Chapter 11
Upgrading Servlets to Agents

This chapter is designed especially for two groups of people: (1) Those interested in learning more about the relationship of Servlets to Agents; and (2) those of you who have a non-Domino Web server (shame on you!), have Servlets in production, and are interested in learning how to convert those Servlets to Domino Agents. Sites that have installed Lotus Go and are upgrading to Domino, for example, should read on if they have any Servlets that they want to enhance and preserve. I'll show you how you can write an Agent/Servlet Adapter that allows your existing Servlets to be turned into Domino Agents.

What Is a Servlet?

A Servlet is kind of a cross between an Applet and a CGI/bin program. It's like an Applet in that you write it in Java, and it typically inherits from a base class such as javax.servlet.GenericServlet (you can alternatively implement one or more of the standard interfaces in your own class). It's like a CGI/bin program in that it sits around on a server machine waiting to be invoked, usually via some URL syntax. Unlike CGI/bin programs, though, the server doesn't have to start up a new process each time a Servlet is invoked—all Servlets run in the server's Java VM process, and the VM will cache Java classes for a period of time, which is a big performance win.

The Servlet class is not part of the standard Java language. It's considered an extension, which is why the package name is "javax.servlet" instead of "java.servlet". You can download a free copy of the Java Servlet Development Kit from the JavaSoft Web site (http://java.sun.com).

Unlike Applets, Servlets don't have any user interface, because they run on server machines where there's usually no one present to enjoy it anyway.

Why would you use a Servlet? There are lots of things a Web-based application wants to do that are best done on the server machine, as opposed to from the browser. Anything you'd have previously torn your hair out to write as a CGI/bin program or Perl script can be done (more easily, in my opinion) in Java as a Servlet: connections to back-end relational database systems, redirection to other servers, and, of course, Domino applications (although I'll try to convince you later in this chapter that Domino Agents offer advantages over Servlets that you should seriously consider).

Setting Up Domino To Run Servlets

This section might appear out of order in the chapter, but if you want to try out any of the samples on a Domino server, then you'll have to do a couple of things to set it up first. For other kinds of servers refer to the provider's documentation.

Servlet support is by default disabled in Domino 4.6, so you need to do a couple of things. First of all, add the following lines to your server's notes.ini file:

```
DominoEnableJavaServlets=1
JavaUserClasses=c:\notes\data\domino\servlet
```

Of course you would use the correct path name for your machine. I needed to create the servlet subdirectory under the domino directory, it wasn't created by default at install time. The first line tells the HTTP server to initialize the Servlet Manager (you'll see a message about that in the server console when HTTP starts up). The second line adds the directory where you'll place your compiled Servlet code, so that the Servlet Manager can find it. You can use any directory you like, but putting it under \notes\data\domino is convenient.

Next, make sure that HTTP is running in your Domino server. If your notes.ini includes HTTP among the tasks that the server always runs (look for the "Servertasks=" line), then you're all set. If you want to start it up manually, then type "load http" in the server console window.

The next thing you need is a Servlet configuration file. This file tells the Servlet Manager how to map URLs to .class files for running Servlets. Here's one I'm using to run some sample Servlets that Sun has on their Web site:

```
#Servlet configuration
Servlet HelloWorldServlet { }
Servlet SnoopServlet { }
Servlet DateServlet { }
Service HelloWorldServlet /servlet/HelloWorldServlet
Service SnoopServlet /servlet/Snoop
Service DateServlet /servlet/Now
```

The servlet.cnf file goes in your Notes data directory. This one, for example, maps the URL "http://<server>/servlet/now" (note that the names are not case sensitive) to the DateServlet.class file, located in the Domino HTTP server's "/servlet" directory.

Finally, just put some Servlet .class files in the specified directory, bring up a browser, and type a URL to invoke one of the Servlets. The Servlet sends back an HTML response which you see in your browser. The Java Servlet Development Kit which I mentioned above contains the sample Servlets (including source code) referenced in this servlet.cnf sample file.

Figures 11.1 and 11.2 show what I got when I ran the DateServlet and HelloWorldServlet programs from my browser using Domino.

Figure 11.1 *Running the DateServlet sample.*

Figure 11.2 *Running the HelloWorldServlet sample.*

If you also want to be writing your own Servlets, you need to add the location of the classes.zip file installed from the Java Servlet Development Kit to your CLASSPATH variable.

Writing a Servlet

The javax.servlet package is pretty straightforward—there are only a few classes and interfaces that you need to deal with for a simple Servlet:

- **javax.servlet.http.HttpServlet**: This is a basic framework for a Servlet implementation. Implements the javax.servlet.Servlet interface, and adds specialization for handling HTTP requests to the more general javax.servlet.GetnericServlet class. Most of the time you can write a class that extends either HttpServlet or GenericServlet and go from there.
- **javax.servlet.http.HttpServletRequest**: This is an interface that is used to package up all the incoming information that a Servlet needs to do its job, including information about the invoking URL and about the current execution context (meaning essentially, information about the HTTP server that is invoking the Servlet). You can use (or extend) a class that JavaSoft provides that implements this interface called sun.servlet.http, but I prefer to avoid any classes that are not part of the "java" or "javax" packages. As you'll see in the examples coming up, it isn't very hard to implement your own class for this.
- **javax.servlet.http.HttpServletResponse**: This is also an interface, and it is used to collect the Servlet's output and transmit it back to the user's browser session. The typical way of doing that is to write output to the output stream maintained by the HttpServletResponse instance. As you'll see below, you can get an instance of the javax.servlet.ServletOutputStream class by calling HttpServletResponse.getOutputStream(). Again, you can use the sun.servlet.http.HttpResponse class if you like, but I prefer not to.
- **javax.servlet.ServletConfig, javax.servlet.ServletContext**: We'll use these when we write our Servlet-running Agent in the next section of this chapter. For now, you can think of these interfaces as providing a Servlet with information about its initialization parameters (ServletConfig) and about its current execution context (ServletContext). You get a ServletContext instance from a ServletConfig instance, and the Servlet gets a ServletConfig instance passed to it at initialization time.

The Servlet paradigm is pretty simple: You invoke it with a URL, as in the above examples, and you can pass input (or "request") parameters as part of that URL.

Everything following the Servlet's name is parsed into parameter input for the Servlet. For example, the URL

```
http://LocalHost/servlet/SnoopServlet/foo/bar?a=z
```

invokes the Servlet SnoopServlet with a path parameter of "/foo/bar" and a query string of "a=z".

The Servlet gets a special object instance of the class javax.servlet.ServletRequest (or one of its derivatives) containing all the input parameters, if there are any. The Servlet also gets an instance of the javax.servlet.ServletResponse class (or one of its derivatives), which it uses to format its output. The contents of the ServletResponse instance are transmitted back to the browser.

I took one of Sun's standard Servlet examples and modified it a bit in Listing 11.1.

Listing 11.1 Sample Hello World Servlet (HelloWorldServlet.java)

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

public class HelloWorldServlet extends HttpServlet
{
    public void doGet (HttpServletRequest req,
HttpServletResponse resp)
    {
     try {
          resp.setContentType("text/html");

          ServletOutputStream out = resp.getOutputStream();
          out.println("<html>");
          out.println("<head><title>Hello
World</title></head>");
          out.println("<body>");
          out.println("<h1>Hello World, from Looseleaf
Software, Inc.</h1>");
```

```
            out.println("</body></html>");
           }
        catch (Exception e) { e.printStackTrace(); }
        }

        public String getServletInfo()
        {
         return "Servlet to say hi, from Looseleaf Software";
        }
    }  // end class
```

Note how similar this is to a server Agent that formats HTML output and sends it back to the browser, as in the Ex82Browser example in Chapter 8 (except that with Agents we use a java.io.PrintWriter stream and a different call to get it). If you compile this code and install it in your domino\servlet directory, you should be able to invoke it (assuming you've set up the servlet.cnf file as I showed above) from a browser. This is a pretty simple one, as it takes no input arguments and does nothing of much interest.

Note that if you're using Domino to run Servlets and you're in a develop/test/fix cycle, you'll have to shut down and restart the HTTP server task (not the whole Domino server) to get it to reread the Servlet's .class file. You can do that by going to the Domino console window and typing "tell HTTP quit," then after you get the shutdown message, type "load http." This is a bit of a pain, but actually is better for run-time performance: The Servlet Manager caches Servlet .class files in the Java VM for a while, and therefore doesn't have to reload them from disk on every invocation.

Now let's add some NOI calls to this Servlet. Listing 11.2 adds the name of the current Notes id. Observe that I didn't say "effective name," because that is only available with Agents. This call (from the Session class) will return the actual id in force when the program is run.

Listing 11.2 HelloWorld Example Modified For NOI (HelloNOIServlet.java)

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

public class HelloNOIServlet extends HttpServlet
{
    public void doGet (HttpServletRequest req,
HttpServletResponse resp)
    {
     try {
          NotesThread.sinitThread();
          resp.setContentType("text/html");
          Session s = Session.newInstance();

          ServletOutputStream out = resp.getOutputStream();
          out.println("<html>");
          out.println("<head><title>Hello
World</title></head>");
          out.println("<body>");
          out.println("<h1>Hello World, from Looseleaf
Software, Inc.</h1>");
          out.println("Current user name is: " +
s.getUserName());
          out.println("</body></html>");
         }
    catch (Exception e) { e.printStackTrace(); }
    finally { NotesThread.stermThread(); }
    }

    public String getServletInfo()
    {
     return "Servlet to say hi, from Looseleaf Software";
    }
}  // end class
```

Figure 11.3 shows the output of this Servlet (don't forget that you have to add an entry in servlet.cnf and restart the HTTP server for this to work).

Figure 11.3 *Output of the HelloNOI Servlet (HelloNOIServlet.java).*

Here's a real-life case where we really need to use the static NotesThread init/term calls. We can't have the Servlet extend NotesThread, because it has to extend one of the Servlet classes. Note also that even though the Servlet is being run by a Domino task, we still have to explicitly initialize and terminate the thread for Notes if we want to use any of the NOI objects. Note that the id in effect is the server's, so watch out!! We could have created a multithreaded Servlet which spawned a NotesThread instance to do the work and avoided the static NotesThread calls, but this way seemed much simpler.

Now that you know how to set up Servlets in general and also how to make them NOI-aware, you're all set to write that killer Servlet for Domino. Actually, if you don't mind that all Servlets that use NOI run with the server's id, then you can write NOI Servlets for any HTTP server, not just Domino.

Transforming Servlets to Agents: The Servlet/Agent Adapter

Let's say you've been using Servlets and now you're interested in seeing how to better integrate them with Domino. You can, of course, continue to use them as is on a Domino machine, as we've seen above. You can even enhance them to interact with NOI, as I've shown. No big deal there, why would you want to "transform" a Servlet (especially one that already works!) into a Domino Agent? Well, to paraphrase every real estate broker you've ever talked to, "Security, security, security," and "convenience, convenience, convenience."

That's not to say that Servlets enforce no security at all, because you do have to get a server administrator to install the .class or .jar file on the server and update the servlet.cnf file. Depending on the temperament of your server administrator, this can

either be relatively painless or it could be worse than getting a root canal. If you find yourself in the latter category, imagine further what it will be like when you decide, after deploying your Servlet for a while, that you want to enhance it—every month.

The big advantages of Agents over Servlets are:

- Agents are part of the database design in which they reside. You can update them remotely from a Notes Client, and they replicate around with the rest of the database.
- Agents are signed and can be selectively set to run with the privileges of the signer (the default) or (when invoked via the HTTP server) with the privileges of the Web user (and Domino will authenticate the Web user for you as well).
- Agents can be restricted further in their access to system resources (see the discussion of restricted and unrestricted users in Chapter 8). Servlets have godlike access to the systems resources, including network and disk.

There are no real drawbacks to using Agents instead of Servlets (that I could think of, assuming you're using Domino 4.6, of course). All the argument passing features are available to Agents as well, as you'll see.

If you already have a production environment that makes significant use of Servlets, and you'd like to migrate them as painlessly as possible into a Domino environment, you have three choices:

1. You can just install them on the Domino machine as is, set up a servlet.cnf file to tell Domino's HTTP task where they live and what they should be called, and you're done. Of course you don't get any of the advantages of Agents this way.
2. You can recode each Servlet as an Agent. This option isn't too terrible, unless you have lots and lots of working Servlets. You have to make AgentBase the base class, instead of Servlet or HttpServlet, and you have to access any CGI variables through the Document returned by AgentContext.getDocumentContext().

3. I've saved the best for last. You can use the Agent-to-Servlet Adapter that we discuss in the following sections. It's an Agent that can parse URL arguments and run any Servlet whose .class file it can find.

Servlet-Running Agent

The basic approach here is to write a Java Agent that knows how to load and run Servlets (which, after all, are just Java programs), passing through any arguments that it can parse out of the invoking URL. You've already seen in Chapter 8 how easy it is to parse CGI variables out of a URL using the AgentContext.getDocumentContext() call (Ex84CGI.java). Now let's enhance that basic program to pull out a Servlet name, load, and run it.

We'll adopt a convention that the URL should contain a parameter called "DOMINO_SERVLET", and that the value of the parameter is the name of the Servlet to run. We'll also assume that the Notes environment variable "JavaUserClasses" is set up to point to the directory where not only our Servlet class(es) lives, but where any other supporting classes (such as all the classes used in the Agent) reside. This is an important point, because the HTTP server's Java Virtual Machine (VM) doesn't have a class loader that knows how to pull .class files out of a Notes database, as the Agent Manager does. Thus, because our Servlet is using some of our supporting classes directly (such as AgentRequest and AgentResponse, discussed below), they have to be available. If these new classes were part of the lotus.notes package, we wouldn't have to worry about it.

This first version of the Agent will ignore parameters beyond the Servlet name. We'll name the Agent RunServlet and store it in the database Ex11.nsf.

A word of caution: Dipping our toes into this kind of technology is, so far as I can tell, pretty pioneering of us. We're writing a Domino Agent that, in fact, implements part of the functionality of an HTTP server, in that it locates and loads Servlet programs, as well as provides context for them. There is very little documentation on

how to do this correctly, other than the specification on the classes themselves (which only tell you what the API looks like, not how to use it).

To implement this Agent I had to create some support classes that implement some required interfaces (all the source code for these classes is on the CD):

- **AgentCGIEnumerator**. Implements java.util.Enumeration for the CGI items in an Agent context Document. Called from AgentServletConfig and AgentRequest.
- **AgentRequest**. Implements javax.servlet.http.HttpServletRequest. Provides the API so that a Servlet can query the contents of the context Document, which contains the standard CGI variables.
- **AgentResponse**. Implements javax.servlet.http.HttpServletResponse. Most of what this is for is to allow the Servlet to get an output stream to write to.
- **AgentServletConfig**. Implements javax.servlet.ServletConfig. You need one of these in order to load a Servlet. Provides an alternative API to access the CGI variables.
- **AgentServletContext**. Implements javax.servlet.ServletContext. You need one of these in order to implement a ServletConfig. It gives the Servlet access to information about the network context, and a way to log messages.
- **AgentOutputStream**. We needed something that could extend the ServletOutputStream class (the stream that the Servlet uses to write its output; it comes from the response object) to accommodate the Agent's java.io.PrintWriter stream (set up by AgentBase for exactly the same purpose). Unfortunately, although both the PrintWriter and ServletOutputStream classes make use of java.io.OutputStream to do their thing, you can't cast a PrintWriter instance to ServletOutputStream, as they don't have a common ancestor. Writing a class that extends ServletOutputStream allows us to wrapper a PrintWriter instance and redirect calls to it. This helps ensure that Servlets won't have to be recoded when they get run by our Agent.

The main class for this Agent is RunServlet. All the classes are loaded into the RunServlet Agent in Ex11.nsf. The URL we'll use to try it out is:

```
http://localhost/book/Ex11.nsf/RunServlet?OpenAgent&DOMINO_S
ERVLET=HelloNOIServlet2
```

Note that this time we need an explicit "?OpenAgent" action directive, because we want to add some "query arguments" afterwards. If you don't have query arguments and the Agent name is the last thing in the URL, then the ?OpenAgent directive is implied. If you do have arguments following the Agent name, you need an explicit ?OpenAgent directive. You can have as many argument name/value pairs, as long as you start each with an "&" and separate the name from the value with an "=".

We had to modify the HelloNOIServlet a bit from the way it was before, and we'll discuss that after talking about RunServlet. Listing 11.3 shows the code for the RunServlet Agent.

Listing 11.3 Simple Agent/Servlet Adapter (RunServlet.java)

```
import lotus.notes.*;
import java.util.*;
import java.io.*;
import javax.servlet.*;

public class RunServlet extends AgentBase
{
    public void NotesMain()
      {
      try {
          Session s = this.getSession();
          AgentContext ctx = s.getAgentContext();
          Document doc = ctx.getDocumentContext();
          PrintWriter pw = this.getAgentOutput();

          if (doc == null)
              {
              pw.println("No context document");
              return;
              }
```

```
            // create request/response objects
            AgentRequest req = new AgentRequest(doc, s);
            AgentResponse resp = new AgentResponse(doc, s,
pw);

            /* We have some context, look for the servlet
name.
               One of the arguments to the agent is supposed
to
               be &DOMINO_SERVLET=xxx. It appears in the
QUERY_STRING
               CGI parameter, but we need to parse it out. An
easy
               way to do that is to use the request object's
getParameter()
               call, which does the parsing for us */

            String servname =
req.getParameter("DOMINO_SERVLET");
            if (servname == null || servname.length() == 0)
                {
                pw.println("Invalid Servlet name<BR>");
                pw.println("Query string was: " +
req.getQueryString() +
                                          "<BR>");
                return;
                }

            /* Load the servlet into the VM. We assume that
the Classpath
               is set up correctly */

            Servlet servlet = null;
            try {
                Class cls = Class.forName(servname);
                if (cls == null)
                    {
```

```
                    pw.println("Servlet " + servname + " not
found <BR>");
                    return;
                    }

                Object obj = cls.newInstance();
                if (obj == null || !(obj instanceof
javax.servlet.Servlet))
                    {
                    pw.println("Invalid Servlet class " +
servname +
                                                "<BR>");
                    return;
                    }

                servlet = (Servlet)obj;
                }
            catch (Exception e)
                {
                e.printStackTrace();
                pw.println("Servlet " + servname + " not found
<BR>");
                return;
                }

            // we need a ServletConfig instance for the
servlet
            ServletConfig config = new AgentServletConfig(s,
doc);

            // init the servlet
            System.out.println("Initializing Servlet " +
servname);
            servlet.init(config);

            // send the request, get the response
            System.out.println("Invoking Servlet " +
servname);
```

```
            servlet.service(req, resp);

            // all done, destroy it
            System.out.println("Destroying Servlet " +
servname);
            servlet.destroy();

            }  // end try

        catch (Exception e) { e.printStackTrace(); }
        }  // end NotesMain

}  // end class
```

Discussion of the RunServlet Agent

Most of what this Agent does is not rocket science. It's a regular old Agent that looks for the name of a Servlet on the URL with which it was invoked, loads the corresponding Servlet class file, and runs it. Actually, it isn't quite that simple, mostly due to the fact that if we want to support argument passing (and we do), then we have to deal with the fact that all our CGI arguments are in a Notes Document instance, not in some stream managed by the server. The "adapter" implementation I'm presenting here is really a skeleton, it doesn't handle a bunch of the options (POST requests, for example) that you'd want from a production piece of code. Still, you could take what I'm giving you here and enhance the heck out of it pretty easily.

The Agent starts out normally, getting the Session, AgentContext, context Document, and output stream objects. We want to pass the standard PrintWriter stream to the Servlet in the response object, because that's the stream that Domino will serve up to the client's browser as the result of running the Agent.

Next we create the request and response objects. The JavaSoft-supplied classes sun.servlet.http.HttpRequest and sun.servlet.http.HttpResponse, while part of the Servlet Development Kit, are not part of the javax package. Instead they belong to the package sun.servlet.http, which means that not all versions of Java will include them, so

we try to avoid using those classes here. Aside from that, the standard request and response object interfaces (javax.servlet.http.HttpServletResquest and javax.servlet.http.HttpServletResponse) that the Sun classes implement don't give you any way to tell these objects what output stream to use, nor where to get their input parameters, which is another reason for writing our own code.

So, I had to create my own request and response objects (implementing the standard interfaces). I called the new classes *AgentRequest* and *AgentResponse*. The code for these two classes is on the CD, but not here in the text, mainly because they're a bit long and tedious: They take the information they need as arguments to their constructors and implement all the methods to get CGI variables and HTTP headers, and so on. The big difference between these objects and the ones in the kit are that they know how to get the parameters and headers from the Agent's context Document. If you're interested in the details, go ahead and plow through the code on the CD. You'll find that much of the standard functionality is not implemented yet, but there's enough there to get some sample Servlets running.

One of the nice things that the request object does is parse through the CGI QUERY_STRING variable for you and pull out the various argument names and values (if any). So rather than parse through the query string ourselves, we let AgentRequest do it. That gives us the name of the Servlet (case sensitive, as all Java class names are) to run.

The next thing we have to do is get the class for the Servlet loaded into the VM. The Servlet Development Kit contains a sun.servlet.ServletLoader class, but I didn't want to use it because it's not part of the javax package (it belongs to the sun.servlet package). So I just used the static method Class.forName() to load the Servlet's .class file (it has to be on the Classpath). That call returns an instance of the java.lang.Class class, which you can then use to create an actual instance of the class that you just loaded. Note that only Unrestricted Agents are allowed to do this.

Once we have an object instance for the Servlet's class, we can use the *instanceof* operator to verify that it really is an instance of javax.servlet.Servlet. Assuming all is well to this point, we can generate a configuration object (our own AgentServletConfig class) and use that to initialize the Servlet. Then all we need to do is invoke the Servlet.service() method, passing in the request and response objects we generated earlier. The Servlet will use the request object to get all its parameters, and will use the response object to write out its results. The base class implementation of service() (in the javax.servlet.http.HttpServlet class) figures out the type of request we're making, and dispatches the call appropriately. Our example Servlets will use the doGet() method to respond to GET requests.

When we return from the service() call, we clean up the Servlet, and then we're done. If we were writing a real Servlet manager, we probably would invent a Servlet management scheme that allowed us to initialize and destroy Servlet instances only once, rather than every time they were invoked. That exercise is left to the reader.

I mentioned above that I modified the HelloNOIServlet code to take advantage of being run from an Agent. Listing 11.4 shows the revised code, and I've renamed it HelloNOIServlet2 (original, huh?), so that the files on the CD wouldn't conflict.

Listing 11.4 The HelloNOIServlet2 Example (HelloNOIServlet2.java)

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

public class HelloNOIServlet2 extends HttpServlet
{
    public void doget(HttpServletRequest req,
HttpServletResponse resp)
    {
     ServletOutputStream out = null;
```

```
        System.out.println("From Servlet HelloNOIServlet:
doget() invoked");

        try {
            resp.setContentType("text/html");
            out = resp.getOutputStream();
            Session s = ((AgentRequest)req).getAgentSession();

            out.println("<html>");
            out.println("<head><title>Hello
World</title></head>");
            out.println("<body>");
            out.println("<h1>Hello World, from Looseleaf
Software, Inc.</h1>");

            AgentContext ctx = s.getAgentContext();
            out.println("Current user name is: " +
ctx.getEffectiveUserName());
            out.println("</body></html>");
            }
        catch (Exception e) { e.printStackTrace(); }
        }

        public String getServletInfo()
        {
            return "Servlet to say hi, from Looseleaf Software";
        }
}  // end class
```

Discussion of HelloNOIServlet2

The entry point where we locate our logic is still the doGet() method. As I stated above, the base class implementation of the service() method that our Agent invokes sees that our request is an HTTP GET, and calls the Servlet's doGet() method.

There's a debug message that will go to the server console. The next line is identical to the original Servlet, it gets the output stream to use for returning a result to the

browser. The stream that's returned in this case, though, is really an AgentOutputStream (but because it extends ServletOutputStream we can treat it as such with no casting). All calls to "out" will be redirected by AgentOutputStream to the PrintWriter stream belonging to the Agent.

The next thing the Servlet does is cast the input request object (of class HttpServletRequest) as an AgentRequest instance, and use a new method on that class to get the Agent's Session. In the old version of the program we just did Session.newInstance(), which would also work here, but we'd lose all of the Agent's context. Using the correct Session is much better, as you'll see.

The next few lines are identical to the original, except that behind the scenes they're really printing to a PrintWriter instead of to a ServletOutputStream.

Then we get the Session's AgentContext instance, and print out the "effective" user name. This also is different from the original, where we printed out the result of the Session.getUserName() call, which is always the server's id. By getting the effective user name, which is only available to Agents, we can get the name under whose privileges the Agent is running: either the signer or the Web user. The rest is, as they say, history. We left out the NotesThread init/term calls that were in the original, because we know we're being invoked by an Agent, and so we don't need them.

So, as I said before, we could have run the original Servlet code unmodified from the RunServlet Agent. The small changes I did make were to take advantage of additional Domino functionality.

Figure 11.4 Shows you what happens when we run the Agent from the browser with the URL above. In the first run, the Agent is set up to run with the identity of the Web user.

Figure 11.4 *Running a servlet from an agent (Web user).*

Note that the user's name is Anonymous, because the Web user was not authenticated by the HTTP server. Now let's change the Agent's properties to run with the identity of the signer. Figure 11.5 shows the output for that.

Figure 11.5 *Running a servlet from an agent (signer).*

Now the user is yours truly, the signer of the Agent. Is this cool, or what? I didn't have to modify a single line of code to make that change, just bring up the Agent Properties box for the RunServlet Agent, uncheck the "run as Web user" box, and save it. The Agent's and Servlet's debug messages (System.out.println) all go to the server console and to the server log as is usual for an Agent.

Processing Optional Arguments

Our HelloNOI Servlet examples didn't use any input arguments at all. The RunServlet Agent looks for a single argument in the QUERY_STRING parameter, "DOMINO_SERVLET=", to get the name of the Servlet to run. There's no reason, however, that we couldn't add more parameters to the query string (delimited by "&"), and have the Servlet look for them in the standard way.

For example, we could take our original URL above, and add another parameter for the HelloNOIServlet program to parse:

```
http://localhost/book/Ex11.nsf/RunServlet?OpenAgent&DOMINO_S
ERVLET=HelloNOIServlet2&optional=howdy
```

In the Servlet's doGet() method, all we have to do is invoke the request's getParameter() call, passing the parameter name in ("optional") in this case. The String value of the parameter is returned, if it is found in the query string. The parameters can come in any order, and even the DOMINO_SERVLET parameter doesn't have to be first.

Summary

So, to wrap up, what can we say about Servlets and Agents? I think the following points are key:

1. Agents offer advantages over Servlets in terms of security, control over use of system resources, and deployment.
2. If you don't want to use Agents, you can still make use of Domino's Java NOI by using the Notes classes from your existing Servlets.
3. If you're upgrading to Domino from a server environment where you have lots of Servlets already, I've shown you a way to seamlessly integrate your existing Servlets into the Domino framework, without recoding the Servlets. Servlets that are run this way inherit all the security and context features of Agents, which (I claim) makes them much more interesting.

Coming up next, Chapter 12 covers NOI and Java Beans. Are they really good for your heart? Stay tuned.