CHAPTER 36

Server-side JavaScript

In This Chapter

Where server-side JavaScript scripts go in documents

The server-side object model

Accessing databases through server-side JavaScript

Most of this book is devoted to client-side JavaScript, the scripts that execute when their documents are loaded in a browser. But the JavaScript language originally came into existence (under its old name, LiveScript) as a server-side scripting language. Both client-side and server-side JavaScript share a common core language encompassing such items as how to use variables, control structures, data types, and the rest. Where the two scripting environments diverge is in their object models. Client-side JavaScript is concerned with objects that appear in documents; server-side JavaScript is concerned with objects that govern the two-way connection between the user and server applications.

The subject of server-side JavaScript is large enough to fill an entire book. My goal in this chapter is to introduce the concepts and objects of server-side JavaScript to those who are familiar with the client side of things. If you move on to create server-side applications with JavaScript, this chapter should help you reorient yourself to the server way of thinking about applications.

The flavor of server-side JavaScript covered in this chapter is that found in Netscape's SuiteSpot 3 server family. In particular, the Enterprise Server 3 component is where most of the server-side JavaScript action occurs. This is where you can write scripts that act as go-betweens for the user and large server databases.

Adding Server Processing to Documents

A standard HTML page contains tags that instruct the browser about how to render the content of the page. As a page loads, the browser interprets the tags and lays out the words, form elements, and images from the top of the document to the bottom. If the page contains a form, you may want to capture the information entered or selected by the user.

With Navigator you can certainly submit that form and its contents as an e-mail message without any further help from the server, but many applications want to link the form entries with a database (either to save the data to the database or retrieve some data based on a search entry). Before server-side JavaScript, that interactivity with the server was handled exclusively by a Common Gateway Interface (CGI) program on the server commonly written in the Perl language. The `ACTION` attribute of the form invoked the CGI program by name (as part of the URL). The CGI program would churn on the submitted data and output a full HTML page that comes back in response.

Server-side JavaScript (SSJS) works a little differently. Rather than existing as a separate program on the server, a SSJS script typically resides in an HTML document stored on the server embedded in a compiled program file. A form's `ACTION` attribute is set to the URL of that HTML file containing the server-side scripts. In response to the submission, the server loads the compiled program while temporarily storing in memory information from the form being submitted. Any HTML content of the associated HTML file is passed on to the client. Some server-side scripts in the document also probably dynamically compose some or all of the content being sent out to the client.

Embedded server scripts

An HTML file on the server contains not only the HTML content, but perhaps some client-side scripts and server-side scripts. Tags distinguish the two types of scripts. Client-side scripts are contained by `<SCRIPT>...</SCRIPT>` tags (and run only when the document is in the client machine), while server-side script statements are contained by `<SERVER>...</SERVER>` tag sets. As Enterprise Server starts to respond to a submission from a browser, it runs whatever scripts appear inside these tags. None of the server script statements in these tags or the tags themselves appear in the source code of the document sent to the browser.

Scripts inside the `<SERVER>` tags work with the same core JavaScript language as you script for the client. But the server doesn't know about document objects. Instead, it knows how to work with server-side objects, such as the object that contains the contents of form elements that arrived with the last submission.

Because server-side scripts can also create content that goes in the HTML page being downloaded to the client, the HTML page can contain basic template content in plain HTML. Custom content created as a result of server processing can be written interspersed in the middle of the page. It's like when you have client-side scripts use `document.write()` to create a portion of a document as it loads. In the case of the server, it has a `write()` method that outputs content to the data stream that heads for the client.

At the client end, the only source code it receives is the HTML that defines the page (and perhaps client-side scripts, if that's what the design calls for). Therefore, it is possible to place browser detection on the server (it has access to userAgent data), and let the server dish up the HTML and scripting appropriate for the browser connecting at that instant.

To demonstrate this process, Listing 36-1 shows the server-side HTML file for a simplistic application that does nothing more than display the server version in a document. Then, in Listing 36-2, I show what the source view looks like on the client. (Neither of these listings is on the CD-ROM.)

Listing 36-1: Server-side HTML File

```
<HTML>
<HEAD>
<TITLE>Simple Server App</TITLE>
</HEAD>
<BODY>
<B>This application is running on the following Netscape Enterprise
Server version and platform:</B><BR>
<SERVER>
        write(server.httpdjsVersion + ".")
</SERVER>
</BODY>
</HTML>
```

Listing 36-2: Client-side Source

```
<HTML>
<HEAD>
<TITLE>Simple Server App</TITLE>
</HEAD>
<BODY>
<B>This application is running on the following Netscape Enterprise
Server version and platform:</B><BR>
3.0 Windows NT.
</BODY>
</HTML>
```

Server-side libraries

The process of creating an Enterprise Server application that uses server-side JavaScript includes a compilation step. Enterprise Server 3 comes with a tool called jsac.exe, which is a JavaScript application compiler. The output of the compiler is a server file (with a .web extension) that is the application as far as the server is concerned (some other application management tools are also provided to install each application into the server). That .web file contains all the HTML files to be used in the application.

In addition to HTML files, which encapsulate their server-side scripting within each document's <SERVER> tag sets, you can add library files of scripts that have global scope among all the HTML files that are part of the application. Not surprisingly (if you know about client-side library files), the server-side libraries are script-only documents whose file names have the .js extension. All HTML and .js files are compiled in the .web application file.

(If the HTML files that are downloaded to the client are to use client-side .js libraries, those .js files are not compiled in the .web application. They simply

reside in the same directory as the HTML files that get compiled in the .web application. When Enterprise Server receives a request for one of these client-side source files, it knows where to find the file and sends the library down to the client.)

Essential Server-side Objects

Although the full complement of SSJS objects is beyond the scope of this book, I want to introduce you to the key objects that the server knows how to work with. If for no other reason, this exposure can give you an appreciation of how the core JavaScript language can be applied to different object models (for example, server-side objects versus client-side document objects).

The discussion here focuses on four server-side objects: server, project, client, and request. This order goes from the most "global" in scope to the narrowest. For the most part, all four objects are stored in the server memory (except for some ways of specifying the client object) and don't represent any physical entity. Not every application addresses each object directly, but the objects are almost always present when an application is running.

The server object

Whenever the Enterprise Server software starts up, it automatically generates a server object in the server's memory. The object goes away when the server is stopped (the server software can be started and stopped independently of the server hardware).

Information that the server object stores includes items such as the host, hostname, port, protocol, and server version. All applications running on the server share this information. An application can also create additional custom properties appropriate to the server-level — any information that needs to be shared among all applications running on the server. The syntax for reading or adding a server object property is similar to other JavaScript object properties reading and writing:

```
var hostName = server.hostname
server.adminEMail = "serverdude@giantco.com"
```

Server-side JavaScript offers provisions for maintaining the integrity of changeable data, such as custom server properties. To prevent two applications from simultaneously attempting to change the same property, the server object can be momentarily locked and then unlocked with the `server.lock()` and `server.unlock()` methods.

The project object

The scope of a project object is a single application running on the server. But this also means that all clients that access the application (even simultaneously) have access to project properties.

A project object is created when the application is started on the server. No properties are assigned to the project object by default. It is, rather, a convenient place for an application to store small chunks of data that all users of the

application (or the application, itself) can benefit from. Think of it as a global store for all instances of the running application. Perhaps the most common use of the project object is as a place to store the latest unique ID number for new user accounts or database records. If a new record is being added to a database, it is faster to access the project object property that holds the last assigned number, grab a copy, and increment the value in the property for the next time.

For the sake of data integrity, the project object can be locked momentarily while getting and incrementing that value. Other clients wanting access to that property are held in a queue for the next available access (that is, when the property object is unlocked). The simplest example of using the `lock()` method is shown here:

```
project.lock()
var newCustomerID = project.nextID
project.nextID = project.nextID + 1
project.unlock()
```

This sequence locks the project object, copies the current custom property that contains the next customer ID to be issued, increments the property, and unlocks the project object.

The client object

Narrowing the object scope further, the client object is unique to a particular client access of an application. A client object has no predefined properties, so your application is at liberty to assign property names and values that are of interest to a single connection to the application.

You can imagine that a highly trafficked application would drain a server of processing time, memory, and disk space if it had to track the client objects of perhaps hundreds or thousands of simultaneous connections. To counteract the problem, you specify in your application control panel (on the server application manager) whether client object data should be stored on the server or client and how that should be accomplished. From the server's point of view, the most convenient place for such storage is a client cookie. One example of how this works is a server-based shopping cart. When the user clicks to add a product to the cart, the form is submitted to the application. The application extracts pertinent information bits about the product and stores that information in one or more client properties (Enterprise Server writes such a cookie with a special name in the format `NETSCAPE_LIVEWIRE.`*`propertyName=value`*). With a client object specified as a client cookie, the data is sent to the client and saved in the browser cookie area (the same `document.cookie` described in Chapter 16). At check-out time, the application can retrieve the cookie data from the client and assemble a final order form.

Other client object techniques include client-side URL encoding, server cookies, and server URL encoding. Each approach has its advantages and disadvantages, all of which are detailed in Netscape's documentation for building server-side JavaScript applications.

The request object

The narrowest in scope of all objects is the request object. It contains information about a single submission from a client.

Even before you get to the form data being submitted, the request object has several properties filled automatically. By and large, these preset properties mirror the environment variables that traditional CGI programs extract from submissions. Table 36-1 shows the standard properties for the request object.

Table 36-1
Standard request Object Properties

Property	*Description*
`agent`	Browser type and version (the same as client-side `navigator.userAgent`)
`auth_type`	Authorization type (if any)
`auth_user`	Remote user (if available)
`ip`	IP address
`method`	Form method setting
`protocol`	Client protocol
`query`	Query string passed with method=GET
`imageX`	Area map click horizontal coordinate
`imageY`	Area map click vertical coordinate
`uri`	Partial URL of request (after protocol, hostname, and port)

For every named element in the form being submitted, a property is added to the current request object, and the value of the element is assigned to that property. Only one request object is available per client, so these values last only until the next submission (which might be in another frame immediately after the first frame's form is submitted).

Accessing the request object is key to capturing information entered or selected in a form by a user. For example, if you are updating a database with one field's data, you can use the request object's property to fill that column in the database table:

```
database.execute("UPDATE customer phone1 = '" + request.phone1 + "'
WHERE custID = " + request.custID)
```

A form's hidden objects are included among the request object's properties. It is not uncommon to use a hidden object to pass information from one server script execution to the next by writing hidden objects to the page and letting the next request object pick out the data for further processing.

Database Access with LiveWire

LiveWire is Netscape's brand name for the technology that links a server-side application to a database on the server. In the early days of this technology, it was a separate component of the company's server offerings. In SuiteSpot 3, LiveWire is integrated into the offering.

The purpose of LiveWire extends beyond the basic ability to link up a Web application and a database. The goal was to create an application programming interface (API) that lets Web application authors write essentially the same database-related code, regardless of whether the database is IBM's DB2, Oracle, Informix, Sybase, or any ODBC standard database. In these days of competing formats and standards, that is a tall order. Even so, LiveWire comes a long way in establishing a one-code-fits-all implementation, at least for the basic database access tasks (a lot of credit also goes to the Structured Query Language — SQL — standard adopted by many large database makers).

Database access process

Successful database access requires a specific sequence of statements at various places within your code. LiveWire provides a database object, the reference of which you use to open the connection and read or write data to the database.

One page of your application (it might be the home page) contains `<SERVER>` tag code that initiates the connection to the database on behalf of the client accessing the application. The basic format of this statement is as follows:

```
database.connect(databaseType, serverName, username, password,
databaseName)
```

The precise content of the parameters to the `connect()` method vary a great deal with the type of database you are accessing. In my experience, setting up an ODBC database to accept the connection can take a bit of doing, especially in getting the proper ODBC drivers installed.

Accessing records

Before you start writing code to insert or update records in a database, you should be familiar with common SQL commands and syntax for these operations in your database. Using LiveWire, you wrap the SQL expressions inside a `database.exec()` method. The parameter of the method is the precise SQL statement that the database needs for inserting or updating the table.

The real JavaScript fun comes when retrieving data from tables. You use the `database.cursor()` method to send an SQL `SELECT` command to the database, specifying the columns you want to extract and the search criteria to be applied to the table. For example, to extract the name and phone number columns for all records of a customer table whose state column is "FL," the JavaScript code reads

```
var cursor = database.cursor("SELECT name, phone from customer where
state = 'FL'")
```

If any data matches the criteria of the SELECT statement, the database.cursor() method returns the data in the form of a cursor object. The cursor object is something like a JavaScript array, but not quite. You cannot, for instance, obtain the number of records returned, because you cannot get the length of the object. But you can cycle through each row of returned data with the help of the cursor.next() method.

The instant the cursor object is created with the returned data, an internal row pointer is positioned immediately above the first record. One invocation of cursor.next() moves the pointer to the first row of data. At that point, you can extract individual column data by using the column name as a property. You continue looping through the cursor until there are no more records, at which point the cursor.next() method returns a value of false. The following script fragment extracts the same data as described above and then consolidates the data into an HTML table that is eventually written to the client:

```
var cursor = database.cursor("SELECT name, phone from customer where
state = 'FL'")
var output = "<TABLE><TR><TH>Name</TH><TH>Phone</TH></TR>"
while (cursor.next()) {
        output += "<TR><TD>" + cursor.name + "</TD>"
        output += "<TD>" + cursor.phone + "</TD></TR>"
}
output += "</TABLE>"
write(output)
```

For simple data extractions and table displays, such as the one just given, LiveWire even provides a shortcut statement for creating the HTML table, ready for writing to the client document. Many other cursor and database methods offer sufficient flexibility to build very complex applications atop very complex databases. This includes controlling transaction processes when they are implemented for the database.

Server or Client JavaScript?

When you have the ability to implement an application with server-side and client-side JavaScript, you may wonder how to use both—or if you should use client-side JavaScript at all. A lot depends on the known base of browsers used by application users and the amount of traffic on the site.

One supreme advantage an author has by implementing everything on the server is independence from the JavaScript vagaries from one browser version or brand to the next. Not that it solves the HTML compatibility issues, which you must still face. But when the only concern is HTML compatibility, the testing matrix of browser versions and brands is smaller.

Putting all the processing on the server end, however, may do a disservice to the users of browsers that have the power to do some of that processing locally. For every image change or other adjustment to the HTML page, the user must wait for the transactions with the server, server processing, and another download of data. If you have a high-traffic site, this can also place extra burdens on the server that hinder access to all.

Most commonly, however, authors strike a balance between server-side and client-side scripting to take care of the job. For example, data-entry validation on the client is orders of magnitude more efficient for the user and your server. The server still needs to do validation for non-scriptable browser users, but at least those with scriptable browsers won't be slowed down by server-generated error messages and form redrawing.

As more capabilities are built into modern browsers, such as positionable elements (Chapters 41 through 43), it makes more sense to imbue intelligence into documents that utilize those facilities. The experience will be much crisper for users, as if they're using a local software package, rather than constantly waiting for screen refreshing.

✦ ✦ ✦