Level: Intermediate
Works with: Notes/Domino
Updated: 06-Oct-2003

by
Mark
Polly

Keyword lists can drive you crazy. You know keyword lists—they let you provide choices for users, so they don't enter incorrect data. Many applications have screens that contain several keyword lists, often in which one list's choices depend on the answer chosen in previous lists. When developing for the Notes client, there are many features available to make these keyword lists work nicely. You can use @Dblookup formulas to create choices for the lists. Because Notes computes fields from top to bottom, keyword lists lower on a form can be based on answers provided in previous fields. And you can easily use LotusScript to build the choices when you need to access data from non-Notes databases.

When you develop for the Web, keyword lists begin to pose several challenges. First, if you have several keyword lists on your form and want to build the choices from previous answers, you introduce other issues. You can use the options "Refresh fields on keyword change" and "Refresh choices on document refresh" to have Domino rebuild the keyword lists. These features cause extra traffic on the server, cause the screen to jump around, and can cause other problems with JavaScript on your page. See the *LDD Today* article, "Application Performance Tuning, Part 2" for information about how these options can affect application performance.

You can resort to using JavaScript arrays to pre-build the lists and then to fill in the keyword list based on the arrays. This method works fine for small keyword lists, but can quickly become unusable when multiple lists are needed. For example, assume you need three keyword lists with the second list built off the first and the third list built off the first two lists. Your first list consists of 50 choices, your second list has 20 choices for each choice in the first keyword list, and the third list has 20 choices for each combination of keyword list one and two. You need three JavaScript arrays, the first containing 50 elements, the second containing 1,000 elements (50 x 20), and the third containing 20,000 elements. Pre-building these arrays takes a while.

Second, sometimes keyword lists need to be built using more complex coding than the Notes formula language can handle. Building keyword lists on the fly using LotusScript is not easy to accomplish because it isn't something you can code in response to an event triggered in the browser.

Finally, if you have a keyword list whose choices are based on the answer to a previous keyword choice, and the second list must be built from a call to a relational database, there is no obvious way to use LotusScript to access the relational database and to build the second keyword list.

In this first of a two-part article series, you will discover ways to overcome these keyword list limitations for your Web applications. We'll show you how to build a keyword list based on other keyword list answers or on data entered by the user without refreshing the page or building huge JavaScript arrays.

In part 2 of this series, we'll expand on these techniques to use LotusScript to build our keyword lists. Finally, we'll use this same technique to build our keyword lists based on data contained in a relational database.

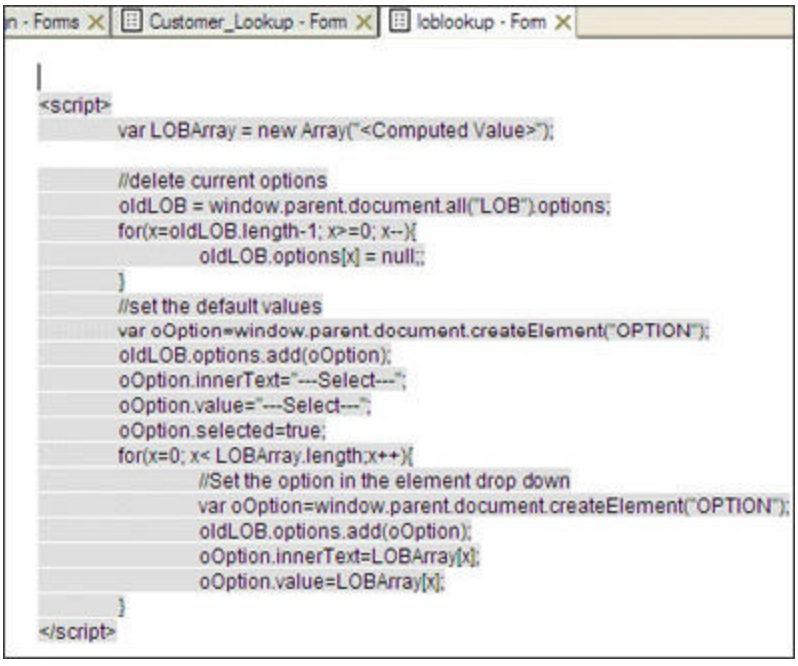
Building a single keyword list using @DBColumn

To implement this feature, we use a combination of @DBColumn and the <iframe> tag implemented by Microsoft Internet Explorer. An iframe is an internal frame on a Web page. If you aren't familiar with iframes, they're similar to embedded views on a Domino form. The embedded view appears to be part of the page to the end user, but in fact, the view is a completely separate page. The iframe works the same way—it appears embedded on the page, but its content is a separate page. Unlike an embedded view in Domino, the source page for an iframe can be on the same server or on a different server.

Note: If you use a browser that doesn't support iframes, you can use the same technique by opening a new browser window off-screen. You can also use a standard frameset and hide one of the frames.

In the following screen, which displays our input form, the iframe is shown with the parameters width=0, height=0 and src=/bigdb.nsf/loblookup?Readform. The width and height parameters tell the browser to display the iframe with no width and height, which effectively hides the iframe from the user. The src parameter tells the browser which page to load in the iframe. You can see that the page is actually a Domino form from our server.

The loblookup form is in our big database on our server. It is here that we perform our @Dbcolumn. The loblookup form is shown in the following illustration:



Because this form is never displayed to the user, it only consists of JavaScript that runs after the page loads in

the iframe. The complete JavaScript code is as follows:

```
<script>
   var LOBArray = new Array("<Computed Value>");

    //delete current options
    oldLOB = window.parent.document.all("LOB").options;
    for(x=oldLOB.length-1; x>=0; x--){
        oldLOB.options[x] = null;;
    }
    //set the default values
    var oOption=window.parent.document.createElement("OPTION");
    oldLOB.options.add(oOption);
    oOption.innerText="---Select---";
    oOption.value="---Select---";
    oOption.selected=true;
    for(x=0; x< LOBArray.length;x++){
        //Set the option in the element drop down
        var oOption=window.parent.document.createElement("OPTION");
        oldLOB.options.add(oOption);
        oOption.innerText=LOBArray[x];
        oOption.value=LOBArray[x];
    }
</script>
```

Line 2 of the JavaScript code contains a <Computed Value> with this formula:

```
temp := @DbColumn( "" ; "" ; "LOBs" ; 1 ) ;
val:=@If(@IsError(temp); "No LOB"; temp);
@Implode(val;"\",\"")
```

Our formula performs an @Dblookup on the current database (bigdb.nsf) on line 1. If the @Dbcolumn is successful, the value returned is imploded using “,” as the separator. If the data contained Marketing:Finance:Public Affairs, the @implode function on line 3 would return Marketing”,”Finance”,”Public Affairs. This is inserted into the JavaScript code resulting in this line at run time:

```
Var LOBArray = new Array(“Marketing”,”Finance”,”Public Affairs”);
```

Now we have a standard JavaScript array containing all the values from our @Dbcolumn in bigdb.nsf.

The rest of the JavaScript does the work of filling in the keyword list choices on our page running on server1. Line 5 sets the oldLOB object to the LOB keyword list located on the parent.document, which is the page containing the iframe.

```
oldLOB = window.parent.document.all("LOB").options;
```

Lines 6-7 delete the choices (options) in the LOB keyword list.

```
For(x=oldLOB.length-1; x>=0; x--){
    oldLOB.options[x] = null;;
```

Line 10 creates a new Option object on the parent.document.

```
var oOption=window.parent.document.createElement("OPTION");
```

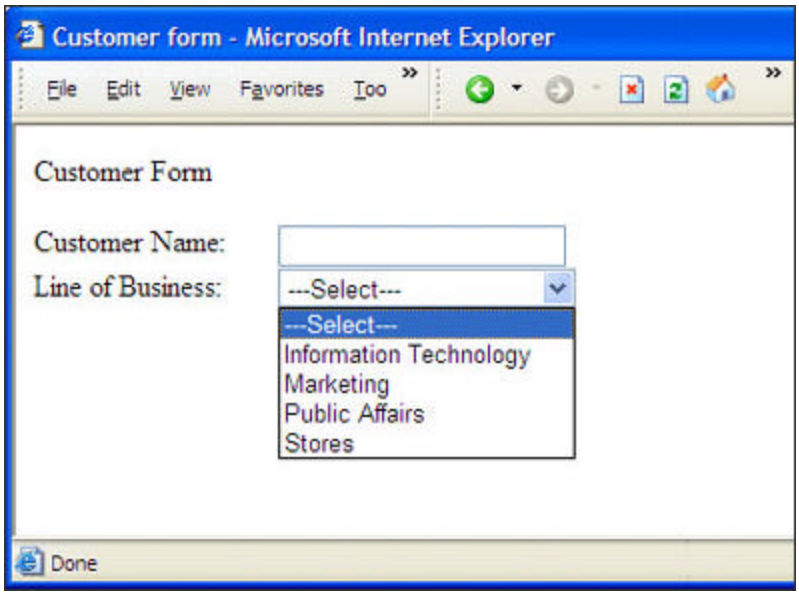
Lines 11-12 add the new option object to our LOB keyword list and set the first option value to “---Select---“.

```
oldLOB.options.add(oOption);
oOption.innerText="---Select---";
```

Lines 15–21 create an option object for each element of our LOBArray.

```
    for(x=0; x< LOBArray.length;x++){
    //Set the option in the element drop down
    var oOption=window.parent.document.createElement("OPTION");
    oldLOB.options.add(oOption);
    oOption.innerText=LOBArray[x];
    oOption.value=LOBArray[x];
}
```

When this JavaScript runs, the LOB keyword list on our main form is filled with choices gathered from bigdb.nsf.



For the page that loads in the iframe, Domino authenticates the user if that database restricts anonymous access. One easy way to ensure that the user isn't prompted for a password is to utilize Single Sign-on across the servers.

Let's review

Although the code looks simple, we've learned quite a bit already. Through this sample, we've learned:

1. How to simulate a keyword refresh by launching a page in an iframe
2. How to perform @DBLookup and @DBColumns formulas
3. How to combine @Formulas with JavaScript to fill in a keyword list

Now that we know the basics, let's turn our attention to working with several keyword lists.

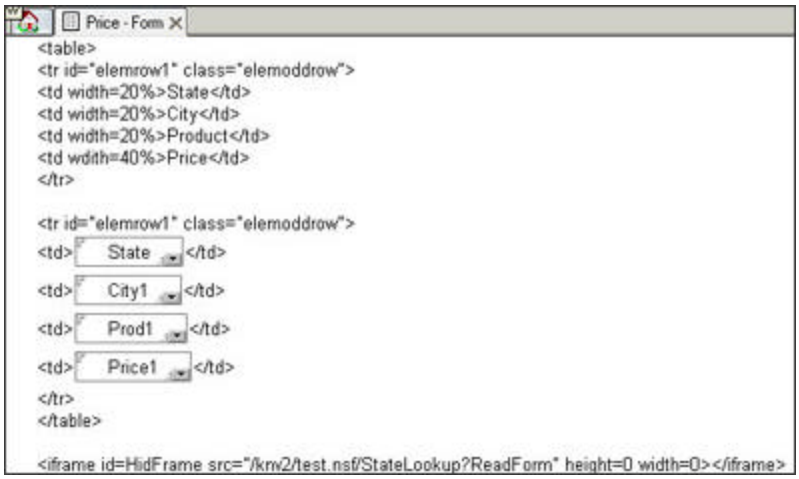
Building cascading keyword lists

As mentioned, there are several techniques for building keyword lists. Each method works well when the lists are small. However, when you have list combinations that reach into the thousands, the other techniques fall short.

As an example, let's work with a pricing database. In our application, we have prices customized by state, then by city, and then by product. Our first keyword list contains 50 entries, one for each U.S. state. After we select a state, we then need to pick a city. Each state has between 30 and 40 cities where we do business. The total combinations of state and city are at most 2,500. After we select a state and city, we then have to pick a product.

For each city, we have up to 20 products. As a result, our combination of state-city-product gives us 50,000 combinations.

Because of the large number of combinations, we can't build the keyword lists when our Web page loads. We can only fill in the city's list when the user selects a state. Likewise, we have to wait to fill in the product list until the user selects both a state and city. A sample screen appears as follows:



A look at the code

Now let's see how to do this. First, we build off the work we did in one keyword list. For our first keyword list (the list of states), we fill in the keyword list using the same technique as described earlier in this article. The iframe code is:

```
<iframe id="HidFrame" height=0 width=0 src="/bigdb.nsf /StateLookup?ReadForm"></iframe>
```

After the user selects a state, we need a way to build the city list. In our state field, we use the onChange event, which requires JavaScript. The code is:

```
1. st = this[this.selectedIndex].text;
2. document.all('HidFrame [delete: lookupframe]').src= '/bigdb.nsf /CityLookup?ReadForm&state=' + st;
```

Line 1 retrieves the name of the state the user selected. Because this code is in the onChange event, "this'" refers to the state keyword field. Line 2 sets the source parameter of the iframe to a new page (CityLookup) and passes the state name to that page using the query string. When this line executes in the browser in response to the user selecting a state name, the CityLookup page is loaded into the iframe. To use JavaScript to set the source page of the iframe, we simply give our iframe an ID parameter like this:

```
<iframe id="HidFrame" ….></iframe>
```

On the CityLookup page, we again use the same combination of @Dblookup and JavaScript that we used before, except we add a field to capture the state value from the query string formula for "State" field:

```
QSLst:=@Explode(@Explode(Query_String_Decoded;"&");"=");
loc:=@Member("state"; QSLst);
@If(loc>0; @Subset(@Subset(QSLst;loc+1);-1);"")
```

There is a wide variety of methods for grabbing a value from the query string. The formula for our state field uses one technique. On line 1, it gets the Query_String_Decoded value and explodes it once using "&" as the separator. This results in a list in which each element is in the form &var=value. Next, another @explode is performed using "=" as the separator. The result is a list in which the elements are in the form var:value:var:value. On line 2, we find the location of state in our list. Finally on line 3, we get the value of state by

getting the next element in the list.

The JavaScript on the page is as follows:

```
<script>
    var CityArray = new Array("<Computed Value>");

    //delete current options
    oldCity = window.parent.document.all("City").options;
    for(x=oldCity.length-1; x>=0; x--){
        oldCity.options[x] = null;;
    }
    //set the default values
    var oOption=window.parent.document.createElement("OPTION");
    oldCity.options.add(oOption);
    oOption.innerText="---Select---";
    oOption.value="---Select---";
    oOption.selected=true;
    for(x=0; x< CityArray.length;x++){
        //Set the option in the element drop down
        var oOption=window.parent.document.createElement("OPTION");
        oldCity.options.add(oOption);
        oOption.innerText=CityArray[x];
        oOption.value=CityArray[x];
    }
</script>
```

Line 2 contains a <Computed Value> with this formula:

```
temp := @DbLookup( "" ; "" ; "CityState" ; “State”; 2 ) ;
val:=@If(@IsError(temp); "No City"; temp);
@Implode(val;"\",\"")
```

Just as before, we perform the @Dblookup to get a list of cities based on the state value passed in the query string. The list of cities is stored in a JavaScript array. This works great because we only return 40 cities at a time. The rest of the code takes the values of the array and builds the city keyword list on the parent document.

We’re almost done. Now that we have a list of cities, we need to wait for the user to select a city, and then we can build the list of products. As you can imagine, we simply reuse the same process that we used for the city field. Briefly, we code the onChange event of the City field to load a new page in our iframe. In the source URL, we pass both the city and state values in the query string. The page in the iframe performs a Dblookup on our remote server and sets the product list to the results. Again, the resulting list contains only 50 entries at most, which we easily handle with JavaScript.

Finally, we use the same technique on the product field to perform a lookup to get our custom price.

Summary

We’ve now demonstrated a very powerful way of building keyword lists in our Web-based applications that can only be created when the user selects an option on the page. To summarize this technique, we followed these steps:

1. Include a hidden iframe on our page.
2. Use JavaScript to load a new page into the iframe when the user makes a selection in a keyword list.
3. Get our lookup value from the query string and perform a Dblookup using that value.
4. Use JavaScript to build the next keyword list or to fill in a field based on the Dblookup.

In the next installment, we look at how to use an agent to build our keyword list. Finally, we examine how to build our keyword lists by accessing a relational database.

ABOUT THE AUTHOR

Mark Polly is a Technical Architect with Meritage Technologies, Inc. Meritage is an employee-owned technology consulting company that provides professional services to Global 3500 and Public Sector organizations. Mark is a certified Lotus Notes developer and has worked with Lotus Notes since Release 1. Since 1996, Mark has consulted with a variety of companies on Notes, Domino, and other technology projects.

