileDialog.LOAD );
  fd.show();
  String path = fd.getDirectory();
  String fname = fd.getFile();
  if( path == null || fname == null )return;
  File f = new File( path, fname );
  try{
    FileInputStream fis = new FileInputStream( f );
    Properties p = new Properties();
    p.load( fis );
    setProp( p );
  }catch(IOException e){
```

```
            System.out.println( e.toString());
          }
        }
      }
```

The properties file is very simply formatted, with only three variables as shown in Listing 5.9. Note that the `Properties` class reads lines starting with the pound sign (#) as comments and ignores them.

Listing 5.9: A properties file for UtilSnoop

```
# Chapter 5 snoop utility properties
host=localhost
browserport=9000
hostport=8080
```

Here are some possible extensions to the snoop utility that could make it even more useful:

- Saving the recorded text to a file.
- Sending a request that emulates a browser. You could create a text file containing an example request and add a menu command to read and send the file. This would give you an easily repeatable test case.
- Providing for the capture of binary data. The `watchServer` method could look for Content-Type and Content-Length headers and read the data into a byte array. Note that you would have to change the reading method from `readLine` to one of the byte array reading methods.

Web Server Errors

At the Web server/servlet/JSP engine level of organization, a number of things can go wrong. Hopefully, by the time you read this, Tomcat—the official "reference" implementation of servlets and JSP—will have a detailed set of installation instructions and an online FAQ system for installation problems. Here are some of the reasons people have not been able to get Tomcat running (commercial systems have similar problems):

Out of Date JDK Installation There have been reports of problems with JDK 1.2.0 and 1.2.1, so make sure you have the most recent compiler and standard library files.

Leftover Jar or Class Files If you have been working with the earlier Sun JSDK version, make sure you remove all traces of the previous jar or class files.

Out of Date Linking Software If you are using a servlet engine as an add-on to a commercial server such as IIS, be sure you have the latest DLL or other software that links the Web server to the servlet engine.

Log Files

It is good debugging practice to have a servlet record progress messages to a log file. Unfortunately, there does not appear to be any standardization among servlet engines as to where log files are stored. For example, the JRun 2.3 servlet engine has the rather complex directory structure summarized in Listing 5.10. There are eight different directories named *logs* within this structure. Fortunately, if you are using this engine, you will usually be concerned with only two of these log directories.

Listing 5.10: Schematic diagram of the JRun directory structure

```
|
|-- bin
|-- classes
|-- connectors
|-- examples
|-- jre
|-- jsm-default
|     |-- logs (stdout.log, error.log)
|     |-- properties
|     |-- services
|           |
|           |-- jcp
|           |-- jse
|           |     |-- logs (servlet log method, JSP logs)
|           |     |-- servlets (JSP compiled classes)
|           |
|           |-- jws
|                 |
|                 |-- htdocs
|-- lib
|-- properties
|-- servlets (general servlets directory root)
```

Whichever servlet engine you choose, use your early testing of the installation to get familiar with the location and use of log files. Don't jump into doing your own projects until the examples provided by the vendor work as expected and you understand where the log files are written.

Application Locations and Alias Settings

There is no particular uniformity about the installation and setup of the current generation of Web servers and Java servlet/JSP engines. However, the Servlet 2.2 API documentation makes some clear statements about this subject, so the commercial products might become more uniform.

Sun has whole-heartedly adopted XML for the purpose of configuring applications in the J2EE (Java 2 Enterprise Edition) and the Servlet 2.2 API. This makes for configuration files that are bulky but very easy to read and parse. Still, many users have problems in the configuration area.

Applications have two effective locations: the location as seen by the Web server and the location as seen by the native file system. The Web server converts between request URLs and real files by means of aliases that you, the operator, have to set up. If you get the dreaded "404 - File Not Found" error when trying to access your JSP file, this is the place to start looking. Because different commercial servers have different conventions, I am going to concentrate on the Tomcat server.

Locating HTML and JSP Pages

The Tomcat server uses the `TOMCAT_HOME` environment variable plus context information from the `server.xml` file to locate HTML and JSP files. Listing 5.11 shows the xml-coded text that sets up three separate contexts for URLs, and Table 5.1 shows how the Tomcat server converts URLs using this information plus the `TOMCAT_HOME` value of "c:\tomcat" to create an absolute path. Note that file paths in configuration files use forward slashes—even if your server is running under Windows—to ensure that the files are portable. Java automatically handles path separator differences between platforms.

Listing 5.11: Example Tomcat context settings from server.xml

```
<Context path="/examples" docBase="webapps/examples"
   debug="0" reloadable="true" >
</Context>
<Context path="" docBase="webapps/ROOT" debug="0" reloadable="true" >
</Context>
<Context path="/training" docBase="c:/InetPub/wwwroot/training"
   debug="1" reloadable="true" >
</Context>
```

TABLE 5.1: URL to file path conversion

URL	Resulting path
http://localhost/index.htm	c:\tomcat\webapps\ROOT\index.htm
http://localhost/JSPbook/Chap04/Game.jsp	c:\tomcat\webapps\ROOT\JSPbook\Chap04\Game.jsp
http://localhost/examples/jsp/source.jsp	c:\tomcat\webapps\examples\jsp\source.jsp
http://localhost/training/signin.htm	c:\InetPub\wwwroot\training\signin.htm

In Table 5.1, note that when the server does not have a specific context for a path (as with the "JSPbook/Chap04" URL), the ROOT value is used as a starting point. When the context specifies a complete path (as with the "training" context), the server uses that instead of building one based on TOMCAT_HOME.

The other parameters in each context shown in Listing 5.11 are also significant for debugging. The debug value of "0" minimizes the amount of information the server writes for applications in this path, whereas values up to "9" are more prolific. When you are debugging a new application, if you put it in a separate server context, you can set the level of debugging messages high for this application alone. That way you won't have to wade through debugging messages from other applications.

The reloadable attribute should be set "true" if you want Tomcat to check the timestamp on servlet classes and reload them if a new class file has appeared. This is the normal setting for debugging, but it does take extra time for the engine to check the timestamp with every use.

Locating Servlets

Locating Java servlets follows a convention based on the servlet package. No matter how a servlet engine selects a root directory for servlets, the class file location must use the package naming convention. Let's consider the NumberSoundServ servlet we created in Chapter 4. The Tomcat server will expect to find the servlet class at this absolute path

```
c:\tomcat\webapps\ROOT\WEB-INF\classes\com\JSPbook\Chap04\NumberSoundServ.class
```

which is created by concatenating the following information:

c:\tomcat This comes from the TOMCAT_HOME environment variable.

webapps\ROOT This comes from the context setting (Listing 5.11).

Web-inf\classes This comes from the Sun standard application layout specification.

com\JSPbook\Chap04\NumberSoundServ This comes from the package and servlet class name.

The client Web browser requests the URL as follows:

```
http://localhost/servlet/saynumb.au?digits=123
```

and the Tomcat server looks up "saynumb.au" using configuration data provided by the programmer in the web.xml file for this application, as shown in Listing 5.12. This configuration data gives the complete package and class name.

Listing 5.12: Configuration data in Web.xml for a servlet

```
<servlet>
   <servlet-name>saynumb.au</servlet-name>
   <servlet-class>com.JSPbook.Chap04.NumberSoundServ</servlet-class>
     <init-param>
        <param-name>basepath</param-name>
        <param-value>c:\\tomcat\\webapps\\Root\\JSPbook\\Chap04\\sounds
        </param-value>
     </init-param>
</servlet>
```

If and when you distribute your application in a jar file, that file will have to be in the `Web-inf/lib` directory instead of the `Web-inf/classes` directory.

JSP Debugging Problems

In the Tomcat implementation, JSP pages are handled by a servlet that has its own log file, called "jasper.log" by default. However, because the JSP servlet is run by the normal servlet engine, errors during compilation presently end up in the main "tomcat.log" file. If you are using another servlet and JSP engine, it will probably have different log file conventions. Therefore, the most important advice I can give on JSP debugging is that you should become very familiar with all the logging capabilities of your chosen platform.

JSP Syntax Errors

A JSP page has to go through both translation to a Java servlet and compilation of the servlet before you even get to make runtime errors. You are probably going to be chasing a lot of simple syntax problems. An example that seems to occur a lot has to do with escaping quotation marks in JSP expression statements. The JSP code in Listing 5.13, an example from Chapter 3, gets translated into a single Java `out.write` statement that writes part of a `<form>` tag in HTML. Because the `<form>` tag requires quoted values, a mind-boggling number of escaped quotation marks is required; missing any one of these could cause a translation error, a compilation error, or an error in the resulting HTML page.

Listing 5.13: A JSP expression involving many escaped quotation marks

```
<%= "<form method=\"post\" action=\"Chat.jsp\" > " +
       ChatRoom.getRoomsAsSelect() +
     "<br>Handle: <input TYPE=\"TEXT\" VALUE=\"\"" +
     " name=\"handle\"><br>\r\n" +
     "<input TYPE=\"submit\" VALUE=\"Enter Chat\"" +
     " name=\"chatgo\">\r\n</form></center><hr>\r\n" %>
```

When debugging a statement like this, you have to be very methodical. It also helps to examine the Java code produced by the JSP translation.

Importing Classes

Due to the way JSP pages mix HTML and Java, it is easy to forget to import required classes. The JSP translator creates the necessary import statements based on the code it writes, but you have to provide your own for the code you write. When writing the memory.jsp code used later in this chapter, I got the compilation error shown in Listing 5.14. Note that the error message is formulated in XML, a design decision by Sun that will make creation of more advanced tools for JSP much easier.

If you wade through the verbiage, you can see that the compiler error is reported as an inability to find the `Date` class in the Java class with the astonishingly long name that Tomcat makes up when translating JSP pages.

Listing 5.14: An example JSP compilation error

```
<l:ctx path="" ><b>Internal Servlet Error:</b><br>
<pre>
org.apache.jasper.JasperException: Unable to compile class for JSPC:\tomcat➥
\work\localhost_8080\_0002fJSPbook_0002fChap_00030_00035_0002fmemory➥
_0002ejspmemory_jsp_0.java:87: Class JSPbook.Chap_00030_00035.Date not found.
                out.print( new Date().toString() );
                               ^
1 error

  at org.apache.jasper.compiler.Compiler.compile(Compiler.java:240)
  at org.apache.jasper.runtime.JspServlet.loadJSP(JspServlet.java:414)
  at org.apache.jasper.runtime.JspServlet$JspServletWrapper.loadIfNecessary➥
  (JspServlet.java:149)
  at org.apache.jasper.runtime.JspServlet$JspServletWrapper.service➥
  (JspServlet.java:161)
  at org.apache.jasper.runtime.JspServlet.serviceJspFile(JspServlet.java:261)
  at org.apache.jasper.runtime.JspServlet.service(JspServlet.java:369)
  at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
  at org.apache.tomcat.core.ServletWrapper.handleRequest➥
  (ServletWrapper.java:492)
  at org.apache.tomcat.core.ContextManager.service(ContextManager.java:558)
  at org.apache.tomcat.service.http.HttpConnectionHandler.processConnection➥
  (HttpConnectionHandler.java:160)
  at org.apache.tomcat.service.TcpConnectionThread.run➥
  (SimpleTcpEndpoint.java:338)
  at java.lang.Thread.run(Thread.java:479)
</pre>
</l:ctx>
```

This bug was cured by adding the `java.util` class to the page directive statement as shown here:

```
<%@ page language="java" import="java.util.*" %>
```

After that fix, the JSP translation proceeded without error. It is instructive to look at the `import` statements produced, as shown in Listing 5.15. Some of the imported classes were not used in the actual servlet produced. Apparently, the JSP translator has a set of `import` statements that it automatically uses.

Listing 5.15: Import statements in the Memory.jsp Java code

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.util.*;
```

Design for Debugging

An alternate name for this section would be "Design for Testing," because the basic idea is that if you plan your Java classes so they can be tested in a controlled environment, there will be fewer bugs when the classes are used in a Web server environment. This principle is one reason Sun emphasizes the use of JavaBeans in JSP and servlet applications.

Tools for Beans

As we discussed in Chapter 2, a JavaBean is simply a Java class that meets the following criteria:

- The class must be public.
- The class must have a no-arguments constructor.
- The class must provide `set` and `get` methods to access variables.

The JavaBean component architecture has been around for several years, and many vendors of IDEs and toolkits have adapted their products to work with JavaBeans. Although most of these tools emphasize JavaBeans as GUI components, they also work with beans used in servlets. The central Sun home Web page for JavaBeans and related technology is at:

```
http://java.sun.com/beans/
```

One of the tools available at that site is the "BeanBox," a tool that is mainly intended for use with GUI components but could be used with any bean. If the JavaBean class is serializable, the JSP programmer has the option of either creating a new object from the class file or reading in a serialized object with variables already set.

Catch That Exception!

When working with JSP, remember that you can designate your own custom error page by specifying it in the page directive tag as in this example from Chapter 2.

```
<%@ page language="java" errorPage="/JSPbook/Chapt02/whoops.jsp" %>
```

Any JSP page that is designated as the error page must include a tag similar to the following:

```
<%@ page language="java" isErrorPage="true" %>
```

which sets the `isErrorPage` parameter. This ensures that the page will have a default variable named `exception` that will refer to the actual error or exception.

When working with a servlet, it frequently makes sense to enclose most of the statements in the `doPost` or `doGet` method with a `try - catch` block structure. However, remember that your variable declarations should be made before the `try` block starts; otherwise, they will be out of scope in the `catch` block. Listing 5.16 shows a skeleton of a `doGet` method. Note that it takes advantage of the fact that when the JVM is trying to find a handler for an exception, it will use the first `catch` statement that fits the hierarchy. This allows us to provide special handling for `IOException` and `MyCustomException` exceptions while ensuring that every exception is caught.

Listing 5.16: Outline of a doGet method

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
     throws ServletException, IOException
{
  resp.setContentType("text/html");
  PrintWriter out = new PrintWriter(resp.getOutputStream());
  // various other variable declarations
  out.println("<HTML>");
   .. // outputof page start
```

```
    try { //  do main calculations
       .. // normal output
    } catch( IOException ioe ){
      ... // output debugging related to IO exceptions
    } catch( MyCustomException mce ){
      ... // output specialized debugging information
    } catch( Exception ex ){
        ex.printStackTrace( out );
        out.println("<br>\r\n");
    } finally {
       out.println("</BODY>");
       out.println("</HTML>");
       out.close();
    }
  }
```

Custom Exceptions

Writing your own extension of a standard library `Exception` class to fit your particular application is an excellent way to get more information than the normal exception provides. Your custom exception class can have several different constructors reflecting different possible problems. If you don't want the compiler to force you to write exception handling code, your custom class should descend from `RuntimeException`.

Using Assertions

Although Java does not have assertion capability built in, it is easy to add. The purpose of an assertion statement is to verify that certain conditions are true before the program is allowed to proceed. It serves as a single statement shorthand for logic that would otherwise be expressed with an `if` statement. For example, if a program required a non-null reference named `userName`, you could code it like this:

```
if( userName == null ){
   throw new IllegalArgumentException("bad user name");
}
```

Or using the `Assertion` class, you could code it like this:

```
Assertion.assert( userName != null, "bad user name");
```

Note that with the assertion, the logic test is expected to yield a value of `true` under normal conditions. If the test yields `false`, an `AssertionException` is thrown. Listing 5.17 shows the `Assertion` class source code, and Listing 5.18 shows the code for the `AssertionException` class.

Listing 5.17: The Assertion class has only static methods

```
public class Assertion {
   public static boolean ASSERTION_ON = true ;
   private Assertion() {} ; // no public constructor
   // method without a message
   public static void assert( boolean validFlg )
        throws AssertionException {
     if( ASSERTION_ON && !validFlg ) {
         throw new AssertionException();
     }
   }

  // method with a message
  public static void assert( boolean validFlg, String msg )
       throws AssertionException {
     if( ASSERTION_ON && !validFlg ) {
        throw new AssertionException(msg);
     }
  }
}
```

Listing 5.18: The AssertionException class

```
public class AssertionException extends RuntimeException {
    // this constructor assures that there will always be a message
  public AssertionException(){
    super("AssertionException");
  }

  public AssertionException(String msg ){
    super( msg ) ;
  }
 }
```

You may reasonably ask, why use a special class when the same logic could be accomplished with an `if` statement? One answer is that using assertions makes the purpose of the logic very clear. You can tell at a glance that the program would be interrupted at that point if the condition is not met, whereas an `if` statement could be used for any purpose.

If you use assertions correctly, so that the program does not depend on a result caused by an assertion statement, then you can confidently remove the assertion statements when you are sure that the program is running correctly. Personally, I prefer to comment out the assertion statements so that they stay in the source code as a reminder of what was checked during program development.

Monitoring

Well, you got your application running in your `localhost` test system. It gives the right results and doesn't throw exceptions, so you must be finished with debugging, right? Wrong, there are still plenty of bugs that can become apparent only after it has been running a while.

For example, you might have a memory leak due if objects are being put into a `Vector` or `Hashtable` and are never removed. If each object is only a hundred bytes, it will take many requests to finally kill the server with a memory error. Your first inkling of a problem may come when you notice that the server is getting slower and slower due to excessive paging of memory.

A `Thread` leak might occur if every request creates a `Thread` to carry out some task but you have neglected to provide a way for the `Thread` to exit the `run` method and die normally. After a few thousand `Thread` objects have accumulated, the application may get very odd.

What you need for this kind of debugging is a way to remotely monitor memory and Thread usage in a running server. That way you can look in from time to time and watch for trends that might indicate a bug. Listing 5.19 shows `memory.jsp`, a simple JSP that can display the current servlet engine `Threads` and Java Virtual Machine memory usage.

Listing 5.19: The memory.jsp source

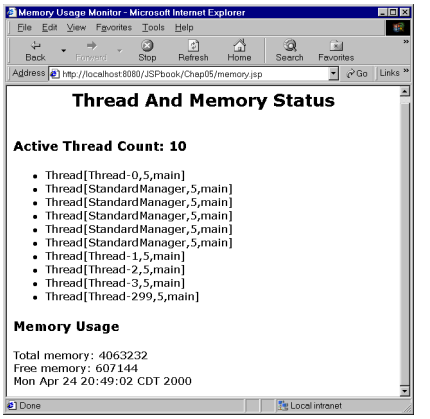
```
<HTML><HEAD>
<TITLE>Memory Usage Monitor</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000"><FONT FACE=VERDANA>
<H2 ALIGN=CENTER>
   Thread And Memory Status
</H2><BR>
<%@ page language="java" import="java.util.*" %>
<%! String brcrlf = "<br>\r\n"; %>
<%
   ThreadGroup tg = Thread.currentThread().getThreadGroup();
   Thread[] thrds = new Thread[ tg.activeCount() ] ;
   tg.enumerate( thrds ); // fills array
%><h3> Active Thread Count:
<%= thrds.length + brcrlf %></h3>
<ul>
<%
   for( int i = 0 ; i < thrds.length ; i++ ){
      out.print("<li>");
      out.print( thrds[i].toString() );
      out.print( brcrlf );
   }
%>
```

```
</ul><h3>Memory Usage</h3>
<%
      Runtime rt = Runtime.getRuntime();
      out.print("Total memory: " + rt.totalMemory() + brcrlf +
                " Free memory: " + rt.freeMemory() + brcrlf );
%>
<%= new Date().toString() %><br>

</BODY>
</HTML>
```

This page uses the fact that all `Thread` objects created in a servlet engine typically belong to the same `ThreadGroup`. The `toString` method displays the name of the `Thread`, its priority, and the name of the `ThreadGroup` it belongs to. Example output from the Tomcat Web server using this JSP is shown in Figure 5.5.

FIGURE 5.5
Output from memory.jsp on a Tomcat server



Note that some of the `Thread` objects shown in Figure 5.5 are named "StandardManager" and some are named "Thread-*n*" where *n* is a sequence number. The sequence number naming style is the default for `Thread` objects created without a programmer-supplied name. If your application creates any `Thread` objects, you should name them so that they will stand out in this listing.

Custom Logging

The main thing to remember when writing a custom logging facility is that there may be many `Threads` executing the same servlet, so log writing methods must be synchronized on the stream object used to write the log. For example, suppose you wanted to add logging to the `ChatRoom` class created in Chapter 3. The code in Listing 5.20 can be added to the `initChatRooms` static method to open the log file in the append mode.

Listing 5.20: Adding a log file to the ChatRoom class

```
try {
    FileOutputStream fos = new FileOutputStream( "chat.log" , true );
    chatLog = new PrintWriter( fos );
}catch(IOException e){
      System.err.println("Unable to create chat.log");
}
```

At a minimum, a custom log should provide for logging a simple message and for logging an exception with a message as shown in Listing 5.21. Note that a call to the `flush` method is needed because `PrintWriter` streams do not flush output by default.

Listing 5.21: Static log writing methods for the ChatRoom class

```
public static void log(String msg ){
  if( chatLog == null ) return ;
  synchronized( chatLog ){
    chatLog.println( msg );
    chatLog.flush();
  }
}

public static void log( String msg, Exception e){
  if( chatLog == null ) return ;
  synchronized( chatLog ){
    chatLog.println( msg );
    e.printStackTrace( chatLog );
    chatLog.flush();
  }
}
```

Miscellaneous Notes

This section contains some debugging hints that I could not figure out a way to categorize. Therefore, I list them here:

- Note that log buffering by the servlet engine means that if you look at the log while the engine is still running, you may not see the latest messages.
- A frequent source of mysterious runtime errors in servlets occurs when the programmer forgets to include a call to `super.init()` in an `init()` method.
- As a precaution, if your servlet does all its work in a `doPost` method, have your `doGet` call `doPost`. That way, if the servlet is ever addressed directly, you will at least get something back instead of seeing a "page not found" error.
- If a program works right the first (n) times but crashes on subsequent tries, look for a resource such as a database connection that has not been released.

- If a JSP or servlet works fine every time you try it, but certain users can never get it to work, the first thing to ask is "does your browser have cookies turned on?"
- When building a Web page with a JSP or servlet, you can avoid confusion about the location of static resources such as image files by using a <base> tag in the <head> tag area of the HTML document. This establishes a base URL for the location of resource files.