Chapter 50: Application: A Lookup Table

In This Chapter

Severless data collection lookup

Data-entry validation

One of the first ideas that intrigued me about JavaScript was the notion of delivering CGI-like functionality along with an HTML document. On the Web, numerous, small data collections currently require CGI scripting and a back-end database engine to drive them. Of course, not everyone who has information to share has access to the server environment (or the expertise) to implement such a solution. JavaScript provides that power.

A Serverless Database

Before you get too carried away with the idea of letting JavaScript take the place of your SQL database, you need to recognize several limitations that prevent JavaScript from being a universal solution. First, any database that you embed into an HTML document is read-only. Although you can script an interface and lookup routines for the user, no provisions are available for writing revised information back to the server, if that is your intention.

A second consideration is the size of the data collection. Unlike databases residing on servers, the entire JavaScript database (or subset you define for inclusion into a single HTML document) must be downloaded to the user's browser before the user can work with the data. As a point of reference, think about image files. At 28.8 Kbps, how large an image file would you tolerate downloading? Whatever that limit may be (anywhere from 10 to 35K, depending on your patience) is what your database size limit should be. For many special-purpose collections, this is plenty of space, assuming one byte per character. Unlike what happens when the user downloads an embedded image file, the user doesn't see special statusbar messages about your database: To the browser, these messages are all part of the HTML coming in with the document.

The kind of data I'm talking about here is obviously text data. That's not to say you can't let your JavaScript-enhanced document act as a front end to data files of other types on your server. The data in your embedded lookup table can be URLs to images that get swapped into the page as needed.

The Database

As I was thinking about writing a demonstration of a serverless database, I encountered a small article in the *Wall Street Journal* that related information I had always suspected. The Social Security numbers assigned to virtually every U.S. citizen are partially coded to indicate the state in which you registered for your Social Security number. This information often reveals the state in which you were born (another study indicates that two-thirds of U.S. citizens live their entire lives in the same state). The first three digits of the nine-digit number comprise this code.

When the numbering system was first established, each state was assigned a block of three-digit numbers. Therefore, if the first three digits fall within a certain range, the Social Security Administration has you listed as being registered in the corresponding state or territory. I thought this would be an interesting demonstration for a couple of reasons: first, the database is not that large, so it can be easily embedded into an HTML document without making the document too big to download, even on slow Internet connections; second, it offers some challenges to data-entry validation, as you see in a moment.

> **Note**
>
> Before young people from populous states write to tell me that their numbers are not part of the database, let me emphasize that I am well aware that several states have been assigned number blocks not reflected in the database. This example is only a demonstration of scripting techniques, not an official Social Security Administration page.

The Implementation Plan

For this demonstration, all I started with was a printed table of data. I figured that the user interface for this application would probably be very plain: a text field in which the user can enter a three-digit number, a clickable button to initiate the search, and a text field to show the results of the lookup. Figure 50-1 shows the page. Pretty simple by any standards.

Given that user interface (I almost always start a design from the *interface* — how my page's users will experience the information presented on the page), I next planned the internals. I needed the equivalent of two tables: one for the numeric ranges, and one for the state names. Because most of the numeric ranges are contiguous, I could get by with a table of the high number of each range. This meant that the script would have to trap elsewhere for the occasional numbers that fall outside of the table's ranges — the job of data validation.

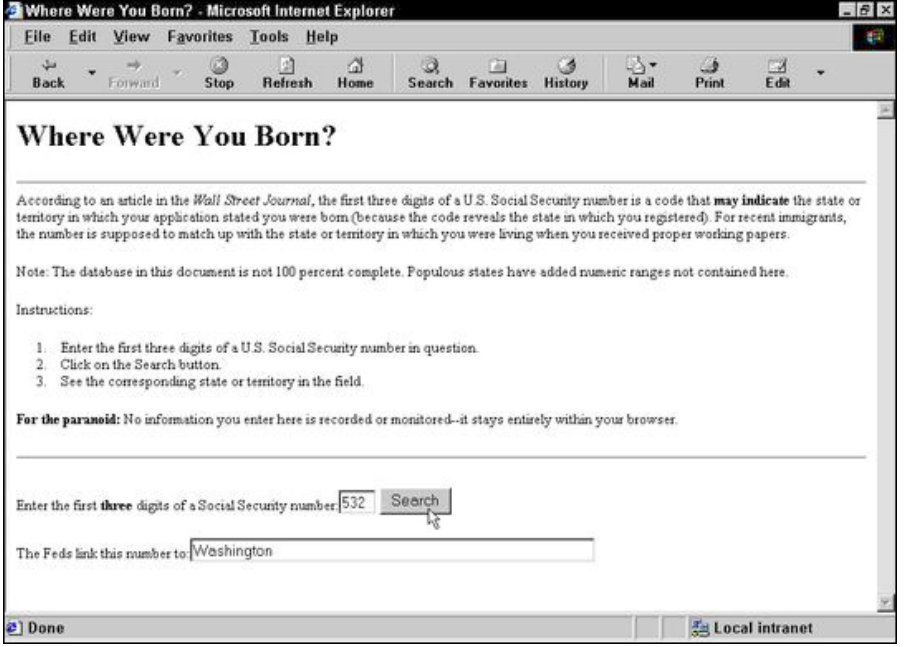

Figure 50-1
The Social Security number lookup page

Because the two tables were so closely related to each other, I had the option of creating two separate arrays, so that any given index value would correspond to both the numeric and state name entries in both tables (*parallel arrays*, I call them). The other option was to create a two-dimensional array (see Chapter 37), in which each array entry has data points for both the number and state name. For purposes of demonstration to first-time database builders, I decided to stay with two parallel arrays. This method makes visualizing how the lookup process works with two separate arrays a little easier.

The Code

The HTML document starts normally through the definition of the document title:

```
<HTML>
<HEAD>
<TITLE>Where Were You Born?</TITLE>
```

Because I chose to use the `Array` object of NN3 and IE3/J2, I added a separate script segment to gracefully handle the click of the button for those who landed at this page with an earlier scriptable browser. By putting a `<SCRIPT LANGUAGE="JavaScript">` tag ahead of the `<SCRIPT LANGUAGE="JavaScript1.1">` tag, I ensure that the one function triggered by the button is treated appropriately for all scriptable browsers:

```
<SCRIPT LANGUAGE="JavaScript">
<!-- hide from non-scriptable browsers
function search(form) {
    alert("This page a more recent browser version.")
}
// end hiding -->
</SCRIPT>
```

Immediately after the starting <SCRIPT> tag comes the HTML beginning comment, so that most non-JavaScript-enabled browsers ignore all statements between the start and end comments (just before the </SCRIPT> tag). Failure to do this results in all code lines appearing in non-JavaScript browsers as regular HTML text.

Now we come to the JavaScript 1.1-level scripts, which handle everything from building the tables of data to looking up data later in response to a button click. I begin by creating the first array for the top numbers of each entry's numeric range.

In this application, you will see that I place utility function definitions close to the top of the script sections and put any action-oriented scripts (functions acting in response to event handlers) closer to the bottom of the script sections. My preference is to have all dependencies resolved before the script needs them. This philosophy carries over from the logic that dictates putting as many scripts in the Head as possible, so that even if the user (or network) should interrupt downloading of a page before every line of HTML reaches the browser, any user interface element relying on scripts will have those scripts loaded and ready to go. The order of functions in this example is not critical, because as long as they all reside in the Head section, they are defined and loaded by the time the field and button appear at the bottom of the page. But after I develop a style, I find it easier to stick with it — one less matter to worry about while scripting a complex application.

After creating an array (named ssn) with 57 empty slots, the script populates all 57 data points of the array, starting with the first entry going into the slot numbered 0. These data numbers correspond to the top end of each range in the 57-entry table. For example, any number greater than 3 but less than or equal to 7 falls into the range of the second data entry of the array (ssn[1]).

```
<SCRIPT LANGUAGE="JavaScript1.1">
<!-- hide from non-scriptable browsers

// create array that lists the top end of each numeric range
var ssn = new Array(57)
ssn[0] = 3
ssn[1] = 7
ssn[2] = 9
ssn[3] = 34
ssn[4] = 39
ssn[5] = 49
ssn[6] = 134
ssn[7] = 158
ssn[8] = 211
ssn[9] = 220
ssn[10] = 222
ssn[11] = 231
ssn[12] = 236
ssn[13] = 246
ssn[14] = 251
ssn[15] = 260
ssn[16] = 267
ssn[17] = 302
ssn[18] = 317
ssn[19] = 361
ssn[20] = 386
ssn[21] = 399
```

```
ssn[22] = 407
ssn[23] = 415
ssn[24] = 424
ssn[25] = 428
ssn[26] = 432
ssn[27] = 439
ssn[28] = 448
ssn[29] = 467
ssn[30] = 477
ssn[31] = 485
ssn[32] = 500
ssn[33] = 502
ssn[34] = 504
ssn[35] = 508
ssn[36] = 515
ssn[37] = 517
ssn[38] = 519
ssn[39] = 520
ssn[40] = 524
ssn[41] = 525
ssn[42] = 527
ssn[43] = 529
ssn[44] = 530
ssn[45] = 539
ssn[46] = 544
ssn[47] = 573
ssn[48] = 574
ssn[49] = 576
ssn[50] = 579
ssn[51] = 580
ssn[52] = 584
ssn[53] = 585
ssn[54] = 586
ssn[55] = 599
ssn[56] = 728
```

I do the same for the array containing the states and territory names. Both of these array populators seem long but pale in comparison to what you would have to do with a database of many kilobytes. Unfortunately, JavaScript doesn't give you the power to load existing data files into arrays (but see the recommendations at the end of the chapter), so any time you want to embed a database into an HTML document, you must go through this array-style assignment frenzy:

```
// create parallel array listing all the states/territories
// that correspond to the top range values in the first array
var geo = new Array(57)
geo[0] = "New Hampshire"
geo[1] = "Maine"
geo[2] = "Vermont"
geo[3] = "Massachusetts"
geo[4] = "Rhode Island"
geo[5] = "Connecticut"
geo[6] = "New York"
geo[7] = "New Jersey"
geo[8] = "Pennsylvania"
geo[9] = "Maryland"
geo[10] = "Delaware"
geo[11] = "Virginia"
geo[12] = "West Virginia"
geo[13] = "North Carolina"
geo[14] = "South Carolina"
geo[15] = "Georgia"
geo[16] = "Florida"
geo[17] = "Ohio"
geo[18] = "Indiana"
```

```
geo[19] = "Illinois"
geo[20] = "Michigan"
geo[21] = "Wisconsin"
geo[22] = "Kentucky"
geo[23] = "Tennessee"
geo[24] = "Alabama"
geo[25] = "Mississippi"
geo[26] = "Arkansas"
geo[27] = "Louisiana"
geo[28] = "Oklahoma"
geo[29] = "Texas"
geo[30] = "Minnesota"
geo[31] = "Iowa"
geo[32] = "Missouri"
geo[33] = "North Dakota"
geo[34] = "South Dakota"
geo[35] = "Nebraska"
geo[36] = "Kansas"
geo[37] = "Montana"
geo[38] = "Idaho"
geo[39] = "Wyoming"
geo[40] = "Colorado"
geo[41] = "New Mexico"
geo[42] = "Arizona"
geo[43] = "Utah"
geo[44] = "Nevada"
geo[45] = "Washington"
geo[46] = "Oregon"
geo[47] = "California"
geo[48] = "Alaska"
geo[49] = "Hawaii"
geo[50] = "District of Columbia"
geo[51] = "Virgin Islands"
geo[52] = "Puerto Rico"
geo[53] = "New Mexico"
geo[54] = "Guam, American Samoa, N. Mariana Isl., Philippines"
geo[55] = "Puerto Rico"
geo[56] = "Long-time or retired railroad workers"
```

Now comes the beginning of the data validation functions. Under control of a master validation function shown in a minute, the `stripZeros()` function removes any leading 0s that the user may have entered. Notice that the instructions tell the user to enter the first three digits of a Social Security number. For 001 through 099, that means the numbers begin with one or two 0s. JavaScript, however, treats any numeric value starting with 0 as an octal value. Because I have to do some numeric comparisons for the search through the `ssn[]` array, the script must make sure that the entries (which are strings to begin with, coming as they do from text objects) can be converted to decimal numbers. The `parseInt()` function, with the all-important second parameter indicating Base 10 numbering, does the job. But because the remaining validations assume a string value, the integer is reconverted to a string value before it is returned.

```
// **BEGIN DATA VALIDATION FUNCTIONS**
// JavaScript sees numbers with leading zeros as octal values,
// so strip zeros
function stripZeros(inputStr) {
    return parseInt(inputStr, 10).toString()
}
```

The next three functions are described in full in Chapter 43, which discusses data validation. In the last function, a copy of the input value is converted to an integer to

enable the function to make necessary comparisons against the boundaries of acceptable ranges.

```
// general purpose function to see if an input value has been entered
// at all
function isEmpty(inputStr) {
    if (inputStr == null || inputStr == "") {
        return true
    }
    return false
}

// general purpose function to see if a suspected numeric input
// is a positive integer
function isNumber(inputStr) {
    for (var i = 0; i < inputStr.length; i++) {
        var oneChar = inputStr.charAt(i)
        if (oneChar < "0" || oneChar > "9") {
            return false
        }
    }
    return true
}

// function to determine if value is in acceptable range for this
// application
function inRange(inputStr) {
    num = parseInt(inputStr)
    if (num < 1 || (num > 586 && num < 596) || (num > 599 &&
        num < 700) || num > 728) {
        return false
    }
    return true
}
```

The master validation controller function (named `isValid()` in this application) is also covered in depth in Chapter 43. A statement that wants to know if it should proceed with the lookup process calls this function. If any one validation test fails, the function returns `false`, and the search does not proceed.

```
// Master value validator routine
function isValid(inputStr) {
    if (isEmpty(inputStr)) {
        alert("Please enter a number into the field before
clicking the button.")
        return false
    } else {
        if (!isNumber(inputStr)) {
            alert("Please make sure entries are numbers only.")
            return false
        } else {
            if (!inRange(inputStr)) {
                alert("Sorry, the number you entered is not part of our
database.  Try another three-digit number.")
                return false
            }
        }
    }
    return true
}
// **END DATA VALIDATION FUNCTIONS**
```

The `search()` function is invoked by two different event handlers (and indirectly by a third). The two direct calls come from the input field's `onChange` event handler and the

Search button's `onClick` event handler. The handler passes a reference to the form, which includes the button and both text objects.

To search the database, the script repeatedly compares each succeeding entry of the `ssn[]` array against the value entered by the user. For this process to work, a little bit of preliminary work is needed. First comes an initialization of a variable, `foundMatch`, which comes into play later. Initially set to `false`, the variable is set to `true` only if there is a successful match — information you need later to set the value of the result text object correctly for all possible conditions.

```
// Roll through ssn database to find index;
// apply index to geography database
function search(form) {
    var foundMatch = false
    var inputStr = stripZeros(form.entry.value)
    if (isValid(inputStr)) {
        inputValue = inputStr
        for (var i = 0; i < ssn.length; i++) {
            if (inputValue <= ssn[i]) {
                foundMatch = true
                break
            }
        }
    }
    form.result.value = (foundMatch) ? geo[i] : ""
    form.entry.focus()
    form.entry.select()
}
```

Next comes all the data preparation. After the entry is passed through the zero stripper, a copy is dispatched to the master validation controller, which, in turn, sends copies to each of its special-purpose minions. If the master validator detects a problem from the results of any of those minions, it returns `false` to the condition that wants to know if the input value is valid. Should the value not be valid, processing skips past the `for` loop and proceeds immediately to an important sequence of three statements.

The first is a conditional statement that relies on the value of the `foundMatch` variable that was initialized at the start of this function. If `foundMatch` is still `false`, that means that something is wrong with the entry and it cannot be processed. To prevent any incorrect information from appearing in the result field, that field is set to an empty string if `foundMatch` is `false`. The next two statements set the focus and selection to the entry field, inviting the user to try another number.

On the other hand, if the entry is a valid number, the script finally gets to perform its lookup task. Looping through every entry of the `ssn[]` array starting with entry 0 and extending until the loop counter reaches the last item (based on the array's length property), the script compares the input value against each entry's value. If the number is less than or equal to a particular entry, the value of the loop counter (`i`) is frozen, the `foundMatch` variable is set to `true`, and execution breaks out of the `for` loop.

This time through the conditional expression, with `foundMatch` being `true`, the statement plugs the corresponding value of the `geo[]` array (using the frozen value of `i`)

into the result field. Focus and selection are set to the entry field to make it easy to enter another value.

Browsers that recognize keyboard events benefit by allowing the search to be initiated if the user presses the Enter key after entering a number. An `onKeyPress` event handler for the input text box invokes the `searchOnReturn()` function. This function employs cross-browser event parsing to find out if the Return key had been pressed while the text pointer was in the text box. If so, then the `search()` function, described earlier, is asked to do its job. Any characters other than the Return key are allowed to pass unchanged to the input box.

```
// start search if input field receives a Return character
function searchOnReturn(form, evt) {
    evt = (evt) ? evt : (window.event) ? window.event : ""
    if (evt) {
        var theKey = (evt.which) ? evt.which : evt.keyCode
        if (theKey == 13) {
            search(form)
            return false
        }
    }
    return true
}
// end code hiding -->
</SCRIPT>
</HEAD>
```

The balance of the code is the Body part of the document. The real action takes place within the Form definition.

```
<BODY>
<H1>Where Were You Born?</H1>
<HR>

According to an article in the <CITE>Wall Street Journal</CITE>, the first three
digits of a U.S. Social Security number is a code for the state or territory in
which your application stated you were born.  For recent immigrants, the number
is supposed to match up with the state or territory in which you were living
when you received proper working papers.<P>
Note: The database in this document is not 100 percent complete.  Populous
states have added numeric ranges not contained here.<P>
Instructions:
<OL><LI>Enter the first three digits of a U.S. Social Security number in
question.</LI>
<LI>Click on the Search button.</LI>
<LI>See the corresponding state or territory in the field.</L1>
</OL>

<P><B>For the paranoid:</B> No information you enter here is recorded or
monitored—it stays entirely within your browser.<P>
<HR>
```

The form's `onSubmit` event handler is set to prevent accidental submission (or pseudo-submission, because no `ACTION` attribute is specified for the form) that IE/Mac does from any form's text box (other browsers submit on Return from only a single-field form). Each of the text objects is sized to fit the expected data. A handful of event handlers invoke the `search()` function (directly and indirectly), passing a reference to the form as a parameter.

```
<FORM onSubmit="return false">
```

```
Enter the first <B>three</B> digits of a Social Security number:<INPUT
TYPE="text" NAME="entry" SIZE=4 onKeyPress="return searchOnReturn(this.form,
event)" onChange="search(this.form)">
<INPUT TYPE="button" VALUE="Search" onClick="search(this.form)">
<P>
The Feds link this number to:<INPUT TYPE="text" NAME="result" SIZE=50>
</FORM>
</BODY>
</HTML>
```

Further Thoughts

If I were doing this type of application for production purposes, I would turn each pairing of range high number and geographical location into separate objects, and store the objects in an array. Making that technique work requires one extra function and a different way of populating the data. The following is an example using the same variable names as the preceding listing:

```
// specify an array entry with two items
function dataRecord(ssn, geo) {
    this.ssn = ssn
    this.geo = geo
    return this
}

// initialize basic array
var numberState = new Array(57)

// populate main array with smaller arrays
numberState[0] = new dataRecord(3,"New Hampshire")
numberState[1] = new dataRecord(7,"Maine")
numberState[2] = new dataRecord(9,"Vermont")
```

The other changes (marked in boldface) occur in the `search()` function, which must address this data in a slightly different way than it did before:

```
function search(form) {
    var foundMatch = false
    var inputStr = stripZeros(form.entry.value)
    if (isValid(inputStr)) {
        inputValue = inputStr
        for (var i = 0; i < numberState.length; i++) {
            if (inputValue <= numberState[i].ssn) {
                foundMatch = true
                break
            }
        }
    }
    form.result.value = (foundMatch) ? numberState[i].geo :""
    form.entry.focus()
   form.entry.select()
}
```

All references to data are to the `numberState[]` array and properties of its objects (either `ssn` or `geo`). With the data for each record arranged in a comma-delimited fashion, it may be easier to transfer data exported from an existing database to your script with less copying and pasting or dragging and dropping.

Another possibility would be to use JavaScript's capability to load `.js` files that have the arrays already populated or have variables preloaded with comma-delimited values. By

using the *`string`*`.split()` method (Chapter 34), you can easily assign data in this format to an array.

From a user interface perspective, the `searchOnReturn()` function can do more with the event object. For instance, it could filter data entry so that only numbers ever reach the input text field. You would still want to perform the data-entry validation in case someone were to paste some non-numeric text into the text box.

I truly believe that serverless data lookups offer a great opportunity to many creative JavaScripters.