Using Java Servlets on AIX

By Jeff Jilg, Greg Flurry, and Michael C. Tulkoff

What are Java servlets and how can you use them? What software do you need installed to use servlets and what are some typical configurations? This introductory article answers these questions and others. It includes programming samples that you can easily deploy on AIX within minutes.

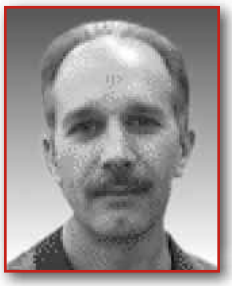
The rising popularity of Java™ has been primarily a client-side phenomenon. Java applets abound in nearly every application type. For server-side programming, Java servlets are easy to code, platform independent, and show relatively good performance.

Java servlets are server-side Java programs that are designed to extend the behavior of the server on which they reside. Servlets are typically, but not exclusively, used to extend Web servers.

Java servlets are similar to Java applets in the way they extend the functionality of Java-enabled Web servers. Since servlets perform server-side processing, they do not have a graphical user interface (GUI). For example, a Java servlet can be deployed on a Java-enabled Web server to receive data from an HTML form and update a database associated with the new information, as shown in Figure 1. In general, servlets are used as a platform-independent replacement for Common Gateway Interface (CGI) programs. CGI replacement servlets are a good alternative because they are easier to write than CGI bins. Also, there is some evidence that servlets perform better than CGI bins. Servlets provide programmers with all the resources that are available in Java, coupled with a mechanism for processing and responding to client requests.

Servlets are not strictly for Web programmers. Sun® designed the servlet model to run in conjunction with a Web server. But it is also possible to leverage the Web server and Java servlet infrastructure to create server-side programs that run in the background. For example, a servlet could be designed to monitor and record transaction types and characteristics on a commerce server. Another servlet might be designed to receive performance alerts from system management software and notify system administrators of possible impending server failure during an overload. The inherited Java methods built into servlets are designed for servlets to receive data and possibly send a response back to the original client.

Now that you understand where servlets can be deployed, you will need more information to create and use them on AIX®. The next section provides a brief overview of the servlet application programming interface (API). Then, we discuss the software you can use on AIX to develop and run servlets, followed by several servlet examples. We included a trivial 10-line example to demonstrate how simple servlets can be created, and a more extensive example that you might consider emulating for real-world



Jeff Jilg

Greg Flurry

Michael C. Tulkoff

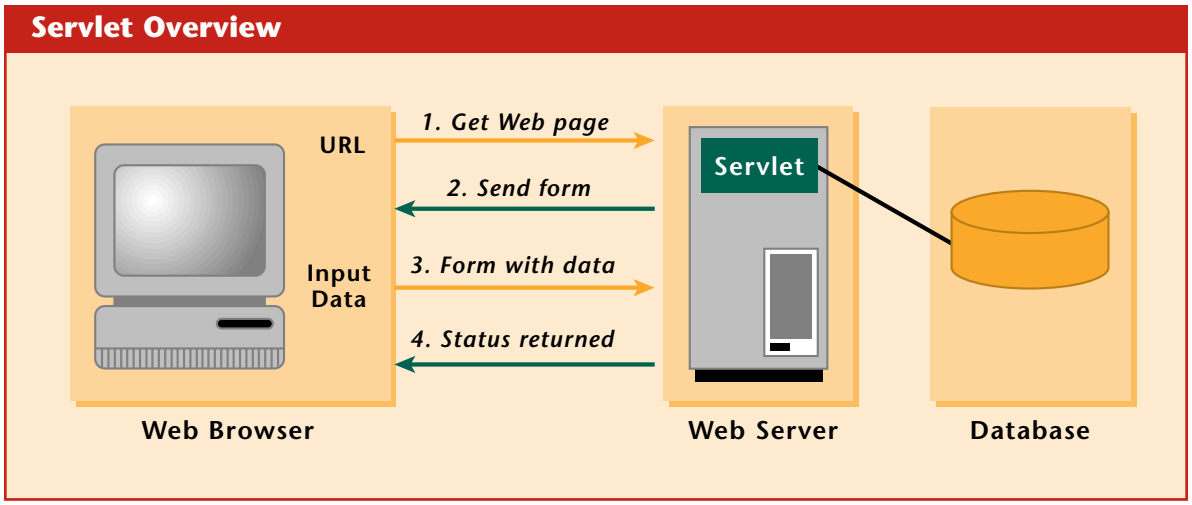


Figure 1. Overview of a Java servlet

programming. The example section is followed by a discussion of servlet programming considerations. We included some hints to help your productivity on AIX and a discussion of some IBM products and programs that exploit Java server-side logic.

Software Requirements and Configuration on AIX

Java servlets typically run from a servlet-enabled Web server. AIX has multiple viable options for the execution environment, which will be discussed below. To create a servlet, you also need a Java Development Kit (JDK) to compile the Java source code that makes up the servlet. Since Java is platform independent, any Java-compatible JDK can be used for development and compilation (that is, AIX, Solaris®, Windows NT™, etc.). The AIX JDK is available on the AIX 4.1 or 4.2 Bonus Packs, the AIX 4.3 Base, or from the IBM Web site (you can download the newest copy from *http://www.ibm.com/java*).

The respective owners of the Web server developer (for example, IBM, Netscape™) must enable Java servlets in that Web server. JavaSoft™ currently provides the enablement kit free of charge. Many servers have been enabled on AIX, including Lotus Domino™ Go, IBM Internet Connection Server, Netscape FastTrack Server™, Netscape Enterprise™, and Apache. Since enablement is fairly easy, others may follow. If your favorite Web server is not listed here, you might check its current status on the Web.

Executing a Java servlet through a Web server usually requires minor reconfiguration, which is discussed below. You should be aware of an important caveat for servlet execution. Some Web servers require the server to be shut down and restarted for the software to recognize servlets, or even new versions of servlets. For example, adding a new servlet to your servlet repository requires that you shut down the Web server and restart the FastTrack Server. You must also shut down and restart the server if you modify and recompile the servlet. *Note:* You can get around this problem by using ServletExpress from IBM, which can fix the problem for the servers supported by ServletExpress.

It is usually not necessary to test multiple Web servers to verify equivalent behavior between different Web servers. Servlets differ from applets in that they are not downloaded to a Web browser. Instead, they run directly on a server. However, if you do run multiple Web servers in your company and you intend to use one machine to do your testing, then you can run two or more servers simultaneously. Just set each one to listen to a different port (for example, set Domino Go to port 80 and FastTrack to port 81).

Depending on the architecture of the Web server, servlets are started either when the Web server starts or when they receive a request. The systems administrator can usually choose to have the servlets started within the Web server process or as a separate process. For example, the Lotus Go™ Web server allows users to modify a configuration file called `servlet.conf` to specify this behavior. It is best to have servlets run as part of the Web server (or servlet server) because it will be much faster—one of the servlets' main advantages over CGI. Much more overhead would be required to use CGI to fire up a Java program compared to using a servlet. This is because CGI would start a unique Java Virtual Machine (JVM) for each Java request. It would be much faster to just redirect the request to a running thread inside the server process.

Domino Go Web Server

The Lotus Go server currently available on the Bonus Pack provides a nearly complete package for servlet development and execution. You also will need a Java JDK for compilation and the AIX Java JDK runtime classes for execution. This packaging allows you to update the Java JDK with the newest version as updates from IBM are available (Web or Bonus Pack).

Lotus Go 4.6.1 includes the Java Just in Time (JIT) compiler-enabled JDK Version 1.1.2. It is fairly easy to configure, requiring only minor changes to the `/etc/httpd.conf` and `/etc/servlet.conf` files, which can be done manually or through the administration screen. If you misplace your Lotus Go password (as we did), the command-line version can be used (`<server_root>/cgi-bin/htadm-adduser`).

By default, servlets can be installed in the directory `/usr/lpp/internet/server_root/servlets/public`. We tested our servlets on AIX 4.2.1 with Lotus Go Versions 4.6.0 and 4.6.1 and found performance to be satisfactory. We found a minor defect in the large `OrderEntry` example servlet, which is discussed further in that section. We attribute the defect to an old (early) version of the Java Servlet Development Kit (JSDK) and found that it occurred in Lotus Go Version 4.6.0, but not in Version 4.6.1.

Netscape FastTrack Server

Enabling servlets is easy on Netscape FastTrack Server Versions 2 and 3.01 and Netscape Enterprise Server Version 2. We tested this enablement on FastTrack 3.01 on AIX 4.2.1. FastTrack currently uses Netscape's own JVM, but you must obtain the JavaSoft JSDK from the Web. As this article went to press, Netscape was considering the use of platform-native JVMs. After installing and configuring FastTrack, the JSDK was downloaded and untarred.

We followed the steps in the JSDK `README.netscape` file as summarized below.

- The `classes.zip` file was unzipped into the `<nshome>/plugins/java/localclasses` directory (where `<nshome>` is the directory that you set up as the FastTrack root directory).
- Java was enabled in the FastTrack administrative interface.
- A few lines were modified in the `<nshome>/https<host>/config/obj.conf` configuration file.
- The configuration changes were activated in the FastTrack administration screen.
- The Web server was restarted to activate the configuration changes.

The JSDK file downloaded from JavaSoft works well on AIX, although it is listed as supporting Solaris. Pure Java classes generally work fine on AIX's Java-compatible Java port. Currently, Netscape uses its own JVM port at the Java 1.02 level. Netscape has stated its intent to support servlets directly in future versions of both Enterprise and FastTrack (see "Notes for Java Programmers" at *http://www.developer.netscape.com/find/index.html*). We hypothesize that they will incorporate and package the corresponding pieces of the JSDK into their servers.

While our Netscape servlet tests were done on FastTrack 3.01, servlets also work on FastTrack Version 2.0 and Enterprise Server 2.0. Although the JSDK `README` file was easy to follow, you also might find the article "Java Servlets in Netscape Enterprise Server" about Netscape's work on servlets helpful (see *http://www.developer.netscape.com/find/index.html*).

Apache

Although we did not test Apache on AIX, servlets can be executed on Apache on AIX. Download both Apache and the JavaSoft JSDK. The JDSK `README.apache` file contains configuration and usage instructions.

ServletExpress Plug-in

ServletExpress enables the use of servlets on AIX without using Lotus Go. ServletExpress is a plug-in that allows existing Web servers to support servlets. For AIX, ServletExpress can provide servlet support in Lotus Go 4.6.1, Netscape Enterprise and FastTrack 2.01 servers, and Apache Server 1.2.4.

ServletExpress provides more than just a servlet execution environment; it also provides a graphical interface for servlet management, additional security features, and dynamic reloading of modified servlets, as well as other features. You can find additional information about ServletExpress at *http://www.ibm.com/java/servexp/*.

Servlet Programming

This section provides an overview of the basic aspects of servlet programming. The JSDK installation steps are described and the typical servlet functional flow is diagrammed. The primary servlet classes and methods are documented prior to their use later in the example section. This section also contains some important considerations concerning threaded Java servlets.

Java Servlet Development Kit

To program servlets, you must first download the Java Servlet Development Kit. The JSDK can be downloaded from the JavaSoft site at *http://jserv.javasoft.com/products/javaserver/downloads/index.html*. For AIX systems, download the Solaris version (compressed `tar` format) of the toolkit, which runs fine on AIX.

After downloading the JSDK, follow these steps:

- Decide the location for the toolkit.
- Issue a `zcat JSDK1.0.1solarisdom.tar.Z | tar xvf .`
- Add the Java classes in the toolkit to your `CLASSPATH`.
- In `ksh`, export `CLASSPATH=$CLASSPATH:<directory>/JSDK1.0.1/lib/classes.zip`.

or

- For `csh`, do a `setenv CLASSPATH $CLASSPATH:<directory>/JSDK1.0.1/lib/classes.zip`.

In addition to the class files needed to run servlets, the toolkit also contains API documentation, an executable called `srun` to run servlets stand-alone without a browser, some sample servlet source code, and the source code for the servlet classes themselves. The toolkit root directory explains this in a `README` file.

> ***A Java servlet can be deployed on a Java-enabled Web server to receive data from a Web form and update a database associated with the new information.***

Servlet Life Cycle

The servlet life cycle begins when the servlet is loaded, shown in Figure 2. As we mentioned earlier, the servlet may be loaded at server initialization time or when the first client request is received by the server. In either case, the server will load the servlet, create an instance of the implemented class, and call the servlet's `init()` method. As with any Java program, you may elect to override `init()`. Although

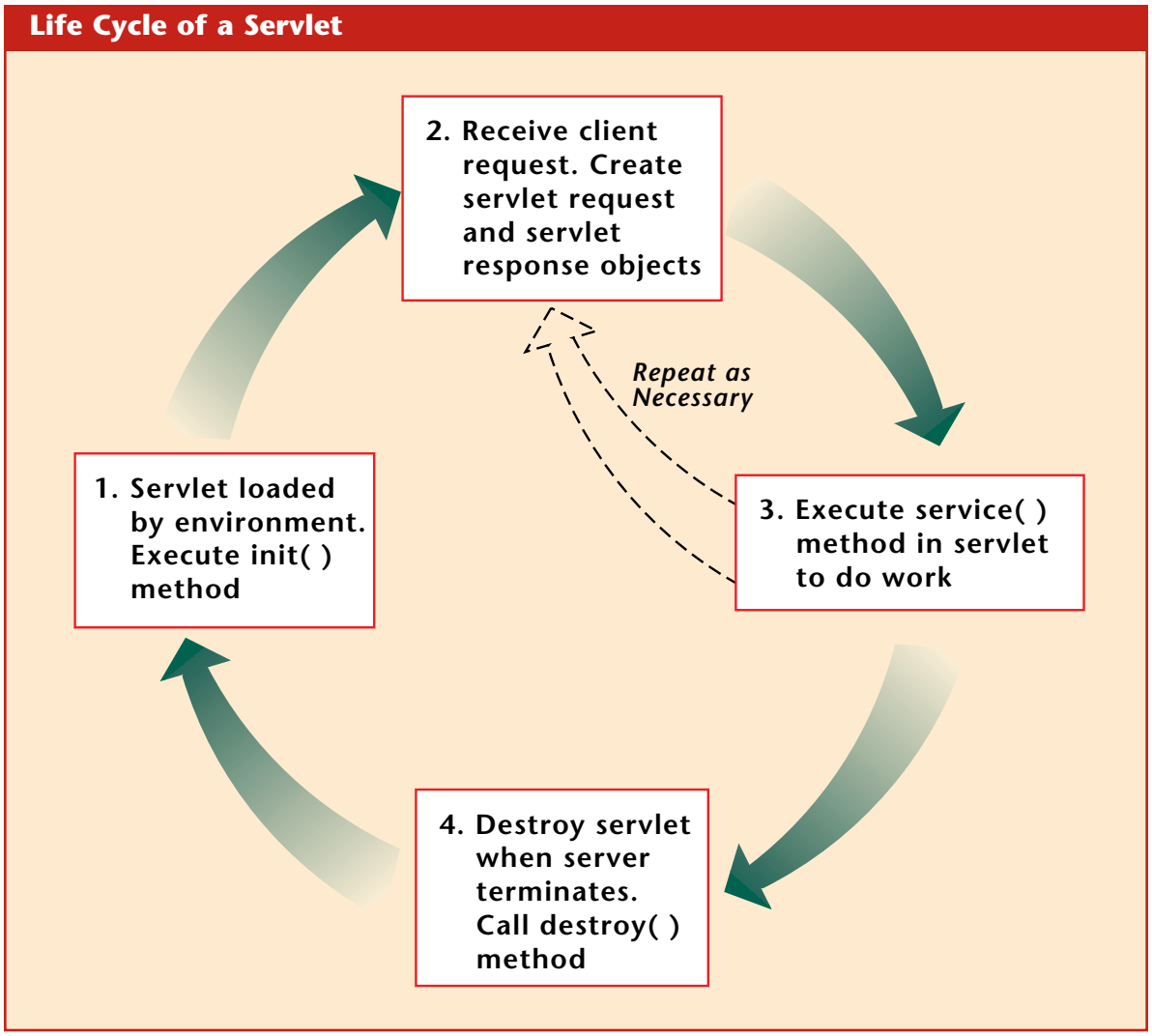


Figure 2. Servlet life cycle in servlet environment

server architectures may vary, typically the servlet will remain loaded until the server is brought down. The `init()` method is only called once per load, not every time a request is received.

The next step is the handling of a client request. Once the server has a client request, it instantiates a special object (contained in the JSDK) called a `ServletRequest` object. The server uses the `ServletRequest` object to pass information from the client to the servlet. At the same time, the server creates a `ServletResponse` object to enable the servlet to communicate back to the client.

The next and most important part of the life cycle is the server invoking the servlet's `service()` method. The `service()` method, which contains the meat of the servlet, is invoked every time a client sends a request. The `service()` method may invoke any other implemented method to process the client request. It will also use the methods in the `ServletResponse` method to interface back to the client.

Eventually, the servlet will be discarded, which typically happens when the server terminates. The servlet contains a default `destroy()` method that is invoked when this occurs. You may override the destroy method if there is any extra accounting or cleanup that you need to do—which is typically not necessary.

Servlet Classes and Methods

When coding a servlet, there are two major classes of concern: the `GenericServlet`

class from the javax.servlet package and the HttpServlet class from the javax.servlet.http class.

The HttpServlet class is actually a subclass of GenericServlet. Both classes implement the servlet interface. You would use the GenericServlet class for a servlet that communicates directly with the client program. For communicating with Web browsers, you will want to use the HttpServlet class, which allows you to create dynamic HTML without clumsy scripting languages. It contains methods that allow the servlet to respond to HTTP GET, POST, and PUT commands, just like CGI.

The GenericServlet class contains ServletRequest and ServletResponse classes. In the HttpServlet class, these are called HttpServletRequest and HttpServletResponse. In both classes, an object of each type is passed as a parameter to the service() method of the servlet. Typically, the servlet would use an InputStream to receive data from the request and an OutputStream to send data to the response. Alternatively, readers and writers could send and receive this data.

GenericServlet Class

The GenericServlet class has the following methods:

- init(): Called on the servlet load
- destroy(): Called when the server is finished with the servlet, usually upon termination
- service(): Called when a request is received; an abstract method that must be overridden and implemented if the servlet is to do any work

Figure 3 contains a shell for a generic servlet.

HttpServlet Class

The HttpServlet class also contains the init(), destroy(), and service() methods. The significant difference between the two classes is that the service() method has already been implemented in the HttpServlet class and does not need to be overridden. In fact, you will not want to override the service() method unless you are adding enhancements to the HTTP protocol or using a later version of HTTP than the one supported by the JSDK (currently it is HTTP 1.0).

In addition to these basic methods, other methods support all of the HTTP protocol commands. These methods should be overridden according to what the servlet will need to handle for a particular application. The default service method will call these HTTP-specific methods automatically when it receives the corresponding HTTP command from the client. These methods include:

- doGet(HttpServletRequest, HttpServletResponse): Handles HTTP GET, conditional GET, and HEAD requests. It returns BAD_REQUEST in default (not overridden) implementation.

```
// Howdy servlet
import java.io.*;
import javax.servlet.*;

public
class HowdyServlet extends GenericServlet {
  public void service(ServletRequest req, ServletResponse res)
       throws ServletException, IOException
  {
    ServletOutputStream out = res.getOutputStream();
    out.println("Howdy Pardner!");
  }
}
```

Figure 3. Shell for a generic servlet

- `doPost(HttpServletRequest, HttpServletResponse)`: Handles `HTTP POST` requests. It returns `BAD_REQUEST` by default.

- `doPut(HttpServletRequest, HttpServletResponse)`: Handles `HTTP PUT` requests. It returns `BAD_REQUEST` by default.

- `doDelete(HttpServletRequest, HttpServletResponse)`: Handles `HTTP DELETE` requests. It returns `BAD_REQUEST` by default.

- `doOptions(HttpServletRequest, HttpServletResponse)`: Handles `HTTP OPTIONS` requests. This method has a default implementation that determines what HTTP options are currently supported and returns a list of these options.

- `doTrace(HttpServletRequest, HttpServletResponse)`: Handles `HTTP TRACE` requests. This method has a default implementation that sends back all of the headers that were contained in the trace request.

- `string getServletInfo()`: Returns whatever string is desired to describe the servlet in case the client application has logic to display information about the servlet. By default, this method returns `NULL` and can be overridden.

HttpServletRequest Class

The `HttpServletRequest` object allows the servlet to access the HTTP client request. The servlet can access the HTTP header information and any arguments passed with the request. It contains the following methods:

- `getParameterValues(string)`: Inherited from the `ServletRequest` class. It will return the parameter values when given the name of the parameter. This method will work with any HTTP method.

- `getParameterNames()`: Inherited from the `ServletRequest` class. It will return the parameter names for the request as an enumeration of strings if they exist, or it returns an empty enumeration.

- `getQueryString()`: Can be used with the `HTTP GET` request to return the string portion of the URL for parsing.

For more complete API information, see the API documentation that ships with the JSDK.

A Word About Threads

By default, the `HttpServlet` class allows multiple threads to execute the service routine concurrently. In other words, every time a client sends a request, the server will execute the `service()` method (including the HTTP method such as `doGet`) in a unique thread. The Lotus Go server, for example, uses a thread-pooling scheme; that is, the needed threads are started beforehand and kept around in a pool. This avoids the thread-creation overhead when the server is started.

When a request is received, it is simply assigned to an available thread and returned to the pool when done. This implies that several copies of the same code can be executing at any given time (at least on a multiprocessing machine).

There are also considerations on a uniprocessor machine. Even though only one thread can be executing at any given time, many threads can be scheduled to execute and a context switch could occur at any time. For this reason, you must be careful about protecting any critical sections of your code.

Accessing a global variable is an example of a critical section. For example, in an application that keeps a running total, client one sends a request that is handled by thread one, which accesses the variable sum and increments it. The code underneath Java loaded the original value and incremented it, but a context switch happens before storing the new value. Client

two has already sent a request, so maybe thread two now tries to increment the value and is successful. Eventually, thread one will execute and complete the increment by storing the value that is no longer correct. Depending on what data is being accessed, this problem could end up a lot worse.

To get around this problem, you must first identify the critical sections, then make them "thread-safe." The easiest way to accomplish this is to synchronize the critical sections. Good Java practice would dictate that you synchronize whole methods instead of lines of code within them.

The following example shows a thread-safe `doGet`:

```
public  synchronized void doGet() {
   // stuff here
}
```

If you wish to protect your entire servlet from running in concurrent mode, you can override the behavior of the `HttpServlet` class by using the `SingleThreadModel` keyword, shown in Figure 4.

```
public class TestServlet extends HttpServlet
   implements SingleThreadModel {
} // end TestServlet
```

Figure 4. Overriding the HttpServlet class

Servlet Examples

Three ways to invoke a servlet include:

- Pointing a browser at the servlet's URL; for example, *http:myserver/servlet/HowdyServlet*
- Referencing the servlet in the `action` parameter of an `HTML <FORM>` tag; for example, `<form method=post action=/servlet/SSIServlet> ... </form>`
- Referencing the servlet in an HTML `<SERVLET>` tag in an `.shtml` file served by a Web server enabled for server-side includes (also known as server-side embeds); for example, `<servlet SSIServlet code=SSI-Servlet.class> ... </servlet>`

We demonstrate all three methods in the examples below. Any good servlet reference will contain more detail about the first two mechanisms. For additional details about server-side includes, see the following Web sites:

- *http://www.ics.raleigh.ibm.com/pub/icswp004.html*
- *http://jserv.javasoft.com:80/products/javaserver/documentation/webserver1.0.2/servlets/servlet_tutorial.html*
- *http://www.ics.raleigh.ibm.com/pub/wpgl0mst.htm.*

Compiling and Running Servlets

Once a servlet is coded, it must be compiled. Although the following information is specific to Lotus Go, the process is similar on other servers. Make sure that the `CLASSPATH` points to class libraries for both the JDK and the JSDK. Set the `CLASSPATH` environment variable or use the `classpath` flag on `javac`. For example, assume you installed JSDK separately on your system:

```
setenv CLASSPATH=/usr/lpp/internet/
  server_root/java/lib:/jsdk/lib/
  classes.zip
javac MyServlet.java
```

After compilation, you must copy `MyServlet.class` to the `<server_root/>servlets/public` directory or the directory appropriate for your Web server configuration (`/usr/lpp/internet/server_root/servlets/public` default for Lotus Go). Then you must restart your Web server.

To run a servlet via a URL, start a Web browser, such as Netscape Navigator™. Point the browser to *http://HOSTNAME:PORT/servlet/MyServlet* (where *HOSTNAME:PORT* is your server, *myserver:80*). The browser contacts the Web server that invokes the `service()` method of the servlet, which can perform a variety of actions based on programming tasks desired.

HowdyServlet via URL

The `HowdyServlet` is a trivial example (shown in Figure 5) that demonstrates the simplicity of servlet programming. Compile and run it using the information above. You should be able to access it easily by entering the following URL into a Web browser: *http://HOSTNAME:PORT/servlet/HowdyServlet*. The browser will indicate a reply "`Howdy Pardner!`"

SSIServlet via <servlet> Tag

Another method for accessing servlets is through the server-side include (SSI) protocol in Web servers. Figure 6 shows the `ssi.shtml` file used to reference the servlet, and Figure 7 lists a simple servlet that is invoked as the result of the Web server processing the `<servlet>` tag. The servlet is simple to invoke. Just enter into a Web browser *http://HOSTNAME:PORT/ssi.shtml*.

```
// Howdy servlet
import java.io.*;
import javax.servlet.*;

public
class HowdyServlet extends GenericServlet {
  public void service(ServletRequest req, ServletResponse res)
       throws ServletException, IOException
  {
    ServletOutputStream out = res.getOutputStream();
    out.println("Howdy Pardner!");
  }
}
```

Figure 5. HowdyServlet.java

```
<html><head><title>Testing SSI</title></head>
<body>
<h2>The SSI example</h2>
<servlet name=SSIServlet code=SSIServlet.class>
</servlet>
</body></html>
```

Figure 6. ssi.shtml

```
import java.io.*;

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

public
class SSIServlet extends GenericServlet {
  public void service (ServletRequest req, ServletResponse res)
       throws ServletException, IOException
  {
    ServletOutputStream out = res.getOutputStream();
    out.println("Server side includes work!");
  }
}
```

Figure 7. SSIServlet.java

```
<html>
<head><title>Forms Servlet Test</title></head>
<body>
<h2>Order Selection</h2>
<p>Please select the items you wish to order, then press Submit:</p><hr>
<form action=/servlet/OrderEntryServlet method=POST>
<p>Enter name: <input name=theName></p>
<p>Department Billed:
<input type=radio name=theDept value="D23" checked>D23
<input type=radio name=theDept value="D24" >D24
<input type=radio name=theDept value="D25" >D25
<p>Select items required: <br>
<input type=checkbox name=Pen >Pen @ $5.25<br>
<input type=checkbox name=Pencil >Pencil @ $0.25<br>
<input type=checkbox name=Eraser >Eraser @ $1.15<br>
</p>
<p> <input type=submit value=Submit Order></p>
</form><hr>
</body></html>
```

Figure 8. OrderEntry.html

OrderEntry Servlet via HTML

You can access servlets from within HTML forms. The next section describes the `OrderEntry` servlet and its logic flow. The `OrderEntry.html` file (Figure 8) should be invoked via *http://HOSTNAME:PORT/OrderEntry.html*.

The Web browser will display the corresponding HTML and invoke the `doPost()` method of the `OrderEntryServlet` (overrides the Web server `POST`), shown in Figure 9. The logic flow in the following section describes the servlet processing of the form information.

While it may seem easy to put static HTML in a form on the `OrderEntry.html` text, this is a limiting mechanism. The HTML contains hard-coded entries, such as the department values D23, D24, and D25. To avoid this, use Java servlets, which can dynamically obtain this data from files or databases. The next example demonstrates this capability.

OrderEntryServlet via URL

Figure 9 shows the source code for a rudimentary order-entry servlet. The `OrderEntryServlet` extends the `HttpServlet` class in order to leverage its built-in HTTP handling framework. `OrderEntryServlet` is invoked by a URL *http://HOSTNAME:PORT/servlet/OrderEntryServlet*.

It dynamically generates an HTML form asking for customer information and product choices (see Figure 10). Once the user submits the completed form, `OrderEntryServlet` processes the information from the form and confirms the order with additional dynamically generated HTML.

A detailed explanation of this `OrderEntryServlet` logic flow is described below. See Figure 10 to follow this logic. It is also helpful to compile and run the example to see it in action.

When the browser issues an `HTTP GET`, the Web server loads `OrderEntryServlet` and calls `OrderEntryServlet.init()`. The Web server then calls `OrderEntryServlet.doGet()` and it generates the HTML form. The Web server returns the form to the browser, which displays it. Once you complete the form and press the Submit button, the browser again contacts the Web server and issues an `HTTP PUT`. The Web server then calls `OrderEntryServlet.doPost()`, and it gathers the parameters from the form, calculates the total cost of the selected products, and generates the HTML for the response. The Web server returns the response to the browser, which displays it.

```
 /*  OrderEntry.java */

import java.io.*;
import java.util.*;
import java.text.*;

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

public
class OrderEntryServlet extends HttpServlet {

  String prodName[];
  double prodPrice[];

  final String deptName[] = {"D23", "D24", "D25"};
  int i;
  String oName, dName;

  public void init(ServletConfig config)
 throws ServletException
  {
    final String pName[] = {"Pen", "Pencil", "Eraser"};
    final double pPrice[] = {5.25, 0.25, 1.15};

    super.init(config);

    // initialize "database"
    prodName = pName;
    prodPrice = pPrice;

    // open connection to transaction engine (not really in this example)
  }

  public void destroy() {
    // close connection to transaction engine -(not really in this example)
  }

  public void doGet (HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException
  {
    // set up initial html
    res.setContentType("text/html");
    ServletOutputStream out = res.getOutputStream();
    out.println("<html>");
    out.println("<head><title>Forms Servlet Test</title></head>");
    out.println("<body>");
    out.println("<h2>Order Selection</h2>");
    out.println("<p>Please select the items you wish to order, then press Submit:</p><hr>");

    // set up the form - first do name
    out.println("<form action=/servlet/TestServlet method=POST>");
    out.println("<p>Enter name: <input name=theName></p>");
    // do department information
    out.println("<p>Department Billed: ");
    for (i=0; i<deptName.length; i++) {
      if(i==0) { // make the first one the default
        out.println("<input type=radio name=theDept value=" + deptName[i] + " checked>" + deptName[i])
;
      }else{
        out.println("<input type=radio name=theDept value=" + deptName[i] + ">" + deptName[i]);
      }
```

(continued on next page)

Figure 9. OrderEntry.java

(continued from previous page)

```
    }
    // do items orderable
    out.println("<p>Select items required: <br>");
    for (int i=0; i<prodName.length; i++) {
      out.println("<input type=checkbox name=" + prodName[i] + ">" + prodName[i] + "&#09&#09@ $
          " + prodPrice[i] +"<br>");
    }
    out.println("</p>");
    // do submit button
    out.println("<p><input type=submit value=Submit Order></p>");
    // end form
    out.println("</form><hr>");
    out.println("</body></html>");
  }

  public void doPost (HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException
  {
    // set up initial part of response
    res.setContentType("text/html");
    ServletOutputStream out = res.getOutputStream();
    out.println("<html><head><title>Forms Servlet Test</title></head>");
    out.println("<body><h2>Order Results</h2><hr>");

    // Get client's form data & process
    double total = 0.0;
    Enumeration values = req.getParameterNames();
    while (values.hasMoreElements()) {
      String zname = (String)values.nextElement();
      String zvalue = req.getParameterValues(zname) [0];
      // String zvalue = req.getParameter(zname);  USE this on older JSDKs
      if (zname.equals("theName")) {oName = new String(zvalue);}
      else if (zname.equals("theDept")) {dName = new String(zvalue);}
      else for (i=0; i<prodName.length; i++) {
        if (zname.equals(prodName[i])) {
          total = total + prodPrice[i];
        }
      }
    }
    // at this point the servlet would create a transaction for the order

    // format response
    NumberFormat nf = NumberFormat.getInstance();
    nf.setMaximumFractionDigits(2);
    nf.setMinimumFractionDigits(2);
    nf.setMaximumIntegerDigits(2);
    nf.setMinimumIntegerDigits(1);
    String oTotal = nf.format(total);
    out.println("<p>Thank you for your order " + oName + ". ");
    out.println("Your department, " + dName + ", will be charged $" + oTotal + ".</p>");
    out.println("<hr></body></html>");
  }

  public String getServletInfo() {
    return "A servlet that shows HTML processing via GET and POST";
  }
}
```

Figure 9. OrderEntry.java

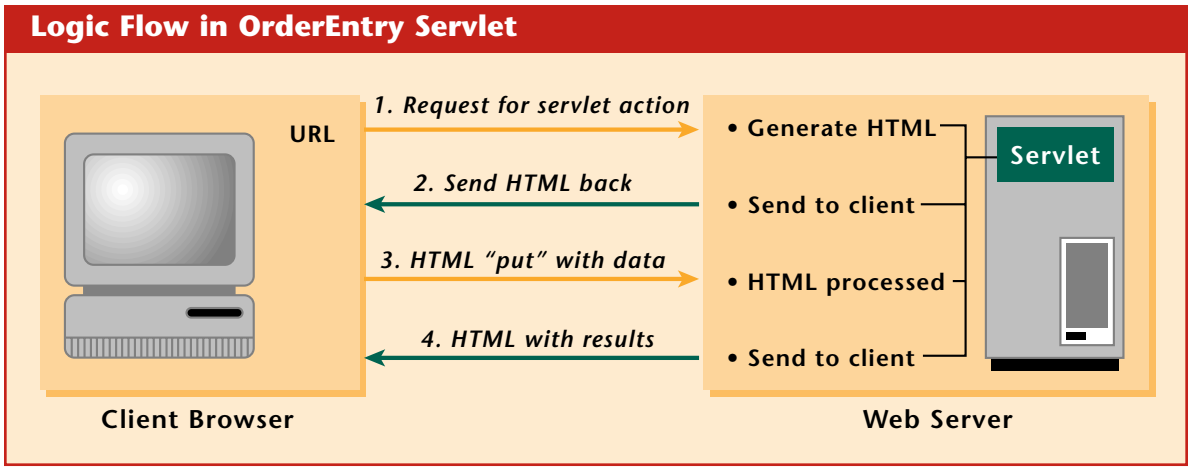


Figure 10. OrderEntry servlet logic flow

The `OrderEntryServlet.init()` method simulates opening a database and populating an array of product information. In this example, the information comes from static data; in a real-world servlet, `init()` would open a database to retrieve the product information and would also open a connection to some transaction engine on a third-tier server. The `destroy()` method in this example does nothing, because there is nothing to do; in a real-world servlet, `destroy()` would close its connection to the transaction engine.

The `doGet()` method, invoked when a browser does an `HTTP GET` operation referencing `OrderEntryServlet`, generates an HTML page containing the form for collecting order information. It does this by writing the HTML to a `ServletOutputStream`. The form calls for a `POST` response, and directs the `POST` to `OrderEntryServlet` using the HTML statement `<form action=http://HOSTNAME:PORT/servlet/OrderEntryServlet method=POST>` for our testing with Lotus Go.

The action may be somewhat different, depending on your Web server configuration requirements. The information used to construct the form statement can also be parameterized at initialization time. The form first asks for a customer name via a text input field, then a department number via radio button, product selections via a set of check boxes, and finally presents a "Submit" button.

The `doPost()` method, invoked when the user hits the Submit button, collects the order information, processes it, and reports the order results to the user. To report the results, like `doGet()`, it generates an HTML page containing the response—in this case, the total cost of the order—by writing the HTML to a `ServletOutputStream`. Then, `doPost()` first sets the total cost of the order to $0.00 and iterates through the parameters passed back via the form. It then collects the value of the user name and department, saving the returned strings.

For the products, it simply determines if a product was selected or not (if a product is not selected, the parameter name does not appear in the enumeration returned by `getParameterNames()`); when a product is selected, `doPost()` adds its price to the total. At this point, `doPost()` in a real-world servlet would create a transaction for the order, and perhaps retrieve additional information, such as the expected delivery date.

Finally, `doPost()` formats the response to the order form by simply echoing the name and department entered by the user and displaying the total cost of the items ordered; in the real world, any additional information about the transaction would also be displayed in the response.

As mentioned earlier in the article, older JSDKs have a bug that does not allow proper array processing. This bug is highlighted in the `doPost()` method on the

`String zvalue` line. We have indicated a way to get around this bug on older JSDKs when you only have a single element to extract from the array.

This simple example does not demonstrate some additional considerations for writing servlets, such as processing initialization parameters, dealing with multithreaded synchronization issues, or exception handling. For a good discussion of such issues, see "Servlet Tutorial" at *http://java.sun.com/products/jdk/1.2/docs/ext/servlet/servlet_tutorial.html*.

Real-World Servlet Deployment

The most visible deployment of servlets within IBM is in IBM's Network Computing Framework (NCF). The NCF is an open, pluggable, standards-based, multi-platform framework for creating, deploying, and managing e-business solutions across the enterprise. The elements of NCF include:

- A set of Web application servers supporting an Object Request Broker (ORB) and a JVM
- A standardized interface to data and materials available on a Web server via a Web browser/Java applet model
- A Java programming model based on applets, servlets, and JavaBeans™
- Internet-ready protocol support, such as HTTP and Internet Inter-Orb Protocol (IIOP), which links JavaBeans components
- Middleware development tools for application development and content authoring
- A set of "connector" services that provide access to existing data, applications, and external services
- A set of built-in collaboration, commerce, and content services that provide a foundation for an industry of partner-built solutions and customizable applications for e-business

In NCF, servlets provide the anchor for every application. NCF supports servlet interaction via HTTP, HTTPS, and IIOP. HTTP interactions proceed much as described above. HTTPS interactions bring in additional security using public key encryption techniques. With IIOP, an applet running on the client can establish an IIOP connection to a servlet in the Web server and pass method calls back and forth, permitting a much richer and more object-oriented interaction.

AIX is one of the primary platforms for NCF deployment. For additional information about the NCF, see *http://www.software.ibm.com/ebusiness/ncf/*.

***Jeff Jilg**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758.E-mail: jjilg@austin.ibm.com. Dr. Jilg is a senior AIX system architect currently responsible for AIX Java technical strategy. His MS in Computer Science from the University of Texas at El Paso complements his PhD from Texas A&M University in the same field.*

***Greg Flurry**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Flurry is a senior technical staff member in the RS/6000 Division. His responsibilities, as part of the Systems Architecture & Technical Strategy team, include network computing and Java. He has a BS in Electrical Engineering from Vanderbilt University and an MS in the same field from the University of Kentucky.*

***Michael C. Tulkoff**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Tulkoff is an IBM advisory software engineer. He has worked as both an AIX developer and AIX system architect. He is currently working on Business Discovery Solutions in the Global Business Intelligence Solutions industry group, where he is heavily using Java and Web-based programming. Mr. Tulkoff holds a BS in Computer Science from the Georgia Institute of Technology.*