3.8 Servlet Debugging

Naturally, when *you* write servlets, you never make mistakes. However, some of your colleagues might make an occasional error, and you can pass this advice on to them. Seriously, though, debugging servlets can be tricky because you don't execute them directly. Instead, you trigger their execution by means of an HTTP request, and they are executed by the Web server. This remote execution makes it difficult to insert break points or to read debugging messages and stack traces. So, approaches to servlet debugging differ somewhat from those used in general development. Here are 10 general strategies that can make your life easier.

1. **Use print statements.**

   With most server vendors, if you run the server on your desktop, a window pops up that displays standard output (i.e., the result of `System.out.println` statements). "What?" you say, "Surely you aren't advocating something as old-fashioned as print statements?" Well, true, there are more sophisticated debugging techniques. And if you are familiar with them, by all means use them. But you'd be surprised how useful it is to just gather basic information about how your program is operating. The `init` method doesn't seem to work? Insert a print statement, restart the server, and see if the print statement is displayed in the standard output window. Perhaps you declared `init` incorrectly, so your version isn't being called? Get a `NullPointerException`? Insert a couple of print statements to find out which line of code generated the error and which object on that line was `null`. When in doubt, gather more information.

2. **Use an integrated debugger in your IDE**.

   Many integrated development environments (IDEs) have sophisticated debugging tools that can be integrated with your servlet and JSP container. The Enterprise editions of IDEs like Borland JBuilder, Oracle JDeveloper, IBM WebSphere Studio, Eclipse, BEA WebLogic Studio, Sun ONE Studio, etc., typically let you insert breakpoints, trace method calls, and so on. Some will even let you connect to a server running on a remote system.

3. **Use the log file.**

   The `HttpServlet` class has a method called `log` that lets you write information into a logging file on the server. Reading debugging messages from the log file is a bit less convenient than watching them directly from a window as with the two previous approaches, but using the log file is an option even when running on a remote server; in such a situation, print statements are rarely useful and only the advanced IDEs support remote debugging. The `log` method has two variations: one that takes a `String`, and the other that takes a `String` and a `Throwable` (an ancestor class of `Exception`). The exact location of the log file is server-specific, but is generally clearly documented or can be found in subdirectories of the server installation directory.

4. **Use Apache Log4J.**

   Log4J is a package from the Apache Jakarta Project—the same project that manages Tomcat (one of the sample servers used in the book) and Struts (an MVC framework discussed in Volume 2 of this book). With Log4J, you semi-permanently insert debugging statements in your code and use an XML-based configuration file to control which are invoked at request time. Log4J is fast, flexible, convenient, and becoming more popular by the day. For details, see http://jakarta.apache.org/log4j/.

5. **Write separate classes.**

One of the basic principles of good software design is to put commonly used code into a separate function or class so you don't need to keep rewriting it. That principle is even more important when you are writing servlets, since these separate classes can often be tested independently of the server. You can even write a test routine, with a `main`, that can be used to generate hundreds or thousands of test cases for your routines—not something you are likely to do if you have to submit each test case by hand in a browser.

6. **Plan ahead for missing or malformed data.**

   Are you reading form data from the client (Chapter 4)? Remember to check whether it is `null` or an empty string. Are you processing HTTP request headers (Chapter 5)? Remember that the headers are optional and thus might be `null` in any particular request. Every time you process data that comes directly or indirectly from a client, be sure to consider the possibility that it was entered incorrectly or omitted altogether.

7. **Look at the HTML source.**

   If the result you see in the browser looks odd, choose View Source from the browser's menu. Sometimes a small HTML error like `<TABLE>` instead of `</TABLE>` can prevent much of the page from being viewed. Even better, use a formal HTML validator on the servlet's output. See Section 3.5 (Simple HTML-Building Utilities) for a discussion of this approach.

8. **Look at the request data separately.**

   Servlets read data from the HTTP request, construct a response, and send it back to the client. If something in the process goes wrong, you want to discover if the cause is that the client is sending the wrong data or that the servlet is processing it incorrectly. The `EchoServer` class, discussed in Chapter 19 (Creating and Processing HTML Forms), lets you submit HTML forms and get a result that shows you *exactly* how the data arrived at the server. This class is merely a simple HTTP server that, for all requests, constructs an HTML page showing what was sent. Full source code is online at http://www.coreservlets.com/.

9. **Look at the response data separately.**

   Once you look at the request data separately, you'll want to do the same for the response data. The `WebClient` class, discussed in the `init` example of Section 3.6 (The Servlet Life Cycle), lets you connect to the server interactively, send custom HTTP request data, and see everything that comes back—HTTP response headers and all. Again, you can download the source code from http://www.coreservlets.com/.

10. **Stop and restart the server.**

    Servers are supposed to keep servlets in memory between requests, not reload them each time they are executed. However, most servers support a development mode in which servlets are supposed to be automatically reloaded whenever their associated class file changes. At times, however, some servers can get confused, especially when your only change is to a lower-level class, not to the top-level servlet class. So, if it appears that changes you make to your servlets are not reflected in the servlet's behavior, try restarting the server. Similarly, the `init` method is run only when a servlet is first loaded, the `web.xml` file (see Section 2.11) is read only when a Web application is first loaded (although many servers have a custom extension for reloading it), and certain Web application listeners (see Volume 2) are triggered only when the server first starts. Restarting the server will simplify debugging in all of those situations.