WHY I FINALLY LEARNED TO LOVE SERVLETS

James W. Cooper

I’ve been deeply immersed in Java for several years now, and the idea of returning to clunky HTML forms and JavaScript didn’t really appeal to me. However, there are some cases where a Java applet is either inappropriate or overkill for a particular system’s web pages. Some of the reasons that have occurred to me include

1. We only want to return static data from the server to the client.
2. There are no expansions or multiple views of that data.
3. The number of user interactions is small and simple.
4. The user community has a diverse and unpredictable set of browsers.
5. RMI is not tractable because of problems with firewalls, or religious issues.

So, in a nutshell, there are cases now and in the foreseeable future where a simple forms-based user interface is the easiest way to get information to users via the web.

So, if we’re just going to write a simple query form, where’s the beef (or Java)? Well, let’s not forget the server.

Let’s suppose we have a typical application for an HTML form page, where we want to look up a list of people in an organization, or find out someone’s phone number or address. We type the name into a field, check off some buttons restricting the search and click on the “submit” button. As you probably know, this sends a data stream back to the server in the form of a CGI (Common Gateway Interface) request. Usually, that request includes a stream of these selectable parameters from the web page, and the name of an executable program that the server is to run. That program then will parse the rest of the stream, act on it, look up the answer in a table or database, and generate an HTML output page to send back to the client.

As usual, the devil is in the details. Parsing that convoluted stream is a bit of work, although there are lots of C libraries and perl scripts that can do it. There is also a package of Java routines that can handle these CGI data streams very efficiently. These are available as a set of jar files in the Java Servlet Development Kit (JSDK). A Java servlet is a small Java program that can extend the HTML request and response service and parse that CGI stream using convenient high-level classes and methods. Then it can call any other Java methods or use any other available data to produce an output HTML page.

Now, when you write a CGI-script in other languages, you point the browser to the actual program that will parse that form’s data stream:

http://someserver/cgi-bin/getData.pl

or

http://someserver/cgi-bin/getData.exe

and the web server knows that it is to execute these programs. Since Java programs are executed through a Java interpreter program, you can’t just point to a class file and have it executed. Instead, most web servers (except Microsoft’s IIS) can be configured to launch Java servlets. But even though you may be running IIS on Windows NT, you can still use servlets. There are a number of solutions we'll discuss later below.

Why Bother with Servlets?

If you've never written a C program or perl script for handling a CGI request, the reasons for using a servlet are obvious: it's in Java. Even if you have written such programs in the past you'll quickly realize that you didn't really want to revisit all of the inelegancies of those two languages compared to Java.

There are some more substantive reasons for using servlets, however. Since the servlet running program keeps the Java servlet loaded once it is requested, future requests from that or from other clients will run much faster than C or perl programs which must reload the entire program infrastructure once they are called. Finally, writing servlets in Java is so simple, it is silly to use more complicated approaches. Occam's razor strikes again!

One reason I had never bothered with servlets before is that in my work environment, I had a great network and up to date software, and could count on people being able to use Java applets and RMI. So even though I controlled the server, the nature of my research did not ever encompass the simple static lookups that CGI programs excel at.

At home, I maintained a few web sites as a hobby, but the server was at a web hosting ISP in Virginia, and I had no way to install and test servlets. Recently, however, I was able to obtain a cable modem, and have a direct internet connection. While I couldn't run a huge web site on this system, I certainly could answer a few database queries.

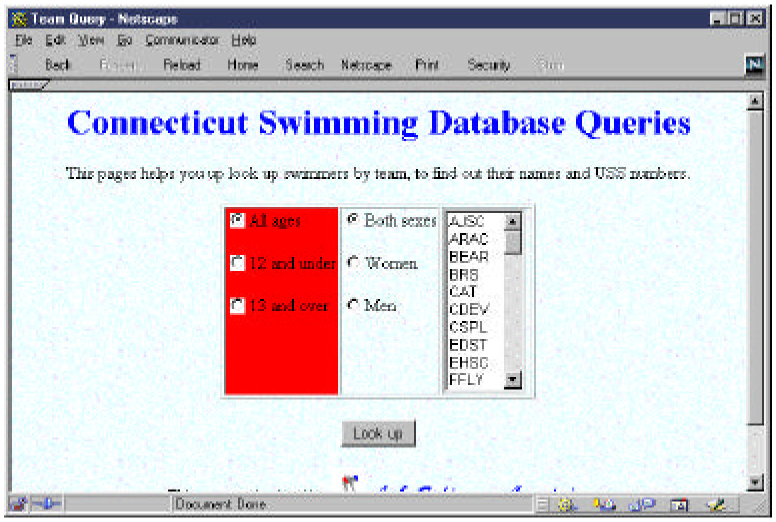
The Problem Statement

I was presented with the simple problem of providing a web site for Connecticut Swimming, where users could look up the correct name and membership ID number of all the swimmers on each team. This is not just a static list problem, because both the teams and their members change quite often.

This seemed like an ideal case for a simple CGI query and servlet connection to a database of all the swimmers in the state. As I saw it, there were two screens:

1. Show a list of teams to select from.
2. Show a list of swimmers on the selected team.

Both of these screens require database queries. For the first, we need to obtain a current list of teams, and for the second a list of swimmers for the selected team. So both screens must be generated dynamically by servlets. The team selection screen is shown below:


Team Query - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Look up
Document: Done

Connecticut Swimming Database Queries

This pages helps you up look up swimmers by team, to find out their names and USS numbers.

[✓ All ages] 12 and under 13 and over

[✓ Both sexes] Women Men

AJSC ARAC BEAR BRS CAT CDEV CSPL EDST EHSC FFLY

The form components in this table are defined for the age buttons as:

```
<input type="radio" name="Ages" value="All" >All ages
<input type="radio" name="Ages" value="12U"> 12 and under
<input type="radio" name="Ages" value="13O"> 13 and over
```

and for the sex selection buttons as

```
<input type="radio" name="Sex" value="Both">Both sexes
<input type="radio" name="Sex" value="F">   Women
<input type="radio" name="Sex" value="M">   Men
```

Note that all 3 radio buttons of the first group are named *Ages* and all three of the second group are named *Sex*. It is these group names that we use to find out which button has been selected.

Writing the Servlet

When you send data from a form to a web server, you use either the GET or the POST method. Originally, these were quite different in implementation with POST being more robust. However, in servlets, they are essentially identical and you can use either one interchangeably. The beginning of the Form section of the web page specifies the posting method and the machine to send the requests to:

```
<form action=http://blahblah.com:8080/servlet/CTSwim method=GET>
```

This line sends the contents of the form to port 8080 (the default servlet port) of the server *blahblah.com*, specifies the CTSwim servlet and uses the GET method to send the form data to the servlet. The web server tells the servlet runner system to load this CTSwim program, initialize it and then calls its *doGet* method.

The actual servlet code is very simple, and you can easily learn to write one from the examples supplied with the Java Servlet Development Kit (JSDK).

```
public class CTSwimServlet extends HttpServlet
      implements SingleThreadModel {

      protected Database db;        //database we talk to
```

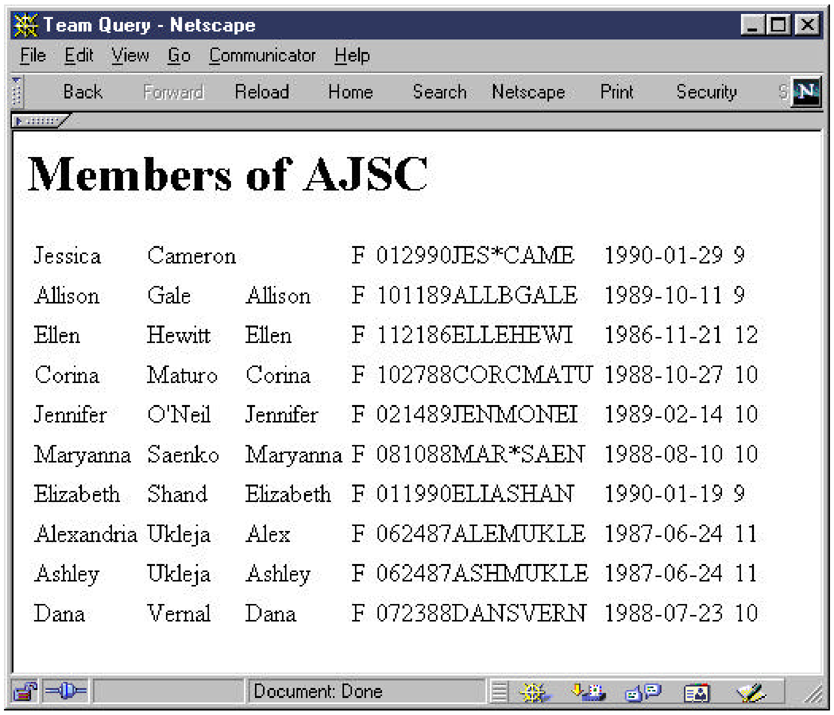
```
   public void init(ServletConfig svg)
                throws ServletException{
       //open database during initialization
      db = new Database("sun.jdbc.odbc.JdbcOdbcDriver");
      db.Open ("jdbc:odbc:CTSwim99",null);
}
   //-----------------------------------------------
   public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
   }
}
```

In the above example, we see a an *init* method, which is called once when the servlet engine first loads and initializes your servlet. Just as in applets, the *init* method take the place of the constructor. This is an ideal place to perform one-time initialization tasks such as connecting to a database or opening special template files.

The *doGet* method is called when a GET method is called. Here, we get the parameters by name to decide the nature of the query to send to the database. Each of the radio buttons has the same *name* parameter, but a different *value* parameter. The three values for the *Ages* buttons are *All*, *12U* and *13O*. For example we can set the age range as follows:

```
       //12 and under
       if (req.getParameter("Ages").equals("12U")) {
            maxage = 12;
            minage = 0;
        }
       //13 and over
        if (req.getParameter("Ages").equals("13O")) {
            maxage = 100;
            minage = 13;
        }
```

Then we can construct the query to the database and generate the output HTML file as shown below:


Team Query - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security
Document: Done

Members of AJSC

Jessica	Cameron		F	012990JES*CAME	1990-01-29	9
Allison	Gale	Allison	F	101189ALLBGALE	1989-10-11	9
Ellen	Hewitt	Ellen	F	112186ELLEHEWI	1986-11-21	12
Corina	Maturo	Corina	F	102788CORCMATU	1988-10-27	10
Jennifer	O'Neil	Jennifer	F	021489JENMONEI	1989-02-14	10
Maryanna	Saenko	Maryanna	F	081088MAR*SAEN	1988-08-10	10
Elizabeth	Shand	Elizabeth	F	011990ELIASHAN	1990-01-19	9
Alexandria	Ukleja	Alex	F	062487ALEMUKLE	1987-06-24	11
Ashley	Ukleja	Ashley	F	062487ASHMUKLE	1987-06-24	11
Dana	Vernal	Dana	F	072388DANSVERN	1988-07-23	10

Sending the output just amounts to writing data to the output channel we can obtain from the *HTTPServletResponse* object.

```
        PrintWriter out = res.getWriter();
        res.setContentType("text/html");

        out.println("<html>");
        out.println("<head><title>Team Query</title></head>");
        out.println("<body>");

        String team = req.getParameter("teams");     //get team name
        out.println("<h1>"+"Members of "+team+"</h1>");
```

Then we query the database and write out a table a line at a time with the various database fields we need as table cells.

Are There Really Two Servlets?

The first screen we showed above shows a list box of team initials and radio buttons for age and sex. Since the teams change frequently, that list must be generated from the database each time a query begins. Thus, it would seem that we need to create that page with the current team list using a second servlet.

On the other hand, these two pages share a common database and really amount to separate queries against that database. So, how can we easily decide what do if there is only one servlet? The simplest way is to use a hidden parameter on the querying web page

```
<input type=hidden name=servletType value=MakeTeams>
```

and have a hidden parameter having the *servletType* name but a different *value* on each page that calls your servlet. You obtain the value of this parameter for each request using the *getParameter()* method of the HttpRequest object. Then the servlet can check to see which methods are to be called and dispatch them appropriately. This leads to some simplifications in our servlet structure. Both of these servlet calls use many of the same methods, and we could simply check the value of the hidden *servletType* parameter and call the right method in a single servlet class, but it is cleaner and more scalable to make each of these methods a separate class.

Let’s consider a base class called *ServletProcessor* which contains at least the following:

```
public abstract class ServletProcessor {
    protected HttpServletRequest req;
    protected HttpServletResponse res;
    protected Results rs;
    protected Database db;
    protected PrintWriter out;

    public void setHttp(HttpServletRequest rqst,
             HttpServletResponse rsp, Database dbase) {
        req = rqst;
        res = rsp;
        db = dbase;
    }
    //-------------------------------------------
    public abstract void Execute(PrintWriter outPrint);
    //-------------------------------------------
    protected void print (PrintWriter out,
             String name, String value)  {
        out.print(" " + name + ": ");
        out.println(value == null ? "&lt;none&gt;" : value);
    }
  //--------------------------------------------
    protected void print (PrintWriter out,
             String name, int value)     {

        out.print(" " + name + ": ");
        if (value == -1) {
            out.println("&lt;none&gt;");
        } else {
            out.println(value);
        }
    }
}
```

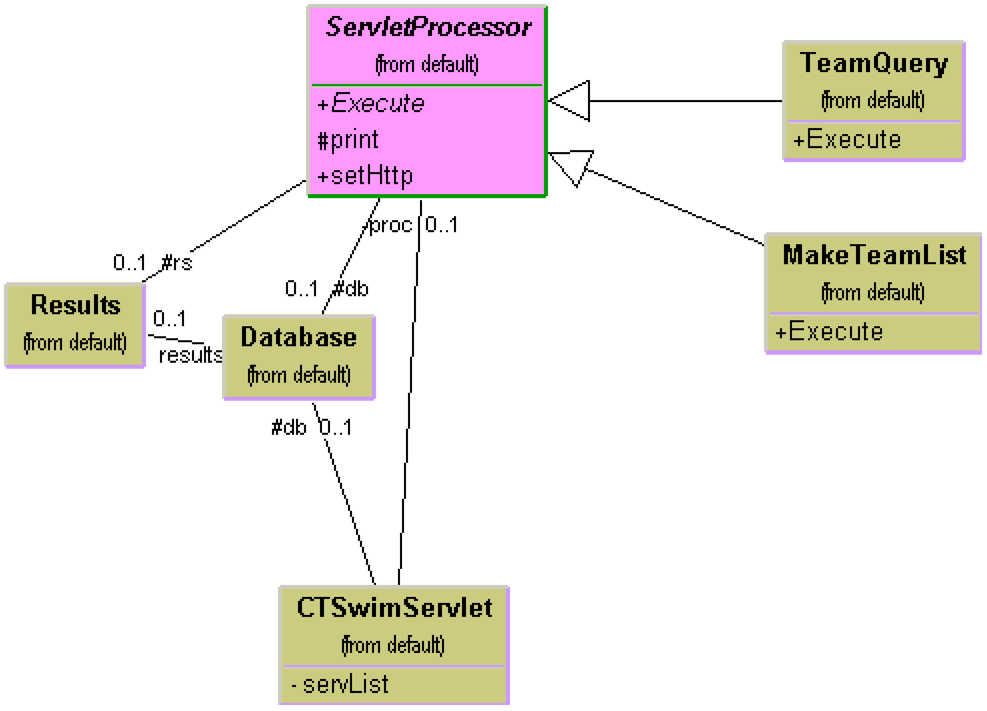
Then for each actual query, you only need to implement the Execute method. For creating the web page containing the list of teams, this Execute method constitutes the entire class. It is just:

```
public void Execute(PrintWriter out) {

        String query =
             "SELECT Clubs.ClubCode, Clubs.ClubName”+
             “ FROM Clubs ORDER BY Clubs.ClubCode;";
        Results rs = db.Execute (query);

        res.setContentType("text/html");
        out.println("<html>");
        out.println("<head><title>Team Query</title></head>");
        out.println("<body>");
//read in template html file and fill in team list
        InputFile fl = new InputFile("CTTQuery.htm");   //get template file
        String s = fl.readLine();
        while (s.indexOf ("<option>")<=0) {
            out.println (s);
            s = fl.readLine();
        }
        while (rs.hasMoreElements ()) {
            out.println ("<option>" + rs.getColumnValue ("ClubCode"));
            //rs.nextElement ();
        }
        s = fl.readLine();
        while (s != null) {
            out.println (s);
            s=fl.readLine();
        }
    }
```

The *Results* and *Database* classes are the ones we developed when discussing using the Façade pattern for handling database manipulations, in the December, 1998 column. The class structure for the entire servlet and its related classes is shown below:



The basic program is the CTSwimServlet. It instantiates one of the subclasses of the abstract ServletProcessor class for each different *servletType* parameter value. The two instances in this simplified example are *TeamQuery* and *MakeTeamList*. One clever way for the program to select the correct class is using a Hashtable where instance of the classes are stored with the servlet type strings used as keys. Using this simple approach our entire main servlet program becomes just:

```
public class CTSwimServlet extends HttpServlet
        implements SingleThreadModel {

      protected Database db;        //database we talk to
      private Hashtable servList;   //list of servlet proc classes
      private ServletProcessor proc = null;

   public void init(ServletConfig svg)
                 throws ServletException{
      super.init(svg);
      db = new Database("sun.jdbc.odbc.JdbcOdbcDriver");
      db.Open ("jdbc:odbc:CTSwim99",null);
      servList = new Hashtable();
//load hash table with candidate classes
```

```
      servList.put ("MakeTeams", new MakeTeamList());
      servList.put ("CTSwimTeams", new TeamQuery());
   }
   //------------------------------------------------
   public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

      String stype = req.getParameter("servletType");
  //selec a class based on the servletType parmeter
      proc = (ServletProcessor)servList.get(stype);
      proc.setHttp (  req, res, db);
      proc.Execute(res.getWriter());
   }
}
```

The nice thing about the simple approach is how easily it scales. As your needs for new queries grow, you just create new subclasses of ServletProcessor and add them to the hash table on startup. Since I began this article, I've add 3 more queries to the site, each requiring only a few minutes programming of a new subclass of ServletProcessor.

Design Patterns we Used this Month

The servletProcessor class is an abstract class that has concrete implementations and qualifies as a simple version of the Template pattern. The instances of the servletProcessor each have an Execute method, but are otherwise identical, and can be consider examples of the Command pattern. The database classes we use to access our data are examples of the Façade pattern. And finally, the hash table encoding trick the helps us select the right instance of the servletProcessor class amounts to a simple Factory pattern.

Multiple User Accesses to Your Servlet

In a real-world system, many users might make requests to your servlet at once, and each will be launched as a separate thread. This means that your servlet application must not assume that the contents of class-level variables are invariant. You can do a lot to handle this if you are careful with the *synchronized* keyword, but for the simple example here, we simply had our base server class implement the *SingleThreadModel* interface. This assures that each user access your servlet serially and will prevent thread confusion. Of course, if you have a high number of users this is not a sufficiently elegant solution. In this case you should make sure that there are few if any class-level variables that might get stomped on and that the rest are accessed only in synchronized processes.

Running Servlets on Your System

The Sun java web site shows 2 Sun servers and 19 third party web servers that support Java servlets. In addition, there are 5 add on servlet engines, at least 3 of which will work with IIS. If you just want to try servlets in a low key low traffic test, you can use the *servletrunner* program that comes with the original 2.0 version of the JSDK or the *startserver* script that comes with JSDK 2.1. These intercept calls to port 8080 (which you can change to other port numbers if you like) and run with any web server. They are less robust than a full servlet engine, but seem to work fine for me most of the time.

Installing servlets on your server varies with the server, but amounts to putting the classes in a particular path and setting some configuration file to contain that path. This is documented clearly for the JSDK 2.0. For JSDK 2.1, if you use the *startserver* script, you have to create 2 levels of new directories.

1. For a servlet called swimInfo, create the directory

```
...jsdk2.1/swimInfo/WEB-INF/servlets
```

and put all the servlet classes in it.

2. Then, add 2 new lines in the default.cfg:

```
server.webapp.timesheet.mapping=/swimInfo

server.webapp.timesheet.docbase=swimInfo
```

3. Run the startserver script;
4. Access the servlet using the URL:

```
localhost:portnumber/swimInfo/servlet/whateverclassname
```

Learning More About Servlets

There is a nice servlet tutorial on the java.sun.com web site. In addition, the examples that come with the JSDK are pretty revealing. There is also a servlet-interest list on the java.sun.com list server where you can pose questions to the active servlet community. In additon, *Java Servlets* by Jason Hunter, from O'Reilly books is well-regarded.