A Hyperion White Paper

Custom Javascript In Planning
Creative ways to provide custom Web forms

This paper describes several of the methods that can be used to tailor Hyperion Planning Web forms. Hyperion has enabled Custom JavaScript within Hyperion Planning to allow for easy customization and validation of Web forms. This paper also describes tools that can be used, explains some basic JavaScript syntax, and provides some examples. The reader should have a general knowledge of Hyperion Planning and Web development basics.

Tools to use

A text editor. A good text editor is crucial for writing and debugging JavaScript. There are several freeware and shareware text editors that include syntax highlighting, parentheses matching features, and code completion (see the Resources section at the end of this document for links to text editors).

The following code shows the differences between Crimson Editor and Notepad.

Crimson Editor:

```
function limitPrecision(value, limit) {
    if ((value == null) || (value.length == 0) || (value == missing)) {
        return(value);
    } else {
        return((Math.round(value * Math.pow(10,limit)))/Math.pow(10,limit));
    }
}
```

Notepad:

```
function limitPrecision(value, limit) {
        if ((value == null) || (value.length == 0) || (value == missing)) {
                return(value);
        } else {
                return((Math.round(value * Math.pow(10,limit)))/Math.pow(10,limit));
        }
}
```

In both examples, the code is exactly the same but the display makes the first easier to interpret than the second.

A Web browser. Because Internet Explorer does not have a built-in JavaScript console and FireFox is a bit too sensitive, Mozilla Suite is the recommended Web browser for JavaScript development. When you implement your custom script, you could possibly have errors in your code. Mozilla has a built-in JavaScript console that displays error messages with line numbers. Clicking on an error message takes you to the line of code that is in error.
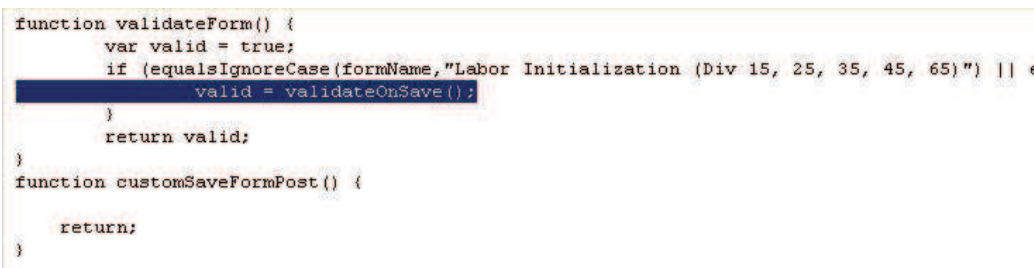
To Launch the JavaScript Console in Mozilla, select **Tools > Web Development > JavaScript Console**.
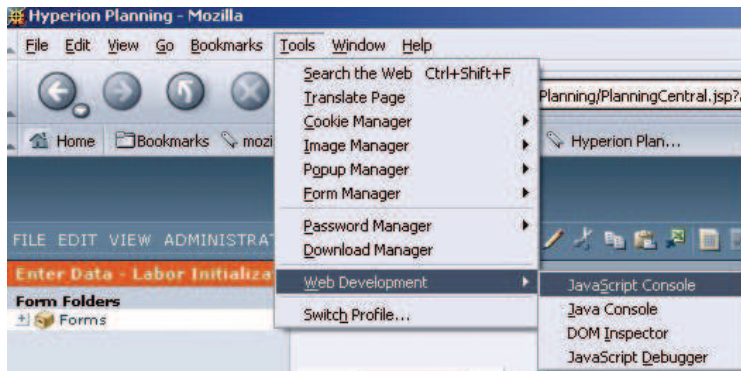
After the JavaScript Console is launched, you can test your code for errors. In the following example, an error message is displayed that the function $validateOnSave$ has not been defined. This message could be the result of a typo or similar error.
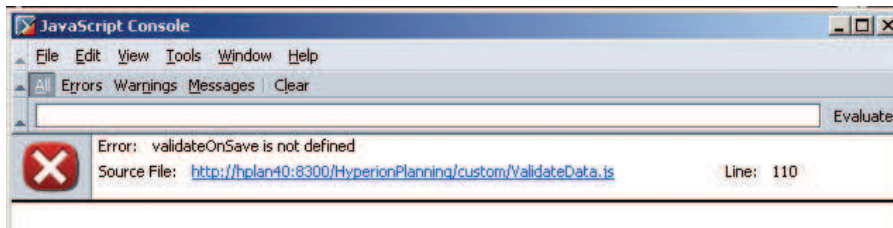
Double-clicking on the error message launches a source browser and highlights the line that is in error.

```
function validateForm() {
        var valid = true;
        if (equalsIgnoreCase(formName,"Labor Initialization (Div 15, 25, 35, 45, 65)") || e
                valid = validateOnSave();
        }
        return valid;
}
function customSaveFormPost() {

    return;
}
```

A good development environment. To test your code, it is often necessary to stop or start the Web server. In a production environment, however, this may be difficult to do.

SampleValidateData.js and ValidateData.js (located in the Custom directory of your Web application server). These files contain the full suite of available functions and samples. You need to put all of your custom code in ValidateData.js so Planning to read it.

Devlopment procedures

The first step in developing custom JavaScript is to determine what you are trying to do and when it should happen. The samples provided with Planning include the following events that you can capture:

- Loading a form
- Clicking save
- Selecting a cell
- Leaving a cell

The samples also include variations on those events as well as additional helper functions that you can call during those events.

An optional second step is to use the native alert function in JavaScript, which displays a message box. The syntax is displayed in the following example:

alert("'Test Message'");

This displays a message box with the text "Test Message." Adding this code where you want your custom code to execute enables you to see the timing and frequency of your customization. The message also provides notification that your code is working. If the message box is not displayed, then your code may not be configured correctly or you may need to stop and restart the Web server.

After you have determined that your test message displays correctly and with the correct frequency, you can develop your code further.

Javascript basics

For those new to JavaScript, there are many tutorials and examples available on the Web (see the Resources section for links to online resources), which you can use while developing your code. However, this section provides a few basic principles.

Building Blocks
JavaScript is organized into objects, properties, methods, functions, and events.

Objects are the building blocks of JavaScript. A Web page is an object, commonly referred to as a *document*. In Planning, objects include forms, grids, columns, rows, and cells.

Properties are the values associated with an object. In Planning, a common property is cell.value, where *cell* is the object, and *value* is the property.

Methods are the actions that you can perform on objects. For example, document.writeln combines the method *writeln* with the object *document*, and tells the Web browser to insert content into the Web page followed with a carriage return. The following sample is from SampleValidateData.js:

document.writeln('<table cellspacing=0 cellpadding=0 border=0><tr><td>');[1]

Events trigger a function. For example, when a user clicks the Save button, that event calls the function validateForm.

Functions do the work that you are trying to accomplish. Functions have the following syntax:

```
function MyFunction (Variable) {
        commands;
}
```

In the preceding example, the function *MyFunction* takes the parameter of *Variable*. A variable is not required, although the parentheses are. The following examples are from Planning:

```
function customCellValidatePost(row, col, cell) {
return;
}
function customSaveFormPost() {
return(true);
}
```

The function customCellValidatePost takes the parameters (or *variables*) of row, column, and cell. The function customSaveFormPost does not have any parameters.

All functions begin and end with braces { }.

Conditional Logic

JavaScript includes an If statement for conditional tests. The syntax for If statements is fairly straightforward, although it can become complex when there are nested conditions and many sets of parentheses and braces. The following example shows the basic syntax for an If test:

```
If (a question that returns true or false)
        {Do something for True}
Else
        {Do something for False}
```

The following is an example of a Planning-related If statement:

```
if (equalsIgnoreCase(formName,"Labor"))
{valid = validateOnSave();}
```

The preceding example calls the function equalsIgnoreCase as a conditional test. If the current form name is "Labor" or the form name is "labor," then the function validateOnSave is executed. This example does not have an *else* case. The next example shows an If statement with multiple commands:

```
if (val < 0) {
alert("Negative values are not allowed.");
validated = false;
cancelDrag();
selectLocation(row,col);
}
```

The preceding example does several things when a user enters a value less than zero. First, a message box provides an alert that the value is incorrect. Second, a variable called *validated* is set to False, alerting the calling function ValidateOnSave that the validation has failed. Next, the function cancelDrag is executed (a built-in Planning function that deselects the range of cells the user has selected). Last, the offending cell is selected.

Operators

If statements can contain mathematical comparisons. The available comparison operators are <, >, !=, ==, <=, >= , to represent less than, greater than, not equal to, equal to, less or equal to, and greater or equal to. When testing for an exact value, it is important that you use the "==" or double equal operator. A single operator is an assignment. For example, Val=1 sets the variable Val equal to one, but Val==1 tests to see if the variable Val is equal to one.

Using a double plus sign (++) allows for incrementing, as in the following example:

```
for (r=startRow; r<numberGridRows; r++) {
```

The variable r is set to the number of the starting row of a grid (zero in this example), and while r is less than the total number of rows in the grid, r will be incremented by one. This example is part of a for loop statement that enables you to execute code against each row of the grid from start to end. A for loop is a set of commands to execute one or more times. In the preceding example, the code is executed from the starting row until it reaches the number of rows in the grid. The initial variable r is set to the value of the starting row (1, for example) and is incremented by one until it no longer meets the condition of being less than the number of rows in the grid.

You can use a single plus sign (+) to concatenate values and strings. For example, you can cause a message to display that includes the value less than zero that a user entered in a Web form:

```
if (val < 0) {
alert("Negative values are not allowed. You entered: " + val);
}
```

When the code in the preceding example is executed, a warning message is displayed that notifies the user of an invalid entry and displays the invalid data.

Putting it all together

The following example demonstrates validating a form. The purpose of this block of code is to check that a user has accurately entered data across all (approximately 30) input cells.

Section 1: Validating the Form

```
function validateForm() {
var valid = true;
if (equalsIgnoreCase(formName,"Labor1") ||
equalsIgnoreCase(formName,"Labor2")) {
        valid = validateVariance();
}
return valid;
}
```

The function validateForm() is a native Planning function, and this section of code is executed after the user clicks Save but before the data is actually sent to the database. This code is capable of canceling the Save operation.

The first step in this section is to create a variable called valid and assign to it a value of *true*. This creates a default value and will remain *true* unless set to a different value. Next, the code tests to see if the form name is either Labor1 or Labor2—the double pipe || characters indicate *or*. If the test is *true*, that is, the form is named Labor1 or Labor2, then it calls the function validateVariance and checks for its value, reassigning the value of validateVariance to the variable valid. Last, it returns the value of valid to Planning. If the value of valid is *true*, the save continues (along with business rules set to run on save). If the value of valid is *false* and then the save is canceled—the data will not be saved to the database unless it is valid.

Section 2: Checking the Grid
The validateForm code calls the function validateVariance, which is where the real work is done, to validate the data entered by the user. You can name this function anything that makes logical sense. For example, you might choose to name it validateTotal, or something similarly descriptive.

```
function validateVariance() {
        var validated = true;
var cellValid = 0;
        for (r=startRow; r<numberGridRows; r++) {
                for (c=startCol; c<numberGridCols; c++)
{
                        var cell = getCellVal(r,c);
                        if (isFinite(parseFloat(cell))) {
                                cellValid =
limitPrecision(cellValid + getCellVal(r,c),3);
                        }
                }
        }
        if (cellValid!=1) {
                validated = false;
                cellValid = limitPrecision(cellValid * 100,2);
                alert("The numbers you have entered do
not equal 100%
\nYour Total Percentage is: " + cellValid + " %");
        }
        return(validated);
}
```

This section of code begins with setting a new variable, validated, to *true* and initializing the numeric value cellValid. Next, it begins two for loops to iterate through each column and row in the grid. Then, it assigns the value from the current data cell in the grid to the variable cell. The next step is to find out if the cell contains a data value—if it's null or missing, it shouldn't be included in the total. This test is performed with the JavaScript function isFinite()[2], which determines if the value is a finite number. A null value won't pass here. Next, we increment our initialized value cellvalid with the contents of the finite number.

This process also calls a follow-on function (LimitPrecision) that only takes the first three significant digits of the number, because there are times when including many digits of precision can produce undesirable results with JavaScript. Finally, after looping through every cell on the grid and adding all of the numbers together, the code tests the results.

In this instance, the code checks if the variable cellValid is not equal to a value of *one*, which would indicate data was over- or under-allocated. If cellValid is not equal to a value of *one*, then the variable validated is set to *false*. Additionally, a message is displayed to the user that the input is not valid and indicates their total entry amount.[3] Finally, the value of *false* is passed all the way back through each of the functions until it is returned to validateOnSave, which cancels the Save. No further actions are taken.

Section 3: Limiting Precision
The last part of the code relates to limiting the precision of the numbers used in the calculation.

```
function limitPrecision(value, limit) {
        if ((value == null) || (value.length == 0) || (value
== missing)) {
                return(value);
        } else {
                return((Math.round(value *
Math.pow(10,limit)))/Math.pow(10,limit));
        }
}
```

This example uses the limitPrecision function, which takes the parameters *value* and *limit*. *Value* is the value to be limited and *limit* is the number of significant digits to store.

The full code example uses both two and three digits for different reasons. Two native JavaScript math functions, round and pow, are used to round the requested value to the requested limit. The pow function is used in this example to raise 10 to the power specified as the limit (10^2 or 10^3 – 100 and 1000 respectively), which allows for the rounding to occur with less precision.

For example:

```
Math.round(121.5 * 100) / 100
```

moves the decimal point prior to rounding, and then moves it back.

Section 4: All of the Code

All of the code is put together and ready for implementation.

```
function validateForm() {
        var valid = true;
        if (equalsIgnoreCase(formName,"Labor1") || equalsIgnoreCase(formName,"Labor2")) {
                valid = validateVariance();
        }
        return valid;
}
function validateVariance() {
        var validated = true;
        var cellValid = 0;
        for (r=startRow; r<numberGridRows; r++) {
                for (c=startCol; c<numberGridCols; c++) {
                        var cell = getCellVal(r,c);
                        if (isFinite(parseFloat(cell))) {
                                cellValid = limitPrecision(cellValid + getCellVal(r,c),3);
                                }
                        }
                }
        if (cellValid!=1) {
                validated = false;
                cellValid = limitPrecision(cellValid * 100,2);
                alert("The numbers you have entered do not equal 100% \nYour Total Percentage is: " + cellValid + " %");
                }
        return(validated);
}
function limitPrecision(value, limit) {
        if ((value == null) || (value.length == 0) || (value == missing)) {
                return(value);
        } else {
                return((Math.round(value * Math.pow(10,limit)))/Math.pow(10,limit));
        }
}
```

Tips

Ensure that you read the entire Appendix A of the *Hyperion Planning Administrator's Guide*—it contains useful tips, hints, and warnings. When you perform a Planning upgrade, you should back up your custom JavaScript, and then re-insert it in ValidateData.js after the upgrade is complete.

JavaScript can be very frustrating to the experienced programmer because there is no compiler, which means that the language cannot be tested prior to its execution. The debugging and writing of JavaScript is best accomplished using some of the tools listed in the Resources section of this document.

You can use any function in the Planning library (for example, EnterData.js), however, those functions are subject to change and their compatibility with future versions is not ensured. If you are going to use an existing Planning function from one of the Planning libraries, it is recommended to copy the function and all of its dependent functions into your custom JavaScript, rename them, and use the renamed functions so that you are better protected against upgrades.

If you use any custom JavaScript, it is important that you use *all* of the custom JavaScript functions contained in ValidateData.js. Add your custom code where necessary and leave the existing functions and their return calls in place. When ValidateData.js is active, Planning looks for all of the functions contained within and if some are missing, errors may occur.

Resources

The *Hyperion Planning Administrator's Guide*, specifically Appendix A, which includes all of the JavaScript samples and documentation.

Mozilla Web Browser: *http://www.mozilla.org/download.html* - the version referenced in this document is Mozilla 1.7.12, which is the full Mozilla suite.

Crimson Editor: *http://www.crimsoneditor.com/* - a free, comprehensive text editor.

CPad: *http://zantii.net/* - another free text editor.

TextPad: *http://www.textpad.com/* - a powerful text editor (not free but well worth it).

The JavaScript Source - *http://javascript.internet.com/* - a Web repository of JavaScript code along with tutorials, documentation, and sample code.

Endnotes

[1] This code is used to draw a table within the form.

[2] There are a number of ways to test for a number. isFinite() is just one of them.

[3] The \n in the alert causes a new line to be displayed in the message box.

Hyperion™
The future in sight