CNS Audit Example (Updated for Version 1.2)

This example demonstrates how to create an extremely flexible audit system to log field changes in your records. The bulk of the work is done through Custom Functions. Once the custom functions are set up, you can add logging to a table easily by creating two or three fields in your table and a layout with the fields that you would like to log. You can track any type of field including repeating fields and container fields.

Custom Functions (Required)

The solution uses three custom functions:

Audit_XMLValues

This custom function is used to generate data for the "Audit_FieldData" field. It creates a very basic XML structure. This custom function is used in the "Audit_FieldData" field to create two nodes: "Prior" and "Current". Both nodes hold values for each field that you are tracking, one for the prior state and one for the current state. The function recursively calls itself in order to create data for all fields.

Audit_GetXMLData

This custom function is used to grab the text of an XML tag and is used in the "Audit_Log" custom function and the "Audit_FieldData" field.

Audit_Log

This custom function does all of the comparing and reporting on field changes for your record. There are a lot of things going on here. The function has several comments in it to help you grasp what is going on and when. The function also recursively calls itself. One interesting thing that it does is check to see if the current changes are made on the same day as the last changes. If the changes are not on the same day, a banner is added to separate the old data from the new so that it will be easier to read.

Custom Functions (Optional)

Two optional custom functions have been added in version 1.2:

RemoveValuesbyPattern (New)

This custom function removes all values from a list that contain a search string. For example, it can remove several gui fields which have the prefix "gui_". You could also use it to remove related fields by using the search string "::".

RemoveValues (New)

This custom function returns only values that are not in an "omit" list. In other words, you can remove fields with the name "one" or "two" from your fields list. You can actually list several fields to remove and the function will remove all of them.

Three Fields...

(In this audit log example, the three fields can be created in any order.)

Timestamp

Create (if you do not already have one) a timestamp field which auto-enters the modification timestamp. This field is used to trigger the auto-enter calculations in the "Audit_Log" and "Audit_FieldData" fields any time the record has been changed.

Audit_Log

This is the actual audit log report field. This field has an auto-entered calculation, and you want to be sure to uncheck "Do not replace existing value of field (if any)" so that it will update each time the timestamp changes. Below is the auto-enter calculation:

```
Let (
        [
                theTimestamp = timestamp ;
                logEnabled = Case ( IsEmpty ( Get ( ScriptName ) ) ; "True" ) ;
                theFields = FieldNames ( Get ( FileName ) ; "Audit Log Fields" ) ;
        theFieldList = theFields // All fields
    ];

    Case (
        logEnabled = "True" ; Audit_Log ( theFieldList ; Audit_FieldData ; Audit_Log ) ;
        Audit_Log
    )
)
```

The "timestamp" field is referenced as a variable so that anytime the timestamp field changes this field will update.

Several variables have been added between version 1.1 and version 1.2 to aid in more functionality.

The first new variable is the "logEnabled" variable which should be set to "True" in order to enable logging. The example ships with the value:

```
Case ( IsEmpty ( Get ( ScriptName ) ) ; "True" )
```

The case statement above checks if there is a script running by using the Get (ScriptName) function to retrieve a script name. If there is no name, the value is true. The end result is that fields modified during a script are not logged as long as you commit the record in the script.

The second new variable is the "theFields" variable which is basically a master list of all of your field names. You can have field names in this variable that you do not want to log. Only fields represented in the "Audit_FieldData" field are actually logged.

In the interest of portability, the "theFieldsList" variable was added so that you can synchronize with your field list in the "Audit_FieldData" field. Remember, this is actually not necessary as long as all the fields you need are represented in the "theFields" variable.

At this point, a case statement has been introduced into the calculation to determine if logging should be enabled for the current record modification:

```
Case (
    logEnabled = "True" ; Audit_Log ( theFieldList ; Audit_FieldData ; Audit_Log ) ;
    Audit_Log
)
```

If the variable "logEnabled" is "True" then we use the "Audit_log" custom function to update the log. If "logEnabled" is not "True" then we simply return the "Audit_Log" field as it currently is. The prototype for "Audit_log" custom function is:

```
Audit_Log ( Fields; Data; CurrentLog )
```

The first parameter of the "Audit_Log" custom function is "Fields". This parameter is a list of field names. The second parameter of the "Audit_Log" custom function is "Data". This is a complete set of data which is generated in the "Audit_FieldData" field which is described below. The third parameter is "CurrentLog" which is a complete copy of the audit log and should be the actual "Audit_Log" field. This allows us to append the history of the log to the end of the new changes.

Audit_FieldData

This field keeps a complete account of the current and prior states of the record. It uses the "Audit_XMLValues" function to create a basic XML structure which we can easily grab values from in the "Audit_Log" function. This field is also an auto-entered calculation, so you will want to uncheck "Do not replace existing value of field (if any)" so that it will update each time the timestamp changes. Below is the auto-enter calculation:

```
Let (
    [
        theTimestamp = timestamp ;

        theFields = FieldNames ( Get ( FileName ) ; "Audit Log Fields" ) ;
        // theFields = ValueListItems ( Get ( FileName ) ; "Audit Log Fields" ) ;

        // ### theFieldList Examples ###
```

```
                //theFieldList = theFields // All fields
                //theFieldList = RemoveValuesbyPattern ( theFields ; "::" )
                //theFieldList = RemoveValuesbyPattern (  theFields ; "nav_" )
            theFieldList = RemoveValuesbyPattern ( RemoveValuesbyPattern ( theFields ; "::" ) ;
"nav_" )

                //theFieldList =  RemoveValues ( theFields; "Company,Repeating[2]" )
        ];

            "<prior>"  & Audit_GetXMLData ( "current"; Audit_FieldData ) & "</prior>¶"  &
            "<current>¶" & Audit_XMLValues ( theFieldList ) & "</current>"
        )
```

As in the "Audit_Log" field, the "timestamp" field is referenced so that this field will be updated when the record changes.

The calculation uses the "Audit_GetXMLData" function to grab the old "current" data from the Audit_FieldData field (itself). This information is placed in the "prior" tags so that we know this information is from the prior state. We then use the "Audit_XMLValues" function to create data for the current record changes and put this information in the "current" tags. The custom function prototype is as follows:

```
Audit_XMLValues ( fields )
```

For version 1.1, we added the "theFields" and "theFieldsList" variables to aid in creating your fields list. The "theFields" variable holds the master list of fields. You can then use the "RemoveValuesbyPattern" and "RemoveValues" custom functions to remove any fields you do not need. For example, you can remove fields from your list such as a container field that is used for a button. Using the "RemoveValuesbyPattern" you can remove any field name that matches a search string. So if you had several container fields on your layout that were buttons, you can remove them from your list with the search string "nav_" if their field names begin with "nav_". The "RemoveValues" function removes a specific field or fields from the list. Also introduced is the the idea of using a Value List to hold your field names. There are at least two drawbacks to using a Value List. First, you have to type in each field name that you want to log. Second, you will have to return to the Value List to update any field names when you change them in the define fields dialog box. The advantage to using a Value List is the ability to log repeating fields easily. For example, you can log a repeating field in a Value List by using the syntax "Repeating[n]" for each repetition of a field. Perhaps a mix of both worlds would be best if you need to log a log of repeating fields. A combination of the FieldNames function and a Value List to adding all of your repeating fields.

To see an example of this data, view the "Pure Audit Fields" layout.

Conclusion

Though this may be a bit to grasp at first, after you set this up once you will begin to understand how it works. After it is set up in a database file, it is extremely easy to add two or three fields to your tables and a layout for your audit fields. Setting up a new audit log for a table should only take 10 minutes or so after the custom functions have been created. At that point, you can add and remove fields from your audit log layout as needed. Keep in mind that you must define a separate field for each repetition of a repeating field on your audit log layout (unless you use the Value List method mentioned above), but you do not need to show all repetitions on your data entry layouts in order to audit them properly. See the "Audit Log Fields" layout for an example.

If you see any mistakes or have any comments or improvements on this solution, please use the contact form on the CNSPlug-ins.com website at http://www.cnsplug-ins.com/contact.htm.