Domino and JavaScript: Dynamic Partners (Part II)

by David MacPhee, Karen Hobert and Russell Lipton

[Editor's note: This article resides in "Iris Today", the technical Webzine located on the http://www.notes.net Web site produced by Iris Associates, the developers of Domino/Notes.]

Using JavaScript for Input Validation in Domino

In our first article, we discussed the relationship among Domino, Notes and JavaScript, and described how Domino developers can use JavaScript to control the user environment within a Web browser. As powerful but conflicting **agendas** for extending HTML muddy the water for developers, JavaScript implementations are actually converging somewhat. This ensures that JavaScript's importance to Notes developers will grow, not diminish, over the coming six to 12 months.

In this article, based on the Domino.Applications SiteCreator application, we explain how JavaScript handles the sticky issue of managing field validation with the least discomfort for the user. We will be using Domino 4.5, Domino.Applications SiteCreator, Netscape Navigator 3.01+ and Microsoft Internet Explorer 3.01+ for discussion purposes.

Field validation in Domino

Field validation is one of the more difficult features to implement in Web-based applications. In Domino, you can write standard Notes conditions for each field validation routine. However, the process for evaluating and warning the user of problems is less than user-friendly (although we were left with no alternatives when handling Internet Explorer checkbox validation).

The Domino field validation process requires a server hit to run the formula. When a document is submitted, Domino evaluates a field validation formula and returns a new page that states a problem, if one existed, with processing a specific field. The user must then return to the original form (using the "Back" button), often to find that all the data that he/she input has disappeared! This becomes quite frustrating when there are many fields to validate.

Another approach is to create a single computed field or QuerySave agent to run all field validations at once and return a single page with information on all the missed fields. However, the user's short-term memory must be top-notch to remember everything that was missed.

To further complicate matters, we found that working with the user-entered/selected values varies based on the type of browser. While the JavaScript object model is similar in both Netcape Navigator and Microsoft Internet Explorer, the two browsers handle entered values differently, specifically when it comes to values selected in checkbox type fields.

To code for both browsers as well as improve the user's experience, we implemented a way to use JavaScript to perform the field validation on the Web client, alert the user about a problem field and return the cursor to the field in question, while maintaining the current page in the browser window. This design reduces server hits, provides rapid feedback to the user and retains the information the user already has entered.

In the Domino.Applications SiteCreator, we accomplished this by creating some generic JavaScript function calls that are called prior to submission of the Web document to the Domino server. We use basic JavaScript Alert methods to prompt users about problem fields and maintain the Web document in the main browser window.

Testing for browser type

While most users have access to a reasonably current browser, it is prudent to test for browser type and version when using JavaScript in Domino applications. In our case, there are enough differences between JavaScript implementations to make this critical.

The following example tests for the browser type and version. In subsequent articles, you will understand why we need to be so specific about this information. For field validation, we only need the browser type to determine whether Netscape Navigator or Microsoft's Internet Explorer is active.

If the browser supports JavaScript (and has it enabled), we launch a new browser window defined by a Notes document that contains the frameset definition as well as all browser-specific JavaScript functions to handle the processing of the Web SiteCreator. Non-supported or un-enabled browsers are left on the About document for the database, which explains the requirements of the application. This way, the user must have the proper environment enabled to enter the application. The following JavaScript code is embedded in the SiteCreator "About This Database" document.

```
//if an unsupported JavaScript browser is encountered, inform them of our requirements...
function nogo() {
        document.write ( "<FONT SIZE=4>Note: To use this version of SiteCreator, you must use Netscape Navigator
3.01+ or Microsoft Internet Explorer 3.01+." );
}

//redirect the browser to its specific code
function go() {
        href = location.href.toLowerCase();  //compare against lowercase href for ".nsf"
        pos=href.lastIndexOf ('.nsf', href.length-1);
        dbPath=href.substring  (0, pos+4);
        winopts = "toolbar=no,location=no,directories=no,status=no,menubar=no,
scrollbars=yes,resizable=yes";

        //we currently only differentiate between Netscape Navigator & Internet Explorer browsers, but here we could do
version-specific things
        view = ( ver == "nn3" || ver == "nn4" ) ? "/$nn/frameset?OpenDocument&A$nn" :
"/$ie/frameset?OpenDocument&A$ie";
        //open a new window
        imgWindow=window.open( dbPath + view,"Frameset", winopts);
}

//check for our JavaScript-supported browsers
if ( ver == "nn3" || ver == "nn4" || ver == "ie301" || ver == "ie302" || ver == "ie4" ) {
        go();
        location= "/cntlcr.nsf"
} else {
        nogo();
}
//-->
</SCRIPT>
```

An overview of the JavaScript functions

We developed several types of JavaScript functions to help us perform the input validation on the Web client. The functions fall into several categories, including:

- Conditional functions
- Get value functions
- Utility functions
- Alert box and field focus functions

We will discuss these different types of functions in depth later. These functions test the field values without polling the server.

Domino's support of pass-through HTML enables us to embed JavaScript in forms and computed subforms that are rendered as users create and edit various documents via a Web client.

More specifically, we replaced the Domino submit button on each form with a submit button of our own. Then, we embedded in the form a JavaScript function, submitDocument, that controls the document submit. The submitDocument function includes the input validation function calls to test the field values, as shown below:

```
<script language="JavaScript">
function submitDocument() {
    form = document.forms[0];
    if ( parent.failNull (
            form.ProfileApprovers1, "You must specify one or more approvers for the first approval step","text"))
            return;
    if ( parent.failContains(
            form.EditProfileFormKeyOverride_1, "N", "You must select Yes to continue.", "radio") )
            return;
    if ( parent.failContains( form.EditProfileFormKeyLabel_4, ',;:',"Labels/questions cannot contain commas, colons, or semicolons.", "text", "mult"))
            return;
    form.submit();
}
</script>
<!-- -->
<input type="button" name="Submit" onClick="SubmitDocument()">
```

If any of the JavaScript input validation routines evaluate to "true", the submit operation is halted. We alert the user to the problem field and focus the cursor in the problem field on the document. If the input validation test passes, we submit the document to the Domino server and continue as usual.

JavaScript functions

As stated above, several types of functions are needed to perform the field validation on the Web client. Some of the functions do the actual testing; some are required to manipulate the browser and still others are required to accommodate the differences in behavior between Netscape Navigator and Internet Explorer.

The conditional functions are generic tests that are performed against the field value to ensure that the data meets the validation criteria. These functions are typically embedded in JavaScript "if" clauses. The basic tests are:

Function	***Returns***	***Syntax***
failNull	True if the field value is Null	functionname (field object,error message, input data type)
failContains	True if the field value contains a specified value	functionname (field object, test value, error message, input data type, *multi-values*) where the multi-values argument, "multi", is optional. This is useful when trying to test for multiple values
failNotContains	True if the field value does not contain a specified value	functionname (field object, test value, error message, input data type)

These functions return either a Boolean True or False value so that we can stop the submit. If the function is True, or if a validation failed, we stop the submit, return a JavaScript alert with a specific failure message, and focus the cursor on the field that failed the validation test. Here is the failNull syntax:

```
if(parent.failNull(form.ProfileApprovers1,"You must specify one or more approvers for the first approval step","text"))
return;
```

The JavaScript function for failNull follows:

```
function failNull ( vField, vMessage, vType) {
    //this function will stop the submit if a field is null or contains all spaces
    //get the field value...
    theValue = getFieldValue (vField, vType);
    //if the field value is null, we fail and return true...
    if ( theValue == "" ) {
    alertBox ( vField, vMessage, vType );
    return ( true );
}
//remove any spaces from the value
trimField = trimBlanks( theValue );
//if the field value is all spaces, fail and return true...
```

```
if ( trimField =="" ) {
    alertBox ( vField, vMessage, vType );
    return ( true );
}
//otherwise continue...
return ( false );
}
```

Here is the syntax for the failContains logic:

```
if (parent.failContains( form.EditProfileFormKeyOverride_1, "N", "You must select Yes to continue.", "radio") ) return;
if (parent.failContains( form.EditProfileFormKeyLabel_4, ',;:',"Labels/questions cannot contain commas, colons, or semicolons.", "text", "mult"))  return;
```

The JavaScript failContains function looks like this:

```
function failContains( vField, vValue, vMessage, vType )  {
    //this function will stop the submit if a field contains a specified value (or value list)
    //get the field value...
    theValue = getFieldValue (vField, vType);
    //check argument list to see if we are testing multiple characters...
    var count = ( failContains.arguments.length == 6 ) ? vValue.length-1 : 0;
    var value = ( failContains.arguments.length == 6 ) ? vValue.substring(0,1) : vValue;
    //fail the submit if the field contains the value(s)...
    for ( i = 0; i <= count; i++)  {
        if ( theValue.indexOf(value) > -1) {
            alertBox ( vField, vMessage, vType );
            return (true);
        }
        value = ( count > 0 ) ?  vValue.substring(i+1,i+2) : vValue;
    }
    //otherwise continue...
    return (false)
}
```

The functions illustrated above call three subroutines: getFieldValue, trimBlanks and alertBox. The getFieldValue function retrieves the actual value the user entered or selected in the field being tested. To further explore the getFieldValue function, we need to consider input data types and browser types.

Input data types

There are five basic input data types that correspond with the different input methods on an HTML page. In Domino, these input data types map to the following field types on Notes forms:

HTML Input Data Types	*Notes Field Types*
Text	Text, Number, Date
Text Area	Rich Text
Radio Button	Keyword Radio Button
Checkbox	Keyword Checkbox
Select	Keyword Text List

What is important is that we must be able to use JavaScript to read the HTML page on the client and get the data value that the user entered or selected in a specific field. This is necessary so that we can test the value to make sure it meets the validation criteria. The JavaScript object model makes it easy for us to identify a field by its name on a page. However, the best way to retrieve the value of that field varies based on the field's input data type.

There are some differences between the way that Netscape Navigator and Microsoft's Internet Explorer handle the functionality of the input types. This is especially true in the case of checkbox values.

To get the field value in JavaScript, we examine the JavaScript Value property of the field. Of course, we found that while getting a text value was relatively simple in JavaScript, other input data types, such as the ones below, were not retrieved quite so easily. The main issue here is that the value property returned by the JavaScript isn't always the same value entered by the user.

Below is a table of the input data types and what the JavaScript value property returns for each:

HTML Input Data Types	*Stores As*
Text	User-entered value as a string
Text Area	User-entered value as a string array of one element
Radio Button	Array of selection pointers to an array of strings representing the choices
Checkbox	In Netscape Navigator: if more than one keyword is specified: array of selection pointers to an array of strings representing the keywords, with the CHECKED property indicating a selected keyword. if only a single keyword is specified: Keyword value as a string, with the CHECKED property indicating a selected keyword In Microsoft Internet Explorer: Apparently an array, but individual values unavailable via JavaScript
Select	Array of selection pointers to an array of strings representing the choices

Before field values could be tested, routines were created to retrieve the user-entered values. Our approach was to feed the input validation function of the field data type to the test routine when it was called. Our test routine would then call the function getFieldValue to determine the user-entered or selected value based on the input data type. The following is an example using the failNull validation function call for a "text" input field.

```
parent.failNull(form,"ProfileApprovers1","You must specify one or more approvers for the first approval step","text")
```

Retrieving the field values and handling Netscape Navigator and Internet Explorer differences

The getFieldValue function is the heart of our validation implementation. It is the common code called from all validation functions to return the data to be validated. It is also the code that handles the object model differences between the browsers.

While much of the object model overlaps between the two languages, there are some discrepancies and differences when trying to manipulate each browser. As discussed earlier, we retrieve the browser type and version when the Web client launches the SiteCreator database. In this example, for clarity, the test is repeated in the checkbox logic.

```
function getFieldValue ( theField, vType ) {
    //this function will return the field value (or value list) based on the element type
    theValue = "";
    sep = "";
    hits = 0;
    //text is the user-entered value as a string
    if ( vType == "text" ) return ( theField.value );
    //textarea is the user-entered value as a string array of one element
    if ( vType == "textarea" ) return ( theField[0].value );
    //checkboxes & radio buttons are not so simple
    if ( vType == "checkbox" || vType == "radio" ) {
        if ( theField.value == null ) {
            //if we're here, we are validating a radio button or a nn multi-element checkbox
            for ( i = 0; i < theField.length; i++ ) {
                if ( theField[i].checked ) {
                    hits++;
                    if ( hits > 1 ) {
                        sep = "; ";
                    }
                    theValue += sep + theField[i].value;
                }
            }
        }
```

```
                return ( theValue );


        } else {
                //if we are here, must be an ie checkbox, or nn with a one-element checkbox")
                if ( navigator.appName == "Microsoft Internet Explorer" ) {
                        //ie. return some data so we can validate on the server;
                        return ("can't validate on client")
                }
                //nn one-element checkbox, see if its checked ...
                if (theField.checked ) {
                        return ( theField.value );
                } else {
                        return ( "" );
                }
        }
        //select is an array of selection pointers to an array of strings representing the choices
        if ( vType == "select" ) {
                for ( i = 0; i < theField.options.length; i++ ) {
                        if ( theField.options[i].selected ) theValue += theField.options[i].text
                }
                return ( theValue );
        }
}
```

Coping with checkboxes

Working with checkbox input data became the most frustrating aspect of our work. We suspect that both Netscape Navigator and Microsoft's Internet Explorer have bugs in this area.

Netscape Navigator's approach is very strange. Navigator does not present the checkbox object as an array if there is only one keyword available (unlike the radio button, which is always an array) but instead treats it as a string with a CHECKED property. To compensate, we have coded the getFieldValue function to perform the necessary tests to return the proper value.

In Internet Explorer, the array of selected values is not available for interrogation. Since the element array is not available from Internet Explorer, we are forced in this single case to hit the server for validation. To do this, and simulate our validation method, we place the following in the Input Validation formula in the Notes field:

```
msgText:= "You must choose at least one area for your site."; <br>
msgText2:="To create a site, you must select the Home Page Area."; <br>
prefix:="<BODY BGCOLOR=#ffffff><SCRIPT language=java Script> ;alert(\' "; <br>
```

```
postfix:=" \'); parent.sessionIndex--;history.go(-1);</SCRIPT>"; <br>

errMsg:=prefix+msgText+postfix; <br>

errMsg2:=prefix+msgText2+postfix; <br>

@If( ProfileSiteAreas = ""; @Failure(errMsg); @IsNotMember("Home Page Area"; ProfileSiteAreas);
@Failure(errMsg2); @Success)
```

This will perform the two required validation tests on the field. A failure on either test will send a small bit of JavaScript code back to the browser (via passthru) to issue a similar JavaScript error prompt, and then by referencing the browser history object, return the page containing the error to the user.

Cleaning up strings for Domino

The trimBlanks function is a utility routine that simply takes the field value and trims any extra space values. The JavaScript function follows:

```
function trimBlanks ( theString, repChar )  {

    //this function replaces the spaces in a string with the provided character

    trimString="";

    for ( i = 0; i < theString.length; i++)  {

            theChar=theString.substring ( i, i+1);

            if ( theChar == " ") {

                    trimString += repChar

            } else {

                    trimString+=theString.substring ( i, i+1);

            }

    }

    return (trimString);

}
```

We use this routine in the failNull function to check for all spaces, but in addition, the trimBlanks routine allows you to specify a replacement delimiter. This means that it can be called anytime data must be passed to the server in the URL format using "+" characters in place of spaces:

```
parent.trimBlanks( data, "+")
```

Since this function is stored in the parent frame document, it is available anytime during a session. It can also be used to properly format data that Domino will "un-format" upon receipt.

Prompts and getting focus

In addition to testing the field value, we needed to create JavaScript alert box and field focus functions to force the user's cursor into the field in question.

This relatively simple alertBox function presents the user with a specified message and places the cursor back into the field that failed the validation.

```
function alertBox (vField, vMessage, vType ) {

    //this function replaces the spaces in a string with the provided character
```

```
        //if we pass a null message, we are doing a multiple validation test.  return so we can check other conditions.
        if ( vMessage == "null" ) return;

        //otherwise, display the error message
        alert ( "Field Contains Incorrect Value:\n\n" + vMessage )

        //and set focus (plus select the text if a text element)

        if ( vType == "text" ) {

        vField.focus();

        vField.select();

        } else {

        vField.focus();

        }

        return;

    }
```

Conclusion

You may recall from the first article that much of our JavaScript strategy is oriented to giving users a familiar, convenient, "Web-like" interface experience as well as to reduce Notes coding and server hits.

By taking advantage of the pass-through HTML capabilities of Domino, we have been able to define and call JavaScript routines that perform client-side field validation on the Web browser. The JavaScript functions and routines have given us an easy-to-maintain code base for testing user-entered values and returning a message prompt, if necessary. The outcome is a rapid, user-friendly interface that uses the Web client to process the validation tests.

The base JavaScript functions are maintained in the browser's parent frame document. As a result, they are available to any document we present to the Web client. This means that only the validation function calls are required for any form needing field validation tests. The code is easy to maintain and can be extended to test other types of conditions.

In the next article, we will apply these types of techniques and other JavaScript options to make the Address Book visible on a browser and available for modification from the Web, while retaining Notes security.

ABOUT DAVID AND KAREN

David MacPhee and Karen Hobert of BadDogBad are long-time Notes developers who have worked closely with Lotus to design and build Web interfaces for Domino.Applications. Bad Dog, BAD! is a Lotus Business Partner that has developed the SiteCreator Web interface for every release of Domino.Action, Domino.Merchant Domino.Broadcast and Domino Intranet Starter Pack. They were a major contributor on the Domino 4.6 WebAdmin Tool and every release of the RealTime Notes product. You can reach the dogs at webdog@baddogbad.com or visit their site at **http://www.baddogbad.com**.