



Level: Intermediate
Works with: Notes/Domino
Updated: 13-Oct-2003

by
Mark
Polly

When you develop for the Web, keyword lists begin to pose several challenges, especially if you have several keyword lists on your form and want to build the choices for fields based on previous answers. You can use the field properties "Refresh fields on keyword change" and "Refresh choices on document refresh" to have Domino rebuild the keyword lists. However, these features cause extra traffic on the server, cause the screen to jump around, and can cause other problems with JavaScript on your page.

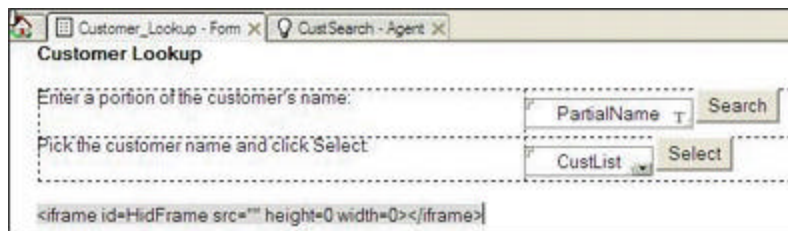
In [part one](#) of this article series, we walked through how to build keyword lists on the Web using JavaScript and the Notes @formula language. We used a hidden iframe to load a Domino form that performed @DBCColumn and @DBLookup functions to build our keyword lists. To summarize this technique, we 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 database lookup using that value.
4. Use JavaScript to build the next keyword list or to fill in a field based on the database lookup.

In this installment, we look at how to use a LotusScript agent to build our keyword lists. Finally, we examine how to build our keyword lists by accessing a relational database. This article assumes that you are an experienced LotusScript programmer with knowledge of JavaScript.

Building keyword lists using LotusScript

Let's assume that in our application, we want the user to select a customer name from a drop-down list. Because our customer database has several thousand customer names in it, we've decided to let the user type in part of the name. The system then performs a full-text search for all customers containing that name and the resulting list of names is included in our drop-down list. The user then picks the actual customer name from that list. The following screen shows our lookup form.



The form uses two fields: PartialName and CustList. The PartialName field is a text field in which you can enter

www.lotus.com/ldd/today.nsf

a portion of the customer's name, then click the Search button to retrieve the list of names. The CustList field is a drop-down field from which you can choose a customer name.

To perform the full-text search after the user types in part of the name in the PartialName field, we use LotusScript to create a search agent that we call CustSearch. (Note that we could also do the same using Java.) The Search button next to the PartialName field has the following JavaScript code:

```
pname=document.all.PartialName.value;
hidframe = document.all.HidFrame;
hidframe.src = "/" + thisdbname + "/CustSearch?OpenAgent&pname=" + pname;
```

Line 1 gets the values of the text entry field. Line 2 sets a reference to our hidden frame on the page. Line 3 then sets the src property of the hidden iframe to a URL that calls the CustSearch agent in our database. When this line executes, the browser launches this URL in the hidden iframe, which causes the agent to execute. The JavaScript variable thisdbname is set in the HTMLHead on the form, so it is a global variable.

The CustSearch agent

The CustSearch agent does most of the work in our customer search application. The complete LotusScript code is available in a [sidebar](#), but we break the code into chunks to describe it. (See the sidebar [Agent security](#) for a brief introduction to security concerns when running agents on the Web.)

In the following code snippet, the first six lines are standard DIMs that define the objects we use in the agent. Line 7 retrieves the current database object, and line 8 retrieves the document context for our agent. The document context for an agent provides the standard CGI variables, such as Query_String_Decoded, defined as fields on the Notes document. On line 10, we grab the Query_String_Decoded value from the Notes document. The query string contains the text the user typed before hitting the search button. Line 11 uses the LotusScript function StrRight to parse the query string and to return the text after the equals sign.

Sub Initialize

```
Dim s As New NotesSession
Dim db As NotesDatabase
Dim note As NotesDocument
Dim custNote As NotesDocument
Dim coll As NotesDocumentCollection
Set db = s.CurrentDatabase
Set note = s.DocumentContext
'get the customer name from the querystring
qs = note.Query_String_Decoded(0)
pname = StrRight(qs, "=") 'get the string to the right of the equal sign
```

Now we have the text the user wants to use for the search. On line 12, we define a string to contain the query for our full-text search. In this case, we search for documents in which the field CustName is like the text entered on our search form. Line 13 issues the FTSearch command using our query and several standard full-text search options. Finally, line 14 gets the first document returned by the ftsearch method.

```
query = "[CustName] Like " & pname
Set coll = db.Ftsearch(query, 0, FT_SCORES, FT_FUZZY + FT_STEMS)
Set custNote = coll.GetFirstDocument
```

Up to this point, we haven't done anything but write standard LotusScript code. Now we need to have our agent generate JavaScript code to fill in the CustList keyword on our Web page. Line 15 begins this by printing the <script> tag. When an agent runs from a Web browser, as ours does, the print statement sends output to the Web page. The <script> tag tells the browser to interpret the following code as JavaScript until it finds a </script> tag.

```
Print "<script>"
```

The first thing our JavaScript code has to do is clear out any old values from the keyword list. Line 17 sets a

www.lotus.com/ldd/today.nsf

JavaScript object to the options object from the Web page. Lines 18 through 20 are a For loop that deletes each exiting option in the option object.

```
Print | oldcustList = parent.document.all("CustList").options;|
Print | for(x=oldcustList.length-1; x>=0; x--){|
Print | oldcustList.options[x] = null;;|
Print | }|
```

In the following code snippet, line 22 begins a Do...While loop, getting one document from our search results during each iteration of the loop. For each search results document, we get the value of the CustName field on line 23. Line 25 begins the next set of print statements, so we are again writing JavaScript code to the browser. Line 25 creates a new option object on the page. Line 26 adds this new option object to our list options in the keyword list. Line 27 writes the JavaScript code to set the text of the keyword option to the value of the CustName field for the document. Line 28 writes the JavaScript code to set the value of the keyword option to the value of the CustName field for the document. At this point, the keyword list has grown by one entry.

```
Do While Not(custNote Is Nothing)
custName = custNote.CustName(0)
'add the new option
Print | var oOption=parent.document.createElement("OPTION");|
Print | oldcustList.options.add(oOption);|
Print | oOption.innerText="| & custName & "||
Print | oOption.value="| & custName & "||
```

Lines 30 and 31 get the next document from our search results and continue the loop. For every document that our search found, the JavaScript code adds an option to our keyword list using the customer name as the entry.

```
Set custNote = coll.GetNextDocument( custNote)
Loop
```

Finally, line 33 prints out the closing </script> tag, and our agent's execution ends. After the agent ends, the Web page in our hidden iframe contains the JavaScript to build our customer name keyword list.

```
Print "</script>"
End Sub
```

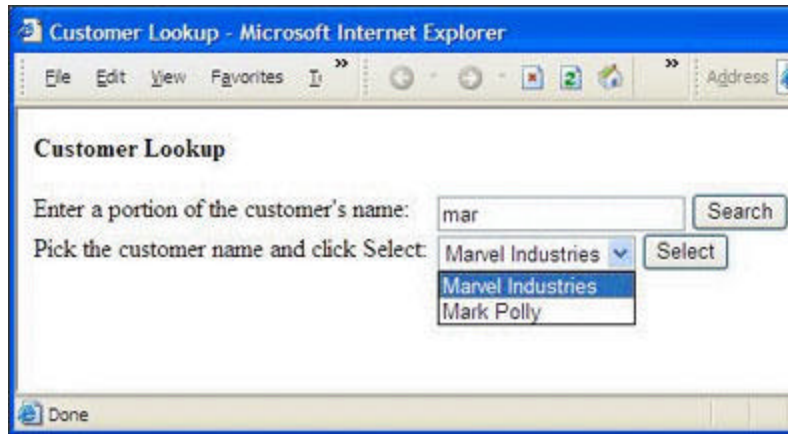
The JavaScript executes when the page finishes loading. Below is the source code for the page generated by our agent. Notice that the lines that set the .innerText (what the user sees) and .value (the data that is submitted, just like an alias in Notes) now contain data from the documents found using FTSearch.

```
<html>
<head>
</head>
<body text="#000000">
<script>
oldcustList = parent.document.all("CustList").options;
for(x=oldcustList.length-1; x>=0; x--){
oldcustList.options[x] = null;;
}
var oOption=parent.document.createElement("OPTION");
oldcustList.options.add(oOption);
oOption.innerText="Marvel Industries";
oOption.value="Marvel Industries";
var oOption=parent.document.createElement("OPTION");
oldcustList.options.add(oOption);
oOption.innerText="Mark Polly";
oOption.value="Mark Polly";
</script>
```

www.lotus.com/ldd/today.nsf

```
</body>  
</html>
```

Our customer search Web page now contains our customized keyword list, as shown.



At this point, we now have a way to use LotusScript to dynamically build our keyword lists based on the results of a full-text search. As we mentioned, you could also use a Java agent to accomplish the same goal.

Integrating relational data into keyword lists

Now that we know we can call an agent to build a keyword list for us, why not extend this concept to include data from a relational database? An agent can easily access relational data using LSX and ODBC. We can also use Java and JDBC to access relational data. In our next example, we build on our LotusScript agent to access our customer list from DB2.

We start with the same concepts as above. Our customer list is too big to include in the keyword list, so we ask the user to narrow down the choices by giving us text to use in a search. Our customer search form is the same as shown previously. Our search button's JavaScript code looks nearly the same, except we call a new agent, RDBCustSearch:

```
pname=document.all.PartialName.value;  
hidframe = document.all.HidFrame;  
hidframe.src = "/" + thisdbname + "/RDBCustSearch?OpenAgent&pname=" + pname;
```

The RDBCustSearch agent

Just as before, our RDBCustSearch agent is a LotusScript agent. The complete code example is available in the [sidebar](#), but to describe the code, we've broken it down into chunks.

The first 20 lines of our agent are very similar to the previous example. We DIM our variables and objects, get the session and document context, and then parse out the Query_String_Decoded data.

Sub Initialize

```
Dim s As New NotesSession  
Dim db As NotesDatabase  
Dim note As NotesDocument  
Dim custNote As NotesDocument  
Dim coll As NotesDocumentCollection  
Dim con As ODBCConnection  
Dim qry As ODBCQuery  
Dim res As ODBCResultSet  
Dim dataSource As String  
Dim userName As String
```

www.lotus.com/ldd/today.nsf

```
Dim password As String
Dim SQLStmt As String

Set db = s.CurrentDatabase
Set note = s.DocumentContext
'get the customer name from the querystring
qs = note.Query_String_Decoded(0)
pname = Strright(qs, "=") 'get the string to the right of the equal sign
```

On lines 21-23, we establish our ODBCConnection, ODBCQuery, and ODBCResultSet objects. Lines 24-26 set variables that are used to connect to the database. Then line 28 attempts to connect to the database using the datasource, user ID, and password values. If our connection succeeds, then on line 31 we connect our query object to the database.

```
Set con = New ODBCConnection
Set qry = New ODBCQuery
Set res = New ODBCResultSet
dataSource = "CUSTDB"
userName = "MPOLLY"
password = "test"

If Not con.ConnectTo(dataSource, userName, password) Then
    Messagebox "Could not connect to " & dataSource
Else
    Set qry.Connection = con
End If
```

Lines 34-37 define the SQL statement we need to query our DB2 database. In this SQL, we ask for the Cust_Nm field when that field is "like" the value we parsed from the query string. Lines 39-41 connect our SQL statement to our query object, and then execute the query. Results of the query are returned in the res object. If there is no data returned, then we exit our agent on line 43. If this happens, the keyword list in our form remains unchanged.

```
SQLStmt = "SELECT LTRIM(TRANSLATE(B.CUST_NM, ", '*')) AS CUST_NAME" &_
"FROM CUSTDB.EDW00T.VRPM_CUST_DIM   B " &_
"WHERE B.CUST_NM LIKE '*' & pname & '%" OR B.CUST_NM LIKE '" & pname & '%" &_
"ORDER BY CUST_NAME"

qry.SQL = SQLStmt
Set res.Query = qry
res.Execute
If Not(res.IsResultSetAvailable) Then ' No matches were found exit
    Exit Sub
End If
```

When we do get results, we start printing out our JavaScript statements on line 46. The <script> tag tells the browser that JavaScript will follow. Just as in our LotusScript example, we first want to delete any old entries in our keyword list. Line 48 sets our oldcustList object to the keyword object on our page. Lines 49-51 go through a For loop and delete all the old entries by setting each object to null.

```
Print "<script>"
'delete current options in the keyword list
Print | oldcustList = parent.document.all("CustList").options;|
Print | for(x=oldcustList.length-1; x>=0; x--){|
Print |   oldcustList.options[x] = null;;|
Print | }|
```

On line 53, we perform a Do loop. This loop runs until we reach the end of our result set (line 63). The first thing we do is retrieve the next row in our result set (line 54), which is record 1 the first time through the loop. On line

www.lotus.com/ldd/today.nsf

57, we create a new option object on our search page. On line 58, we add that new option to our keyword list. Next, we set the text displayed in the keyword list using the .innerText property and set the value property. The value property works like an alias on a Notes form. After all the data from the results set has been processed, we close the <script> tag on line 65.

```
Do
  res.NextRow
  custName = res.GetValue(Cust_Name)
  'add the new option
  Print | var oOption=parent.document.createElement("OPTION");|
  Print | oldcustList.options.add(oOption);|
  Print | oOption.innerText="| & custName & "|;|
  Print | oOption.value="| & custName & "|;|
Loop Until res.IsEndOfData
'close the javascript tag
Print "</script>"
End Sub
```

Refer to the previous screen to see what our page looks like with our Keyword list filled in after we perform a search.

Summary

Keyword lists are very powerful tools in our Web applications. Without them, users can enter bad data into the system. With them, we ensure good data is created from the beginning. However, because of HTML's stateless condition, it is difficult to build keyword lists using data entered on a form or to build them from other keyword lists on the page. The techniques we've outlined provide a powerful set of tools to let you build dynamic keyword lists on the Web just as you would in the Notes client.

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.

CustSearch agent code

Here is the complete LotusScript code for the CustSearch agent.

```

Sub Initialize
    Dim s As New NotesSession
    Dim db As NotesDatabase
    Dim note As NotesDocument
    Dim custNote As NotesDocument
    Dim coll As NotesDocumentCollection
    Set db = s.CurrentDatabase
    Set note = s.DocumentContext
    'get the customer name from the querystring
    qs = note.Query_String_Decoded(0)
    pname = Strright(qs, "=") 'get the string to the right of the equal sign
    query = "[CustName] Like " & pname
    Set coll = db.Ftsearch(query, 0, FT_SCORES, FT_FUZZY + FT_STEMS)
    Set custNote = coll.GetFirstDocument
    Print "<script>"
    'delete current options in the keyword list
    Print | oldcustList = parent.document.all("CustList").options;|
    Print | for(x=oldcustList.length-1; x>=0; x--){|
    Print |   oldcustList.options[x] = null;;|
    Print | }|
    'Now add the list of the customers returned from the ftsearch
    Do While Not(custNote Is Nothing)
        custName = custNote.CustName(0)
        'add the new option
        Print | var oOption=parent.document.createElement("OPTION");|
        Print |   oldcustList.options.add(oOption);|
        Print |   oOption.innerText="| & custName & "|;|
        Print |   oOption.value="| & custName & "|;|
        'get the next customer from the search results
        Set custNote = coll.GetNextDocument( custNote)
    Loop
    'close the javascript tag
    Print "</script>"
End Sub

```

RDBCustSearch agent code

Here is the complete LotusScript code for the RDBCustSearch agent.

```

Sub Initialize
    Dim s As New NotesSession
    Dim db As NotesDatabase
    Dim note As NotesDocument
    Dim custNote As NotesDocument
    Dim coll As NotesDocumentCollection
    Dim con As ODBCConnection
    Dim qry As ODBCQuery

```

www.lotus.com/ldd/today.nsf

```

Dim res As ODBCResultSet
Dim dataSource As String
Dim userName As String
Dim password As String
Dim SQLStmt As String

Set db = s.CurrentDatabase
Set note = s.DocumentContext
'get the customer name from the querystring
qs = note.Query_String_Decoded(0)
pname = Strright(qs, "=" ) 'get the string to the right of the equal sign

Set con = New ODBCConnection
Set qry = New ODBCQuery
Set res = New ODBCResultSet
dataSource = "CUSTDB"
userName = "MPOLLY"
password = "test"

If Not con.ConnectTo(dataSource, userName, password) Then
    MsgBox "Could not connect to " & dataSource
Else
    Set qry.Connection = con
End If

SQLStmt = "SELECT LTRIM(TRANSLATE(B.CUST_NM, ", '*')) AS CUST_NAME" & _
"FROM CUSTDB.EDW00T.VRPM_CUST_DIM B " & _
"WHERE B.CUST_NM LIKE '*' & pname & '%" OR B.CUST_NM LIKE '" & pname & '%" & _
"ORDER BY CUST_NAME"

qry.SQL = SQLStmt
Set res.Query = qry
res.Execute
If Not(res.IsResultSetAvailable) Then ' No matches were found exit
    Exit Sub
End If

Print "<script>"
'delete current options in the keyword list
Print | oldcustList = parent.document.all("CustList").options;|
Print | for(x=oldcustList.length-1; x>=0; x--){|
Print | oldcustList.options[x] = null;;|
Print | }|
'Now add the list of the customers returned from the ftsearch
Do
    res.NextRow
    custName = res.GetValue(Cust_Name)
    'add the new option
    Print | var oOption=parent.document.createElement("OPTION");|
    Print | oldcustList.options.add(oOption);|
    Print | oOption.innerText="| & custName & "||
    Print | oOption.value="| & custName & "||
Loop Until res.IsEndOfData
'close the javascript tag
Print "</script>"
End Sub

```

Agent security

Running LotusScript and Java agents on the Domino server requires knowledge of the security setup of the server. For example, in Release 5, Web-based agents cannot access databases on other servers. In Notes/Domino 6, that restriction has been removed through the use of Run-on-Behalf-of capabilities. In any of the releases, security settings in the server document located in the Domino Directory govern who can run agents and what type of agents they can run.

Notes has two types of agents that can be run: Unrestricted and Restricted. Unrestricted agents are those that access the file system or that run operating system commands. Access to the file system means that the agent can read, write, and delete files through the operating system. Because these types of operations can be potentially dangerous to the integrity of the server, Domino allows only users with the correct access to run unrestricted agents. In the Server document, you enter the names of the users or groups who can run these agents into the Run unrestricted Agents (R5) or Run unrestricted methods and operations (Notes/Domino 6) field. Java agents have similar fields that allow you to define who can run unrestricted Java agents. It is important to restrict the list of users to only those whom you trust to have unrestricted access to the server.

Restricted agents, whether LotusScript or Java agents, don't access the file system or run operating system commands. Typically, restricted agents send email, modify Notes documents, and other Notes-specific tasks. The agents described in this article are restricted agents. Generally, the Server document allows any user to run restricted agents. You can set this up by setting the field Run restricted agents to an asterisk (*). This allows any user to run restricted agents on the server.

Developers sign agents when the agent is saved. Usually, the server uses this signature to determine if the user is allowed to run the agent based on the unrestricted/restricted settings in the Server document. Many times, however, when your form invokes an agent through the WebQuerySave or WebQueryOpen events, you want the agent executed based on the Web user's credentials. On the Property page of the agent, you can select the option Run as web user. When selected, the agent runs as if the current user had saved (signed) the agent. If you allow a limited set of users to run unrestricted agents, then the agent designated as Run as web user may not run. Likewise, if you limit the list of users that can run restricted agents, your agent may not run if the current user is not in the list.

As a general rule, then, make sure your agents are signed by the correct user ID, given your particular server security settings. Also when you select Run as web user, ensure that the users invoking the agent have the appropriate ability to run agents on the server.