



You are here: [Home](#) > [Developers](#) > [View Source Articles](#) > Server-Side JavaScript View Source Article

Developer Program
[Membership](#)
[One-to-One Support](#)
[Newsgroups](#)

Developer Publications
[View Source](#)
[Developer News](#)

Documentation
[Technical Manuals](#)
[White Papers](#)
[TechNotes](#)
[Sample Code](#)
[FAQs](#)
[Books](#)

Technologies
[CORBA](#)
[Directory & LDAP](#)
[Dynamic HTML](#)
[Java](#)
[JavaScript](#)
[Linux](#)
[Security](#)
[SSJS](#)
[XML](#)

Developer Downloads
[Tools & SDKs](#)
[Patches](#)

iPlanet Products
[Technical Resources](#)

Expanding Your Server-Side Programming Repertoire

By [Duane K. Fields](#)

[Send comments and questions](#) about this article to [View Source](#).

Don't get me wrong -- I love developing in server-side JavaScript. I'm lazy by nature and appreciate JavaScript's relaxed approach to variable typing and declaration. LiveConnect gives me access to Java's unique functionality when I need it, and I've created a good number of reusable libraries that speed the development process. I develop many of our internal tools here at Tivoli Systems with Enterprise Server's JavaScript capabilities and use it for rapid application development. But for our more complex, multitier applications developed using a team approach, I've begun employing Enterprise Server's Java capabilities -- through servlets.

In this article -- written primarily for those currently developing on the server in JavaScript -- I'll explain the basics of working with Java on the server through Sun's Java Servlet API. I'll start off with a brief overview of servlets and their role in server-side software development. Then I'll compare JavaScript applications with servlets in a number of crucial areas of program design. I'm assuming that you're already familiar with Java programming, but if not, I hope this article will encourage you to begin exploring the possibilities offered by Java servlets. Once you're comfortable with basic Java programming, it's fairly easy to extend that knowledge to include servlet development. Additionally, if you have experience developing in server-side JavaScript, you're already familiar with all of the application design concepts necessary to work with servlets.

SERVLETS AND THEIR PLACE IN THE WORLD

Servlets are the first standard extension to Java released by Sun Microsystems. The classes that comprise the Servlet API are typically not distributed with the core Java classes but are controlled and managed by Sun itself. What this means to web developers is that support for servlets isn't included in every web server out there, but the API is stable, easily obtainable, and consistent with the rest of the Java API. And almost every Java-enabled web server can support servlets with just a little retrofitting, described in the next section. Remember, too, that since all of our Java servlet code will be running on the server, not the client, no particular requirements for Java are placed on the end user's browser.

The benefits of developing applications in Java or JavaScript instead of with the more traditional CGI scripting method have long been touted. CGI, by comparison, is a slow and inefficient architecture for web applications, requiring as it does that a unique instance of the executable or script interpreter be invoked to handle each incoming request. Persistence and state maintenance are other problems not easily handled by CGI applications. Both Java and JavaScript applications provide good data-persistence options and much-improved performance.

If Java and JavaScript are both attractive development environments, why should JavaScript developers consider servlets for more complex application projects? First and foremost, Java is a richer, more structured language better suited to complicated programming tasks -- especially those being handled by a team of developers. Java's strong typing and memory-access protection create the safer, more predictable runtime environment demanded by mission-critical applications. Working in Java also gives you access to Java's rich core APIs, as well as the ability to easily build on the efforts of others by leveraging their classes. Last, Java servlets are portable -- not just across platforms, but across vendors.

EQUIPPING YOURSELF FOR SERVLET DEVELOPMENT

As mentioned earlier, while very few web servers (Sun's Java Web Server being a notable exception) support the latest edition of the Servlet API out of the box, it's relatively easy and cheap to retrofit a web server with a servlet engine. A servlet engine is a server extension, written in a server-specific API such as NSAPI or ISAPI, that acts as a harness for running servlets on the web server. While Netscape Enterprise Server 3.5.1 does provide support for the 1.0 release of the Java Servlet API, the 2.0 API now available adds some essential new features, so your best bet is to upgrade to the 2.0 level of support with a servlet engine.

Servlet engines are available for Enterprise Server on most platforms, and for many other web servers as well. Each has its own relatively straightforward interface for configuring, starting, and stopping servlets. The most popular of the engines currently available are the following:

- [LiveSoftware's JRun](#)
- [IBM's WebSphere](#)
- [New Atlanta's ServletExec](#)

This issue of *View Source* contains an article by Paul Colton, CEO of Live Software, on adding the JRun Servlet Engine to a Netscape server. ([Adding Java Servlet API Capability to Netscape Servers](#))

To develop servlets, you'll also need a copy of Sun's [Java Servlet Development Kit](#) (JSDK), version 2.0 or later. The JSDK contains the Java classes necessary to develop servlets, as well as ServletRunner, a standalone application that enables you to test your servlet code without a web server. (Note: The ServletRunner documentation isn't too clear on a couple of things. First, when you

specify the location of your servlet directory with the `-d` option, you need to specify the *full* path. Second, if you need to specify an alternate document root, use the `-r` option.)

DESIGNING YOUR APPLICATION

In this article I'll discuss working with servlets in their most popular form, which is similar to working with a CGI script: you write the servlet, put it on the server, and let it handle requests. However, more advanced options modeled after features that were until recently unique to Sun's Java Web Server are becoming available. One such feature, Server Side Include, enables you to embed references to servlets in a web page in much the same way you would embed an applet. At delivery time, this embedded statement is replaced with the output of the corresponding servlet. Another option, Java Server Pages, even more closely follows the JavaScript application model by allowing you to distribute Java code throughout HTML documents. Since these features aren't as commonly available and build on the basic concepts anyway, I won't go into any detail on them here.

Once you're comfortable with basic Java programming, switching gears from JavaScript development to servlets isn't as hard as you might think. The development techniques involved share many aspects, although beginning Java programmers might have some difficulty getting their minds around the idea of one instance of the servlet object being called on by multiple, simultaneous threads. Like JavaScript, servlets are built around a request/response model. When a page request comes into the web server, it's routed to the servlet for handling, and the servlet's output is returned.

To get an idea of what a Java servlet looks like, take a look at the ever-popular "Hello World!" program in Listing 1. This example shows a pretty typical servlet implementation that extends the `HttpServlet` class and overloads the `doGet()` and `doPost()` methods. For an example of an even simpler servlet implementation, see [Listing 1 in Paul Colton's article](#). In that example, Paul has overloaded the `service()` method (whose job is normally to dispatch the request to either the `doGet()` or the `doPost()` method) to handle outputting the "Hello World" statement.

The sample code included here is provided for your use on an "AS IS" basis, under the [Netscape License Agreement - Terms of Use](#).

Listing 1. A simple "HelloWorld!" servlet

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

public class HelloServlet extends HttpServlet
{
    public void doGet (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        // Prepare the client to receive HTML and get our output stream.
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.println("<HTML><BODY>Hello World!</BODY></HTML>");
    }

    public void doPost (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        // Dispatch this to the doGet() method to make post and get synonymous.
        doGet(request, response);
    }
}
```

As you can see, the basic model for a servlet is to provide methods to handle the `doGet()` or `doPost()` methods, using the `request` object to gather information and the `response` object to return it. Once compiled, the resulting `HelloServlet.class` file can be configured through your servlet engine's configuration panel. Any request (either GET or POST methods) will return the earth shatteringly thought-provoking "Hello World!" message.

Don't worry about the details just yet; we'll delve into these later on. Let's first step back and look at how a Java application is put together.

In JavaScript, you start off by creating a set of HTML documents. Some of these documents are plain HTML, while others contain embedded JavaScript designated for use on the server. You generally have some data-entry documents that collect data through a form, and some form-handling code to interpret and work with the data. These are all compiled into a single `.web` package and placed on the server, mapped to the URL specified in JavaScript's application manager. For example, say we have a two-document application that we call `dbtest`. Its two documents, `query.html` and `show_results.html`, would be compiled into a single package called `dbtest.web`. We would then use the application manager to set up our `dbtest` application. To query the database, the user would visit `/dbtest/query.html`, which when submitted would post the data from its input form to the `/dbtest/show_results.html` URL, to be handled by the code inside of the `show_results.html` document.

Thus, JavaScript has a structured system for combining multiple documents (some containing code) into a discreet application file. By contrast, there's no standardized system for compiling a collection of servlets into a single application. Servlets are generally independent of each other, and a collection of servlets is an application only in the mind of the developer and the spirit of the design. For convenience, related servlets can be combined into a JAR file -- a Java Archive -- for easy shuffling between servers, but as far as the server is concerned, this is just a packaging ploy.

If our `dbtest` application were developed with servlets, we would probably make the query page a plain HTML page, call it `query.html`, and design a Java servlet called something like `ResultsServlet`. The `ResultsServlet` class file would be

`query.html`, and design a Java servlet called something like `ResultsServlet`. The `ResultsServlet` class file would be installed on the server through our servlet engine's configuration interface and mapped to any valid URL we wanted, typically something like `/servlet/ResultsServlet`. (Notice that unlike in JavaScript, we're usually not passing control to an `.html` extensioned file but to the servlet class itself.) The `query.html` input form file would be placed on the web server (or *any* web server, for that matter) as a regular document. While there's no particular benefit in doing so, we could have used a servlet to generate the HTML form that made up our query. Or we could define the `doGet()` method to return the HTML form, and the `doPost()` method to handle the results. This has been a common tactic in the CGI programming world for some time.

If you're familiar with developing CGI applications, it's helpful to look at servlets from that design perspective. You can think of servlets as Java CGI scripts, but with an important distinction. A servlet is instantiated and initialized only once, at startup, and then handles requests from the main web server as needed. Unlike CGI scripts or programs, which are evoked separately to handle each incoming request, a single instance of the servlet handles multiple requests simultaneously. The servlet isn't brought up and torn down with each request. This application persistence happens when JavaScript code is executing -- the server dispatches a separate thread to process the code. We'll look at how to avoid conflicts that may arise between competing threads later.

CONFIGURING YOUR APPLICATION'S STARTUP PARAMETERS

In JavaScript, one of the challenges programmers face is managing host- or application-specific initialization parameters. For example, if you have one application that's used on multiple hosts, its data files might live on different paths on each. Or perhaps you have configuration parameters that need to be changed depending on the exact behavior desired. Other than recompiling the application each time, JavaScript can accomplish this by creating routines to manage persistent INI files on the server, or retrieving configuration data from a database. Unfortunately, this means more work for you, the developer, since none of these techniques are built into the core language.

Servlets, on the other hand, do have a built-in mechanism for passing in initialization parameters. Although it's implemented a little differently by each servlet engine, the basic idea is the same across all of them. A collection of name/value pairs is associated with each servlet on the server. When the servlets are first initialized (say at a reboot), each servlet's `init()` method is called. Within the `init()` method, you can use the `getInitParameter()` method to access these configuration parameters and can alter the behavior of your application accordingly. Likewise, Java provides a built-in class for handling INI-style configuration files -- `java.util.Properties`.

COMMUNICATING BETWEEN CLIENT AND SERVER

One of the critical elements of application design is passing information from the user (on the client) to the server for processing. Let's first review how you pass this information in JavaScript. Form data from a GET or POST action surfaces automatically as properties of the `request` object on the receiving JavaScript page. Variables that may have multiple values -- such as a select list -- are accessed through the `getOptionValue()` function, which takes an identifier string and an index variable. To access information from a named form element, we simply read the corresponding property of the `request` object, which returns a string representing the property's value (or a `null` if it's undefined).

Java also employs the concept of a `request` object that's passed into the receiving code. In Java, an object that's an instance of `HttpServletRequest` is passed into the servlet's `doGet()` or `doPost()` service method, depending on the request method used. This object encapsulates any incoming form data as well as information about the request itself. In typical Java style, form data isn't accessed directly but through the `getParameter()` method. A slew of other methods like `getRemoteHost()` and `getRemoteUser()` are available to obtain other specifics about the request. For variables that may have multiple values, use the `getParameterValues()` method, which returns an array of string values. All in all, it's a very convenient and clean way to get to the incoming data.

Let's look, in Listing 2, at a snippet of JavaScript code that retrieves some form data stored in the `request` object, and at its Java equivalent. Let's assume that the form we're accessing was configured with the following HTML:

```
<INPUT TYPE="text" NAME="username">
<INPUT TYPE="text" NAME="userid">
```

Listing 2. A comparison of form data retrieval code

JavaScript version:

```
username = request.username;
if (request.userid)
    userid = parseInt(request.userid);
else
    userid = 0;
```

Java version:

```
String username = request.getParameter("username");

int userid;

try
{
    userid = Integer.parseInt(request.getParameter("userid"));
}

catch (NumberFormatException e)
{
```

```

    userid = 0;
}

```

Notice that in both languages the values come through as strings, and integers (or other nonstring data types) must be parsed out of the parameters. We'll see some examples of this a little later. (Yes, catching exceptions is a pain, but it's for our own good.)

GETTING ENVIRONMENTAL INFORMATION

Information about the environment a request was delivered in -- such as the IP, hostname, and other request- or server-specific values -- is necessary for lots of common tasks in web development. JavaScript uses its `server` and `request` objects, along with the `ssjs_getCGIVariable()` function, to expose such information. In Java, this information is provided through access methods of the `request` object passed to your service methods. A few examples of common attributes are given in the following table, along with the JavaScript properties/functions and Java methods used to retrieve them.

Property	JavaScript property/function	Java method
Hostname of server	<code>server.hostname</code>	<code>getServerName()</code>
Port number of server	<code>server.port</code>	<code>getServerPort()</code>
IP of client	<code>request.ip</code>	<code>getRemoteAddr()</code>
Remote username	<code>request.auth_user</code>	<code>getRemoteUser()</code>
Query string	<code>ssjs_getCGIVariable("QUERY_STRING")</code>	<code>getQueryString()</code>
Request type (get/post)	<code>ssjs_getCGIVariable("REQUEST_METHOD")</code>	<code>getMethod()</code>

Table 1. Environmental

Access Methods

As you can see, everything you're used to working with is still there, but in Java it's been corralled into a more consistent and manageable interface. In JavaScript, the `server` object is also used to store global data -- information accessible by all applications running on the server. We'll look at how Java handles global data later in this article.

MAINTAINING STATE ACROSS REQUESTS

Now let's see how we can keep some of our information around between requests -- persistent data. Both JavaScript and Java applications are built around a series of stateless requests -- that's the nature of HTTP -- but it's critical that we somehow maintain client-specific data from request to request. JavaScript accomplishes this through its `client` object, which it uses to store data for a particular user and application across requests. You can keep information, such as a shopping basket, in a `client` object that's unique to each user. This data is maintained with cookies, either on the client or the server, depending on the options you've selected. JavaScript takes care of sharing the data and moving the cookies around.

JavaScript's `client` object can store only string values. Any other values, such as numbers, dates, or Booleans, must be parsed out of the resulting string. More advanced techniques can be employed to get around the string-only limitation: you can access the `ssjs_getClientID()` method and use it as a user-specific key to data stored on the `project` object. However, you the developer are responsible for creating, managing, and destroying this data on the `project` object, making this a somewhat daunting task.

The 2.0 release of the Java Servlet API provides for session-specific data management through a technique similar in design to the `ssjs_getClientID()` indexing idea in JavaScript. A client-specific identifier serves as a key to accessing a pool of stored data. Fortunately, Java does most of the cleanup and data shuffling for you. To work with session-specific data, you need to associate the servlet with the session identifier early in its code, before writing any output or working with the `response` object. To establish a session, call the `getSession()` method of the `request` object. Once you have a reference to the user's session object, you can use this object's `getValue()` and `putValue()` methods to retrieve and store Java objects. Any type of object, not just strings, can be stored in a session object. Be warned, however, that you may need to cast the data as you get it back out to let the compiler know what you're intending. Another hint: you'll need to use wrapper classes like `Integer` or `Boolean` to store data primitives since `putValue()` accepts only objects!

Listing 3 is an example of a simple servlet that remembers your last visit by storing a `Date` object in your session object.

Listing 3. A guest log servlet

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;

public class GuestLogServlet extends HttpServlet
{
    public void doGet (HttpServletRequest request,
        HttpServletResponse response) throws ServletException.

```

```

IOException
{
    // Retrieve the session object or create a new one if it doesn't exist.
    HttpSession client = request.getSession(true);

    // Prepare the client to receive HTML and get our output stream.
    PrintWriter out = response.getWriter();
    response.setContentType("text/html");
    out.println("<HTML><BODY>");

    // In case this is a new visitor...
    if (client.isNew())
        out.println("Welcome first-time visitor!");

    else
    {
        Date lastVisit = (Date) client.getValue("GuestLog.lastVisit");
        out.println("Your last visit was on " + lastVisit);
    }

    out.println("</BODY></HTML>");

    // Now store the current date and time for next time.
    client.putValue("GuestLog.lastVisit", new Date());
}
}

```

While the hash key used by the session-tracking object's `getValue()` and `putValue()` methods can be any string, we must remember that unlike in JavaScript, the client-specific session object is shared by all servlets running on the server and isn't partitioned by application. Therefore, we must be sure to avoid collisions between servlets by establishing a consistent naming scheme. A useful technique is to append the servlet or application name to the front of the key to assure a unique entry.

One important difference between Java's session-tracking capabilities and JavaScript's `client` object is how the information is stored between requests. JavaScript can be configured to store its information in client-side cookies -- which are stored on the end user's machine and passed back and forth between requests. As covered in detail in [Writing Server-Side JavaScript Applications](#), client-side cookies have storage and speed limitations but maintain the information across servers and through server reboots.

By contrast, Java's session-tracking technique stores only the client's session ID information through client cookies. The actual data itself is maintained in memory on the server. Java provides a mechanism to use cookies for the data as well, though a little less transparently than through JavaScript. You must instantiate a `Cookie` object and set its attributes directly. Another option is URL rewriting, which passes the information around in the URL itself, the only option available if your application must support users who can't or won't use cookies in their browser. Fortunately, Java provides methods that determine the cookie-storing capabilities (and preferences) of the client, allowing a single application to support cookies where available and URL rewriting where necessary.

The lifetime of the session object is configured by the web server's servlet engine and typically defaults to something short -- 30 minutes or so, since it's primarily intended for "life on transaction" style values. If the servlet or server is restarted, the session is cleared and your data will be lost. If you need persistence over a longer period of time, or beyond the life of the servlet, you'll need to employ more advanced options such as storing the client's information in a database or server-side file. One way to do this is to override the servlet's `destroy()` method, which gets called before the servlet is shut down (unless your server crashes unexpectedly). This is a good place to clean up any open resources or temporary files, and to save off any state information. This information can then be restored during the `init()` phase of the servlet as it starts up.

SHARING DATA GLOBALLY

JavaScript provides a `project` object as a mechanism to hold global data and share information among clients accessing an application. You simply assign properties to `project` and access them later. The persistent nature of Java gives you a natural way to share information among clients accessing a particular servlet -- the class's instance variables. Since servlet classes are instantiated only once, you can store data of interest to all clients accessing the servlet in the instance variables of the servlet itself. You'll need to protect writable instance variables from collision, as they're accessed simultaneously from multiple threads, as described below. Let's look, in Listing 4, at an example of a servlet that can track the number of times it's been accessed. The counter is maintained in an instance variable, accessible by all threads that call on the servlet.

Listing 4. A counter servlet

```

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

public class CounterServlet extends HttpServlet
{
    int counter;
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
    }
}

```

```

        counter = 0;
    }

    public void doGet (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><BODY>");
        synchronized(this)
        {
            counter++;
            out.println("Page has been accessed " + counter + " times.");
        }
        out.println("</BODY></HTML>");
    }
}

```

Because servlets are generally independent of each other, this technique won't work when the data needs to be accessed from more than one particular servlet instance. Suppose you need to have an instance variable that's accessible by threads accessing any servlets on the server. In JavaScript you might use the `project` object as a storage bin for such a variable, since all files that make up the JavaScript application have access to it. In Java, a good strategy is to employ a programming technique called the singleton. In this model, you create a class that can only be instantiated once per virtual machine. The instance itself is accessible through static methods of the class, giving any servlet access to the singleton object. Let's look at the counter example again, this time implemented, in Listing 5, through a singleton class so that any servlet can access it. This makes it a counter that can report its access count to any servlets on the server.

Listing 5. A singleton-style counter class

```

public class CounterSingleton
{
    private static CounterSingleton theInstance = null;

    private static int counter;

    public static CounterSingleton getInstance()
    {
        if (theInstance == null)
            theInstance = new CounterSingleton();
        return theInstance;
    }

    private CounterSingleton()
    {
        counter = 1;
    }

    public synchronized void incValue()
    {
        counter++;
    }

    public synchronized int getValue()
    {
        return counter;
    }
}

```

Then, from within a servlet access method you could work with the current counter value as shown in the following snippet of code:

```

CounterSingleton accessCount = CounterSingleton.getInstance();
out.println("I have been accessed " + accessCount.getValue() + " times.");
accessCount.incValue();

```

The singleton model assures that only one actual counter object is ever instantiated; thus, everyone using the class is accessing the same instance. This strategy often comes into play during the design of applications that connect to databases. In JavaScript, we might instantiate a `dbPool` object and store a reference to it in the `project` object. In Java, however, we would access our pool of database connections through a singleton-style class.

PROTECTING DATA INTEGRITY AT RUNTIME

JavaScript's locking feature is used to ensure that shared data on the `project` or `server` object isn't corrupted by simultaneous modifications from competing threads. In Java, each request that comes in for the servlet accesses the servlet object's methods and instance variables in its own thread. Servlets, therefore, must be written to handle multiple service requests simultaneously, making it the servlet writer's responsibility to control access to any shared resources through Java's synchronization techniques. Such resources include in-memory data such as instance or class variables of the servlet, as well as external components such as files, databases, and network connections.

In JavaScript, if there were a variable stored in the `project` object -- say an access-tracking variable referenced by `project.counter` -- that we wanted to increment, we would protect the code with a lock on the `project` object, make our changes, and then remove the lock. Here's an example:

```
project.lock();
project.counter++;
project.unlock();
```

In Java, locking is handled through Java's thread synchronization methods. Routines or blocks of code should be prefaced with the `synchronized` keyword if they're attempting to access or modify data that could potentially be in use by another, simultaneous thread.

ACCESSING A DATABASE

One of the big benefits of server-side JavaScript is its built-in support for communicating with a wide variety of databases. Rest assured that Java provides equally powerful (if not more so) database support through its JDBC (Java Database Connectivity) package in combination with platform-specific drivers provided by each database manufacturer. Java also provides support for ODBC (Open Database Connectivity), a cross-vendor abstraction layer, allowing you to create truly platform- and vendor-independent database applications.

JDBC works very similarly to JavaScript's LiveWire interface. Once you've created a valid connection object, you can execute queries and access cursors of information through it. As in form-data passing, data is retrieved through access methods rather than as direct properties of a cursor object as in JavaScript. Here's a quick snippet of Java code, just so you can get the idea:

```
Connection con = DriverManager.getConnection("jdbc:odbc:mydb.db");
Statement stmt = con.createStatement();
ResultSet rs = executeQuery("SELECT name, id FROM users");

while (rs.next())
{
    String name = rs.getString("name");
    Int id = rs.getInt("id");
}

stmt.close();
con.close();
```

While I won't attempt to cover the details of JDBC in this article, you may be wondering about that `getConnection()` method. Notice that Java has mapped database connection calls into a URL-style syntax. This gives you a vendor-independent way to address a database as a network resource.

One particularly cool thing about JDBC is that the interface to the underlying data is the same no matter what type of database you're actually working with. And since the interface is extensible, any manufacturer can create drivers to tie into JDBC. For example, there's a JDBC driver that stores all of your information in simple flat files but gives you RDBMS-style access to the information!

PARTING WORDS

Java's rich APIs and good formal structure make it an ideal language for the team development of complex web applications. Looking ahead, it's likely that servlets (or their offspring) will become essential components in the development of web-based applications, as well as one of the key methods of extending functionality in web application servers and other multitier architectures. That's reason enough for web developers to pick up this important technology.

FURTHER RESOURCES

- [Java and class files](#) for each of the example servlets shown in this article
- Paul Colton, [Adding Java Servlet API Capability to Netscape Servers](#), *View Source*
- [Java Servlet Development Kit](#), Sun Microsystems
- Cynthia Bloch, [Servlets](#), Sun Microsystems tutorial
- Jason Hunter and William Crawford, [Java Servlet Programming](#)

View Source wants your feedback!
[Write to us](#) and let us know
what you think of this article.

[Duane K. Fields](#) is a web applications engineer with IBM's Tivoli Systems, designing interactive tools for internal and external customers. Duane is an active writer and a frequent speaker on the technical and aesthetic aspects of developing and running commercial web sites.

(10.98)

Related Readings:

- [Frequently Asked Questions](#): Find the answers to your questions on Internet technologies.
- [Documentation](#): A library of technical information.
- [Developer Central](#): Collections of developer resources organized by Internet technology areas.

Any sample code included above is provided for your use on an "AS IS" basis, under the [Netscape License Agreement - Terms of Use](#)
