Lotus Developer Domain
The Technical Resource for Lotus software
LDD Today

Level: Intermediate
Works with: Designer 4.6
Updated: 02-Mar-98

by
Mark Gordon
and
Andrew Broyles

*[Editor's note: This is the second article in a two-part series on how to add "friendlier" user interface features to your Web applications. This article focuses on how to create views that allow users to perform actions on multiple documents, and even change keyword fields on multiple documents. The **first article** showed you how forms and views work together on the Web, and how to use them to add a quick-find search bar to your Web applications.]*

No matter how powerful your Web application is, an unfriendly user interface will drive your users away. Domino uses views to automatically create a way for users to easily navigate through the documents in your application. But these Web views don't automatically include some of the useful features available to Notes users, such as the ability to select multiple documents from a view to do things like delete them or change their status. This article will show you how you can get these features "back" into your Web views to create a much friendlier interface for your users.

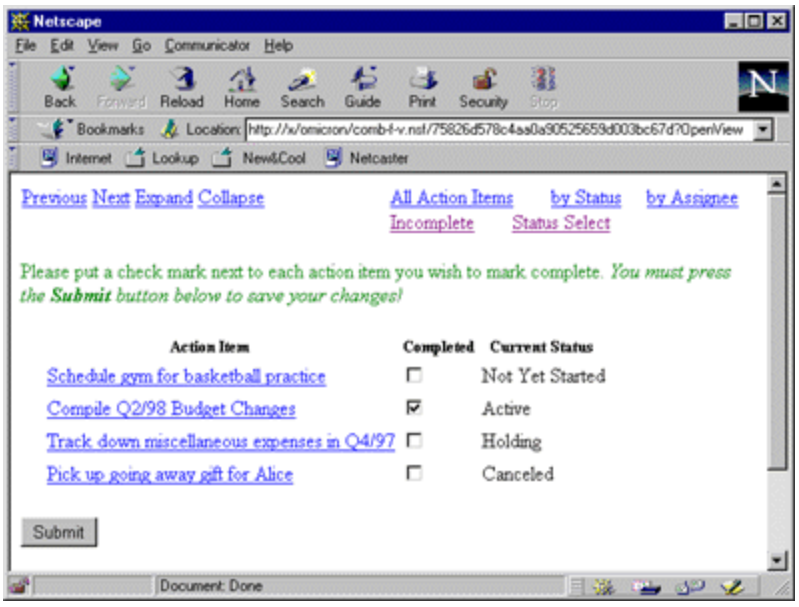
In the **first article** in this two-part series, we introduced you to the technique of combining forms with views by showing you how to add a quick-find search bar to the top of your Web views. To do this, we defined a Quick-search form by adding HTML to a $$ViewTemplate form. Domino displays the form and the view on the same Web page, so the user can simply enter a search phrase in the search field at the top of the view to do a quick search.

In this second article, we'll show you how to use the same techniques to create a view that allows users to select multiple documents and perform an action on them. We'll look at two ways to create the view -- one that uses checkboxes for the selection, and one that uses combo boxes (drop-down lists) to actually change data on the documents from the view. All the examples we'll describe are working examples you can try by **downloading the sample database** from the Sandbox.

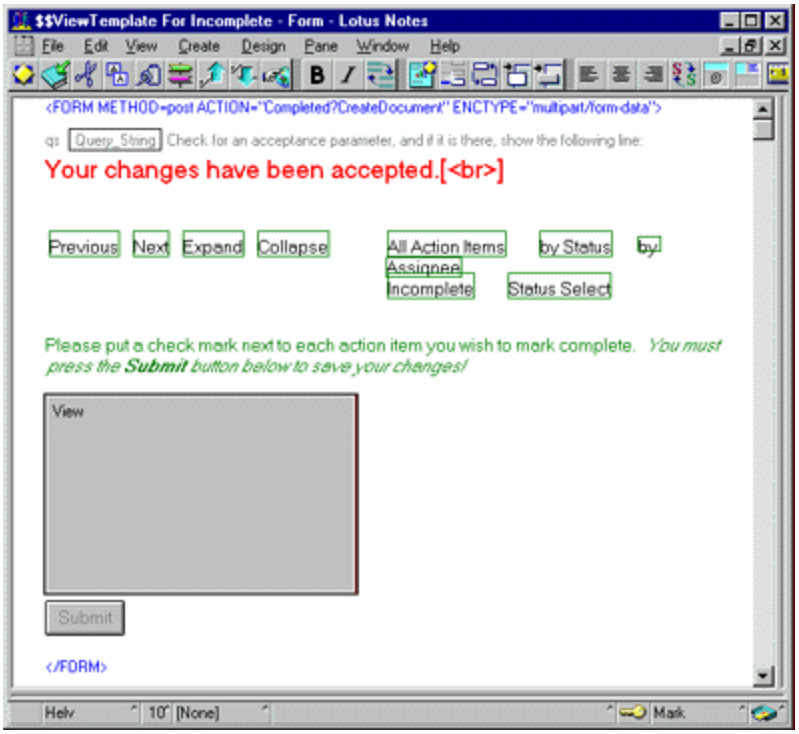
Selecting multiple documents from a Web view -- the checkbox method

As you may know, it is easy to define forms with checkboxes. You simply define a field as a checkbox and supply the allowable values, and Domino automatically generates the HTML when the form is accessed.

Getting the checkboxes in a view involves a different technique. Remember that forms and views don't exist on the Web. Domino simply displays pages, and a page can include one or more forms. So, like the Quick-search example in the **first article**, we can begin creating our Web view by using HTML to define a form. However, this time, the form does not begin and end above the body of the view, but rather *the body of the view is part of the form*. And one of the columns in the view calculates the HTML to define a checkbox for each document. Here is the view we'll create, which shows incomplete action items that the user can check off to mark them complete:



And, here is the "$$ViewTemplate for Incomplete" form behind the Action Items page shown above. Notice that we have embedded the view in the form.



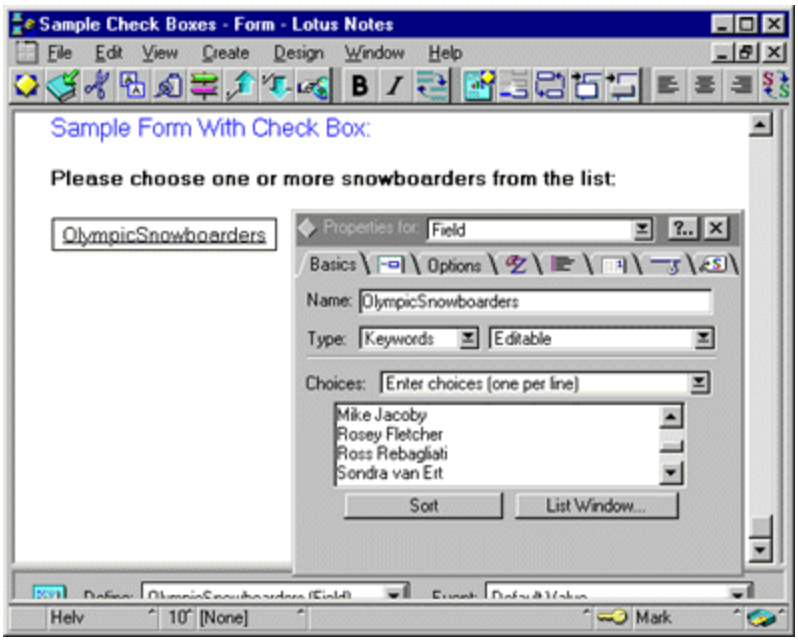
Note: When designing views like this one or the combo box view, you may not want to give users the standard Previous, Next, Expand, and Collapse hotspots like we have here. If they change a couple of the combo box values, but then click something other than the Submit button, their changes won't be processed.

Creating the Action Items checkbox view

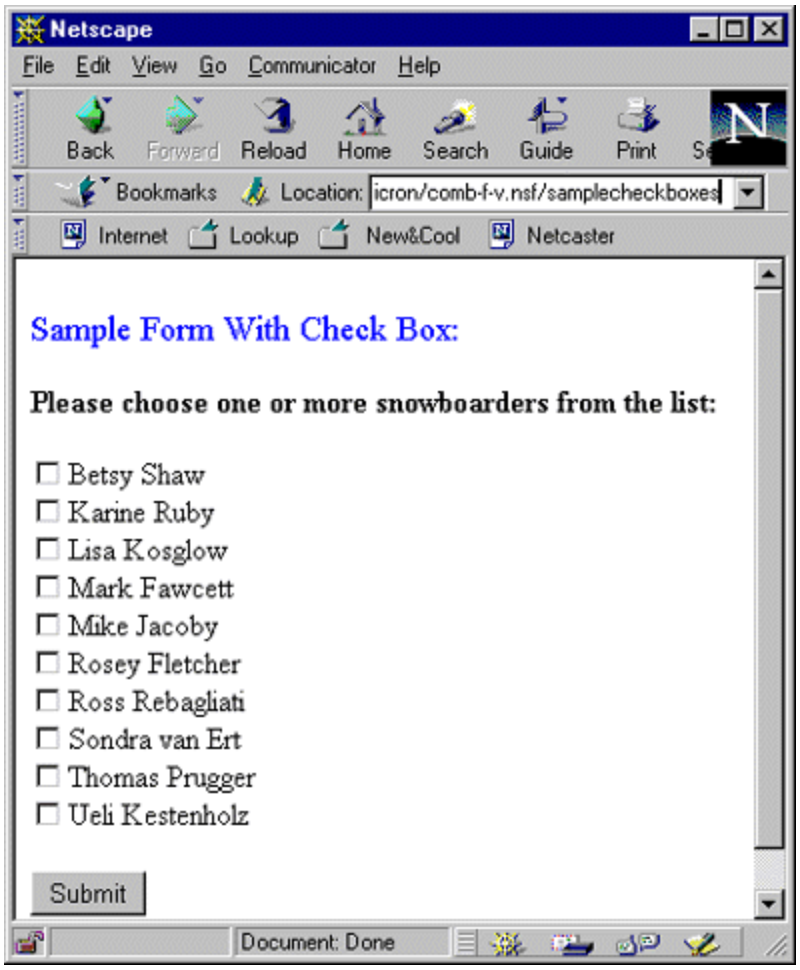
When Domino generates a Web page from a view, it generates on that page an HTML table containing one row for every document in a view, and one column with a link to each document in the view. In our checkbox view, we're not going to interfere with the table Domino generates to represent the view. What

we are going to do is use HTML to define a form that begins before the embedded view and ends after the view, so that the table containing the rows of data in the view is actually part of the form. Then, we just need to get the checkbox to appear next to each row of the view.

Let's first examine how checkboxes work on a Notes form, without any view considerations. Here is a simple Notes form with a checkbox on it:



And, here is how Domino displays the form on the Web:



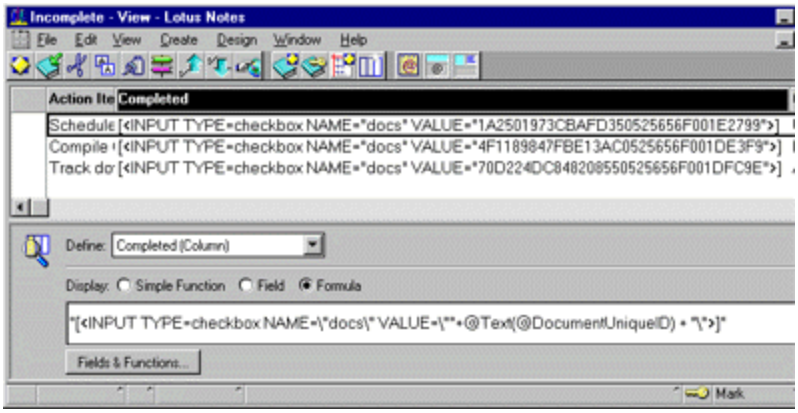
The HTML that defines a given checkbox has this format:

```
<INPUT TYPE=checkbox NAME="OlympicSnowboarders" VALUE="Karine Ruby">Karine Ruby<BR>
```

Each checkbox has both a display value and a stored value, although in this case they are the same.

Now going back to our Action Items view: each row in the view can contain a column defining a checkbox for choosing that row. That way, a checkbox appears next to each row. We don't need a display value for the checkbox, but we do want to store a value. Probably something unique to that document, like the document ID.

Here is the view definition for Action Items view, along with the formula for the checkbox column:



The formula for the Completed column is pure HTML to define a checkbox:

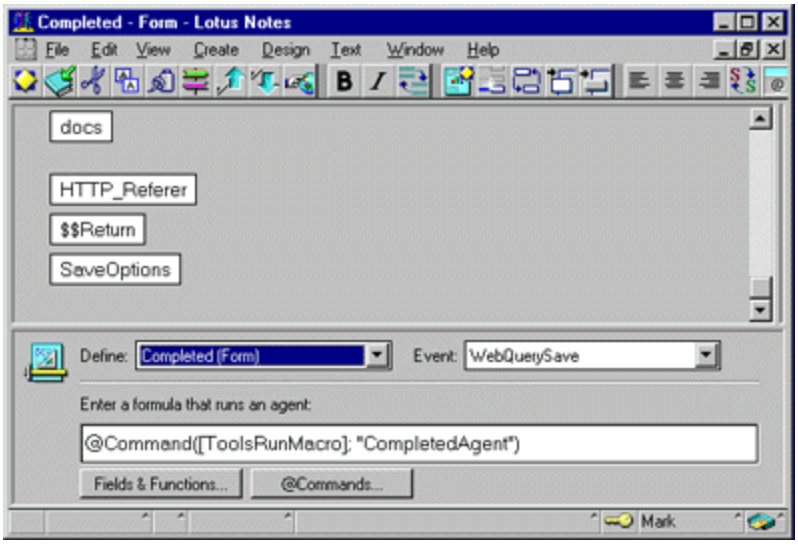
```
<INPUT TYPE=checkbox NAME=\"docs\" VALUE=\""+@Text(@DocumentUniqueID) + "\">
```

The name of the checkbox is the same for each row in the view: *docs*. When the checkbox is checked, the value to be stored in the docs field is the @DocumentUniqueID for that document. There is no display value.

So how do the document IDs of the selected documents get collected and processed? Because the docs field is defined in multiple HTML checkbox tags, it becomes a multi-value field, and it contains a list of document IDs. When the Submit button is clicked, the data stream containing the multi-value docs field is submitted as a new document, according to the following HTML defined at the top of "$$ViewTemplate for Incomplete" form:

```
<FORM METHOD=post ACTION="Completed?CreateDocument" ENCTYPE="multipart/form-data">
```

The Post Action says we want to submit what's entered on the Web form to the server and have it create a document using the Completed form. We used HTML to define a form around the body of our view, but we must also have a Domino form in the database that can accept the data sent to it by the browser. The Completed form needs one data field, *docs*:



The docs field matches the name of the checkbox field defined via HTML on the Web form, so the data will get stored in a Domino document via this form. If we were using a multiple-select view to do something like let customers browse a catalog view and check off items to purchase, then we would want to store a

purchase document with a list of the items to be purchased in a field like *docs*. In our case, we don't really want to store the document that collects the list of action items to be marked completed -- we just want to run a script that runs through the list of documents and marks them completed. Once we've done that, we can discard the document. The SaveOptions field in our form has a formula "0", so the document will not be saved.

The WebQuerySave agent contains the name of the agent that processes the selected action items. WebQuerySave agents run after all field formulas and just before a document is saved. In our case, since we set SaveOptions to "0", this agent will run just before the document is discarded. The name of the agent in this example is *CompletedAgent*, and it is written in LotusScript. Here is the Initialize event for the agent, which contains the main part of the code:

```
Sub Initialize
'Declare local scope variables
Dim s As NotesSession
Dim db As NotesDatabase
Dim ids As Variant
Dim completedDoc As NotesDocument 'Backend reference to the form "catching" the list of selected documents
Dim docReferedTo As NotesDocument 'Backend reference to the document that was selected

'Declare error handling for entire script
On Error Goto Errorhandler

'Set values
Set s = New NotesSession 'Used to capture the "in memory" document context
Set db = s.CurrentDatabase 'Used for agent logging

Set completedDoc = s.DocumentContext 'Sets reference to the doc object created in the server's memory
ids = completedDoc.docs 'Gets list of UNIDs from the multivalue field we referenced on $$ViewTemplate for Completed
For i = 0 To Ubound(ids) 'Loop through all the documents to have their status changed
Set docReferedTo = db.GetDocumentByUNID(ids(i)) 'Navigate the view to the appropriate doc via UNID
docReferedTo.ActionStatus = "Completed" 'Set the status, or do whatever you need to here
flag=docReferedTo.Save(0,0)
Next

Exit Sub 'Exit the sub before the errorhandler

ErrorHandler:
'Use the Err function to return the error number and the Error$ function to return the error message.
Print("Error Number " + Cstr(Err) + " : " + Error$ + " on line number " + Cstr(Erl))
Resume Next 'Continue where the agent encountered the last error
End Sub
```

Even if you have very little experience with LotusScript, you should find this script easy enough to modify for your purposes. Assuming you are simply changing a field value in each document in which the user has marked a checkbox, you need only to change the line marked in bold in the above script:

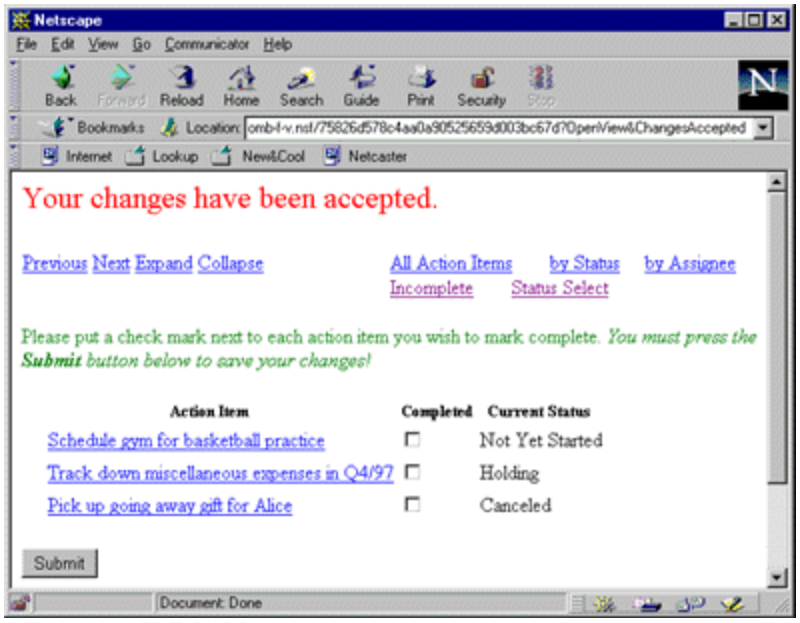
```
docReferedTo.ActionStatus = "Completed"
```

Everything else in the script should work for most any checkbox-in-view situation.

Letting the user know you've accepted the changes

Now for defining what to show the users after they submit their changes. There are a variety of options. We could simply display a page thanking them for their input and giving them some navigation links, or we could just return them to where they were. One approach that works well is to return the users to the same spot in the view where they left off, but show them a message at the top acknowledging their changes.

This is what we do in our Action Items checkbox view. After the user marks one or more of the action items completed, Domino returns the user to the same spot in the view, except that the completed action item no longer shows up in the view, and the message "Your changes have been accepted" appears at the top of the page:



We accomplish this using a combination of $$Return redirection and by passing a flag on the URL that re-opens the view. First, we redirect the user using the $$Return field on our StatusSelect form. Remember, even though the user never sees the StatusSelect form, it's the form Domino uses to record the submitted data, so our $$Return field goes there rather than on the $$ViewTemplate form. The $$Return field on StatusSelect has this formula:

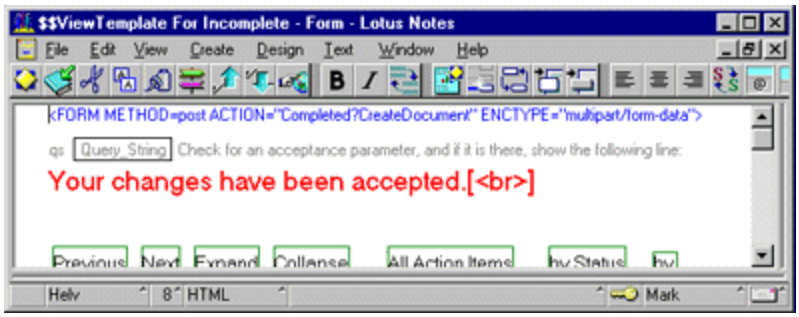
"[" + HTTP_Referer + "&ChangesAccepted]"

The brackets in any $$Return field tell Domino to redirect the user to the URL within the brackets. *HTTP_Referer* is a computed-for-display field defined before $$Return, which captures the CGI variable with the same name. (CGI variables are Internet-standard predefined variable names that Web browsers and servers support.) HTTP_Referer always contains the URL of the *referring page*. In our case, this is the URL of the view -- in fact, the exact position in the view, if the user had browsed through several pages of it before making changes -- where the changes were recorded.

If we redirected the users simply to HTTP_Referer, they would be returned to the view and would see the action items with their new status values. However, we are adding a custom parameter, *&ChangesAccepted*, to the end of the URL. We can "accept" this parameter in our $$ViewTemplate form.

For those of you who haven't seen this custom parameter technique before: you can add anything you like to a URL after an & sign -- provided you don't use any spaces. You can then parse the URL on the receiving page to look for the parameter or parameters you passed.

In our case, we look for the &ChangesAccepted flag and use its presence to trigger a hide-when formula on our $$ViewTemplate. Here is another look at the top of our "$$ViewTemplate for Incomplete" form. Look closely at the code in red and the line above it, with the Query_String field:



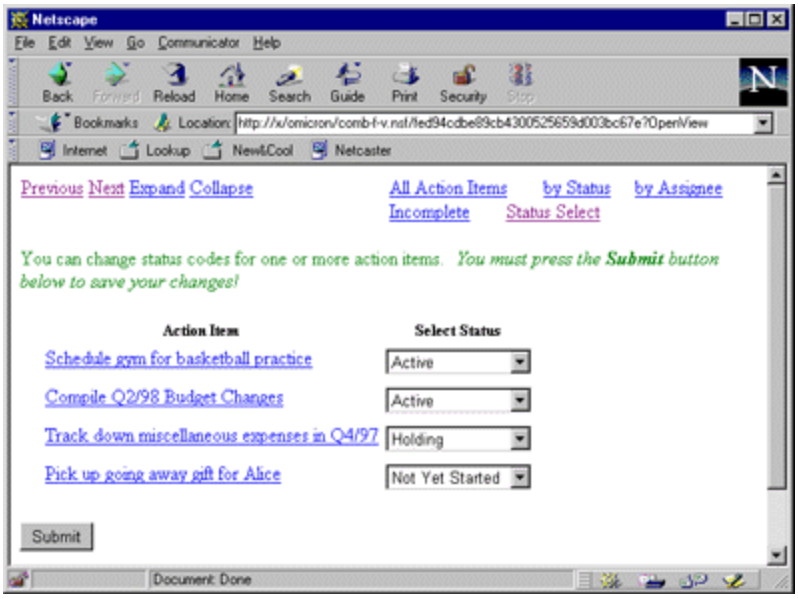
The Query_String field captures another CGI variable: the Query_String is everything after the ? mark in the current page's URL. Since we appended the string &ChangesAccepted to the view URL to re-open the view, it is part of the Query String. The hide-when formula for the line containing the static text "Your changes have been accepted" is:

!@Contains (Query_String; "ChangesAccepted")

So the message displays only if *ChangesAccepted* is part of the URL. Once the user clicks a Previous or Next button or changes views, the &ChangesAccepted parameter will be gone, because Domino will have generated a new URL without that parameter.

Selecting multiple documents from a Web view -- the combo box method

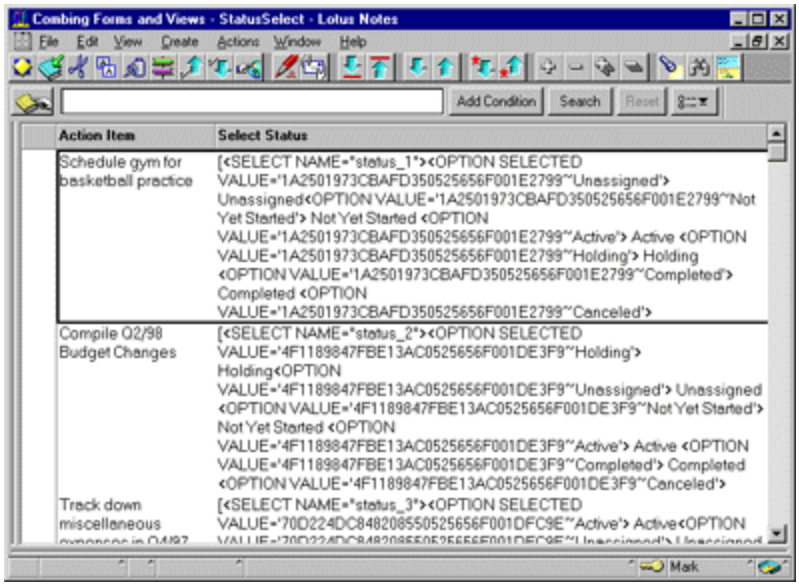
Now that you know how to put checkboxes in views, let's apply the same techniques to use combo boxes instead of checkboxes. Checkboxes give you a multiple-select capability, similar to what you have using a Notes client. Combo boxes give your Web users something not available to your Notes users: the ability to change values from a Web view *without opening the documents*. Here is an example:



Creating the Action Items combo box view

The approach is very similar to putting checkboxes in views, but there is one major difference: each combo box must have a separate field name. With the checkboxes, each row in the view contained the HTML for a checkbox with the field name *docs* and the value (if checked by the user) set to the document ID of the document. The collection of document IDs was, in effect, compiled by the browser and submitted as a single field *docs* with multiple values. Combo boxes are inherently different: each one contains only one value. So they need separate field names.

Our solution is to set up our StatusSelect combo box view to generate field names automatically. To do this, we use the field name *Status_X*, where *X* is the row within the view being displayed. Here is what the StatusSelect view looks like from a Notes client:



Notice the name of each combo box is Status_1, Status_2, and so on. And the values for each combo box -- Unassigned, Holding, Not Yet Started, Active, Completed, and Cancelled -- are specified as the document ID of the action item plus a suffix of ~Holding, ~Unassigned, ~Not Yet Started, and so on. This way, the document ID of each action item is submitted along with the new value of each one. This will provide a QuerySave agent the data it needs to assign each action item a new value.

Here is the column formula that generated the HTML in the "Select Status" column. Notice that for each action item, the current status is highlighted as the current value in the combo box, and the other remaining choices are listed as new choices:

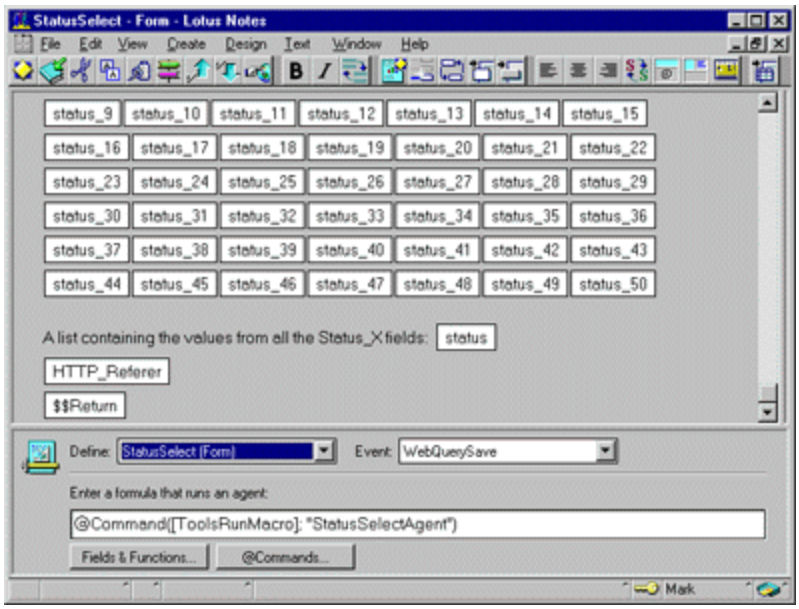
```
rem "The choices will be all the possible values except the current value";
statuschoices := @Trim(@ReplaceSubstring("Unassigned":"Not Yet
Started":"Active":"Holding":"Completed":"Canceled";actionstatus;NULL));

rem "Appending the @DocNumber -- the position of the document in the view -- gives each combo box a
different field name.";
firststring :="{<SELECT NAME=\"status_"+@DocNumber+"\">";

rem "Show the current status as the currently selected value in the combo box";
secondstring:= "<OPTION SELECTED VALUE=\'"+@Text(@DocumentUniqueID)+"~"+actionstatus+"\'> " +
actionstatus;

rem "Since StatusChoices is a multi-value variable, this OPTION VALUE statement will repeat the proper #
of times";
thirdstring:= "<OPTION VALUE=\'"+@Text(@DocumentUniqueID)+"~"+statuschoices+"\'> " +
statuschoices;

fourthstring := "</SELECT>}";

firststring + secondstring + @Implode(thirdstring) + fourthstring
```

Now that we've defined the view to generate HTML fields, we need to define the <FORM> area on the $$ViewTemplate form and then the corresponding form on the server to collect the data. The

"$$ViewTemplate for StatusSelect" form is easy -- it's identical to the "$$ViewTemplate for Incomplete" form, except for the form specified in the <Form> statement:

<FORM METHOD=post ACTION="StatusSelect?CreateDocument" ENCTYPE="multipart/form-data">

The back-end form, StatusSelect, however, is significantly different. Because we had to generate a separate HTML combo box (SELECT NAME) field with a separate field name for each action item in the view (Status_1, Status_2, Status_3, and so on), we also need corresponding field names on our StatusSelect form. And since we don't know how many rows might display in the view at any given time, we need to make our best guess and create a form with as many rows as we think we might need. Here is the StatusSelect form with 50 Status_X fields on it:



The Status_X fields are just editable text fields, and collect the data submitted by our Status Select page. If there are more than 50 rows showing at a time in the view, and you've only specified fields up to Status_50, your users will get an error (any time you submit data from an HTML form that Domino cannot match to a field on the corresponding server-based form, you will get the error message "Error 404 -- HTTP Web Server: Item Not Found Exception").

Your Domino administrator controls how many rows in a view are displayed in each Web page on a particular Domino server. The "Lines per view" setting is in the HTTP Server section of the Server document, and the default is 30. If the "Lines per view" is set too high, or you are not sure what the setting might be changed to in the future, you can override this setting to control how many rows your users will see within your application. Wherever you link users to that view, use an @URLOpen command instead of an @Command ([OpenView]) or a view link. With the @URLOpen command, you can add a parameter, *Count*, to specify how many rows to show in the view:

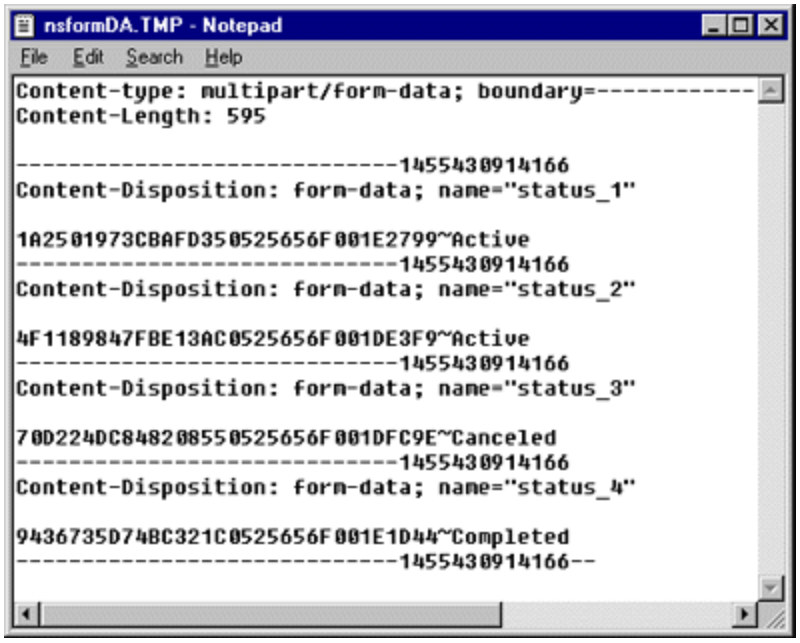
@URLOpen ("http://host/directory/database.nsf/StatusSelect?OpenView&Count=50")

If you use a view link or an @Command ([OpenView]), no Count parameter is included. Your Server document dictates how many rows are displayed on each page.

Now for processing our status changes. The approach is very similar to the approach we used for processing the checkbox view. For the checkbox method, the HTML document that was submitted by the user contained a single multiple-value checkbox field containing a list of document IDs to be marked completed. This example is a little different. What gets submitted here is a different field for each action

item -- Status_1, Status_2, Status_3, and so on. Each one contains the document ID of the action item followed by a tilde (~) character and the new status code.

An interesting way to observe what is really being sent to the server by the browser is to look at where your browser stores local copies of the files it submits. For example, Netscape stores them in your Windows\Temp directory by default. In our StatusSelect view, we had four action items. We changed the first two to *active* and submitted the form. Here is what the browser submitted to the server:



You'll notice that although we only changed the status for the first two items, it submitted data for all four action items. The second two had the prior value submitted again.

On our StatusSelect Domino form, we have a field called *Status*, which is computed to store a concatenated list of all the Status_X fields. This complete list of document IDs concatenated with status codes can be handed off to our script agent. The agent is almost identical to the agent we used to process completed action items in the checkbox example, except that it has to separate the document ID from the new action item to determine what value to set. Here is the core loop from the script where the values are set:

```
For i = 0 To Ubound(idsandvalues) 'Loop through the documents to have their status changed
ids = " "
newstatus = ""
stringidsandvalues = ""
stringidsandvalues = idsandvalues(i) ' idsandvalues put into a string
' the Ubound is not the number of used cells in an array --it is the total number of cells
' the first empty cell represents one cell after the last value in the array
If stringidsandvalues <> "" Then
ids = Left$(stringidsandvalues,32) 'the UNID parsed off of stringidsandvalues
Set docReferedTo = db.GetDocumentByUNID(ids) 'Navigate the view to the appropriate doc via UNID
' set newstatus to be the value of stringidsandvalues
newstatus = Mid$(stringidsandvalues,34) ' store the new status value
docReferedTo.ActionStatus = newstatus 'Set the status, or do whatever you need to here
flag=docReferedTo.Save(0,0)
End If

Next
```

The flaw in this processing is that we are setting the status code for *every* action item, not just the ones

that have changed, so all our action items are needlessly changing whenever the user changes any of them from this view. We could very easily change the script to first check the "new" value to see if it is the same as what's stored, and only update the document if the value has changed.

Conclusion

This two-part series has shown you how to use HTML to combine forms and views to add powerful features to your Domino Web applications. You saw in the first article how to add a search bar to the top of each view. This article has shown how to use the same techniques to add checkboxes and combo boxes to your views, allowing users to change multiple documents directly from the view. We hope you will find many uses for these techniques.

ABOUT ANDREW

Andrew Broyles is president and lead architect of Replica Inc., a Lotus Business Partner in Indianapolis, Indiana. Replica specializes in integrating disparate legacy and client-server data sources in Lotus Notes/Domino.

Our thanks to Steve DeVoll of **Workflow Designs** in Dallas for first introducing us to the technique of including checkboxes in a view and of using a hidden back-end form to collect the data. As far as we know, he invented the checkbox technique.