RECORD SCRIPTS

One way to make your application more modular and customizable, is to off-load some of the control of its behavior to a collection of records in a table with a **Script** field. This is a special sort of field that, when displayed in the form, allows you to write JavaScript code, and provides basic JavaScript linting[1] functionality, syntax highlighting, and some other useful features like automatic indenting of nested code blocks.

The ability to write JavaScript code and store it in a record in a table in ServiceNow is cool but you may be asking yourself, "what does that actually… do?" – Well, think about it; Business Rules are basically just scripts stored in a table (`sys_script`), right? Something must tell that script to execute, and how. Turns out, we can do the same thing!

This will all probably become much clearer with an example, so let's consider a scenario in which we have an application which exports records from the Incident table in ServiceNow, and sends them off to some other system. Maybe the other system is an archive, or maybe they're doing some external analysis on the data – doesn't matter. We need to get the data out of ServiceNow, and into this other system.

That may seem straightforward. Just write a Script Include with some methods to do the work of packaging up an Incident into a JSON object. Make sure it's structured in whatever way the receiving system wants it to be structured. Maybe write a Business Rule or Scheduled Script Execution to trigger the export trigger (depending on whether we want the data exported immediately upon update, or once every so-often, respectively).

Remember though, we're trying to make our application as **modular** and **configurable** as possible.

What if the receiving system is another ticketing platform, and it calculates priority differently than ServiceNow does? If that's the case, you'll have to do some calculation to determine what the appropriate priority in the new system *ought* to be, based on data in the ticket in ServiceNow.

"*Okay, no problem.*" you might say, "*I'll just write a special handler function for calculating the Priority value that we send to the target system*".

Then you learn that the target system has a maximum character count in the description field, of only 1,000 characters; and worse, that passing in too many characters causes a hard failure resulting in the record being rejected.

"*Okay*", you continue. "*I can write a special handler for that as well*".

And the **State** field in the target system is of course, not an integer, like it is in ServiceNow.

"*Third special handler; no biggie*", you say, as you die a little inside.

Oh, and did they forget to mention? The target system only has one layer of category – no subcategory field – and the categories and subcategories don't match up with what's in ServiceNow.

"*Uh… I guess I could write a sort of a nested hash-map that—*"

And they want to be able to easily map custom fields from ServiceNow, as they're added, without having to sort through what is now a *massive* Script Include to slip the code in-between all your special handlers.

"*Er…*"

Wouldn't it be nice if there were a better way[2]?

Oh good, there is: scripted control records.

Let's dive into an example of how one might handle this scenario. I'm going to skip some of the basics (such as which fields to check) when it doesn't matter, just for the sake of this example.

[1] "Linting" means using a tool to analyze your code to flag bugs, errors, syntactical issues, etc. (such as missing semicolons or malformed code blocks).

[2] This is where I would have put a meme of Billy Mays saying, "*There's got to be a better way!*", but I'm pretty sure I'd get dinged for some kind of obscure copyright thing.

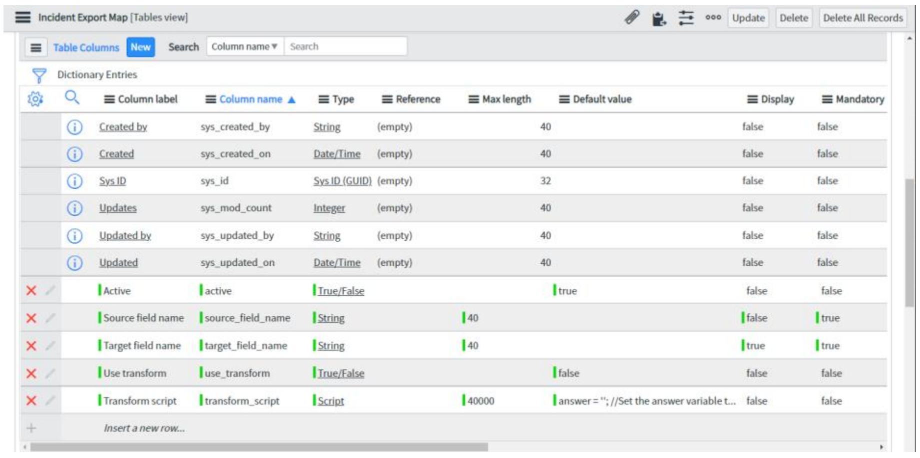
DEFINING SCRIPTED RECORDS

We're going to create the scripted records first, which might be a little confusing until we get to the part where we invoke the scripts in those records, but stay with me; I promise it'll make sense by the end.

First, let's create a table to store some records that will comprise the "field mappings" from the Incident table to whatever table the records are going into, in the target system. Let's call that table `inc_export_map`[3].
After saving the table record, let's add a couple of fields:

1. Active [active]
    a. **Type**: True/False
    b. **Default value**: true
2. Source field name [source_field_name]
    a. **Type**: String
    b. **Max length**: 40
    c. **Mandatory**: true
3. Target field name [target_field_name]
    a. **Type**: String
    b. **Max length**: 40
    c. **Mandatory**: true
    d. **Display**: true
    e. **Unique**: true
4. Use transform [use_transform]
    a. **Type**: True/False
    b. **Default value**: false
5. Transform script [transform_script]
    a. **Type**: Script
    b. **Max length**: 40000

When finished, our table should look something like this, and then we can hit **Save**:



Column label	Column name	Type	Reference	Max length	Default value	Display	Mandatory
Created by	sys_created_by	String	(empty)	40		false	false
Created	sys_created_on	Date/Time	(empty)	40		false	false
Sys ID	sys_id	Sys ID (GUID)	(empty)	32		false	false
Updates	sys_mod_count	Integer	(empty)	40		false	false
Updated by	sys_updated_by	String	(empty)	40		false	false
Updated	sys_updated_on	Date/Time	(empty)	40		false	false
Active	active	True/False			true	false	false
Source field name	source_field_name	String		40		false	true
Target field name	target_field_name	String		40		true	true
Use transform	use_transform	True/False			false	false	false
Transform script	transform_script	Script		40000	answer = ''; //Set the answer variable t...	false	false
Insert a new row...							

Now, just to help ourselves and our users out, let's add a **default value** for the **Transform script** field, that gives the user a bit of scaffolding. We could have the script evaluate to set a variable that we define, such as `answer`, by using something simple like this:

[3] Don't forget what we learned in the **Naming Conventions** chapter: table names are always singular!

```
answer = ''; //Set answer to the target field value.
```

Then, in the code we use to invoke the script in this field, we would simply need to retrieve the final value of that variable after the script's execution (for which there is a simple API in ServiceNow that we'll see later).

However, we're going to want to pass in a couple of other variables so that they can be used *within* the script, which requires a slightly more complex bit of scaffolding. Let's take Business Rules as an example. – In a Business Rule, there is a function which has passed into it, two important variables: `current`, and `previous`. As a reminder, here is the default value for an advanced Business Rule's script field:

```
(function executeRule(current, previous /*null when async*/) {

    // Add your code here

})(current, previous);
```

Fig. todo

As you can see, the `executeRule` function is defined on line 1, and is in what's called an "**IIFE**", which stands for **Immediately Invoked Function Expression**. What makes this function "immediately invoked", is the `()` on the last line, into which are passed the `current` and `previous` variables. This is just a method for declaring and invoking a function in a **single statement**. This pattern functions basically exactly the same as if you were to *declare* a function and *then* execute it using two separate statements, like so:

```
function executeRule(current, previous /*null when async*/) {
    // Add your code here
}

executeRule(current, previous);
```

Fig. todo

The `current` and `previous` variables are defined as **arguments** in the function header on line 1, but they aren't defined outside the function; so how can they be *passed in* to the function as **arguments**? Well, as we'll see shortly, these variables are defined by the script that *invokes* the code in this record. There is an API for executing scripts inside of records like this (`GlideScopedEvaluator`), and it handles that part for us.

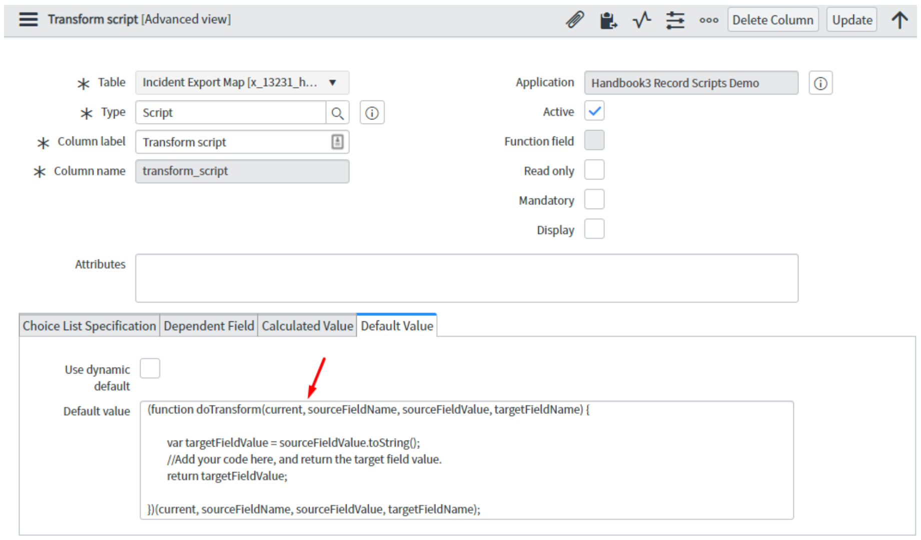
Before we get to talking about how to invoke our script though, let's write our script field's default value. Using the pattern that Business Rules use as an example, we're going to set our script's default value to something like this:

```
(function doTransform(current, sourceFieldName, sourceFieldValue, targetFieldName) {

    var targetFieldValue = sourceFieldValue.toString();
    //Add your code here, and return the target field value.
    return targetFieldValue;

})(current, sourceFieldName, sourceFieldValue, targetFieldName);
```

Fig. todo

In this script, we're passing in four variables: `current` (the current record for which we're doing the transformation), `sourceFieldName` (the name of the source field, based on the current transform record), `sourceFieldValue` (the original value of the source field in ServiceNow), and `targetFieldName` (the target field name, based on the current transform record). In the transform script, we can use any (or none) of those variables to do our transformation, and then return the final target field value.

Now that we've got a decent default value for our script field, we'll go to that field's dictionary record, and set the **Default value** field to our script. It won't be syntax-highlighted in the field, but that's okay – it will be on the transform record.



Transform script [Advanced view]

Table	Incident Export Map [x_13231_h...
Type	Script
Column label	Transform script
Column name	transform_script
Application	Handbook3 Record Scripts Demo
Active	✓
Function field	
Read only	
Mandatory	
Display	
Attributes	

Choice List Specification | Dependent Field | Calculated Value | Default Value

Use dynamic default	

Default value

```
(function doTransform(current, sourceFieldName, sourceFieldValue, targetFieldName) {

    var targetFieldValue = sourceFieldValue.toString();
    //Add your code here, and return the target field value.
    return targetFieldValue;

})(current, sourceFieldName, sourceFieldValue, targetFieldName);
```

If this were a "real" application, this is the part where we'd go in and fiddle with the form layout to make it pretty, and then add a UI Policy to make it so the **Transform script** field only shows up when **Use transform** is set to true, but this is just an example so I'm going to skip those steps.

One thing to know about these script fields, is that there are two main ways to get data *out* of them, after running them (which, again, we'll see how to do shortly). As mentioned above, you can choose a variable (such as `answer`), have the script set the value of that variable to whatever you want the result to be, then *retrieve* the value of that variable after the script has executed. Note that this would **not** work with the pattern that we're using, because our `targetFieldValue` variable is wrapped in a function scope. However, if we were to remove the function wrapper, then we *could* just set that variable to the desired value and retrieve it using our calling script; but then we wouldn't be able to explicitly pass in the variables we're making available to the script (like `current` and `sourceFieldValue`). Even though those variables would still technically be available within the script, it would be more confusing and harder to know what exactly you had available to you while writing the transform script.

The second way of getting some value out of our transform script, is to simply have it **evaluate to** the final target field value. The idea of a script **evaluating to** some value might be a new one, so here's a quick lesson:

To simplify quite a bit, a script "evaluates to" whatever the **final expression** evaluates to. Consider the following script:

```
var myName = 'Tim';
var greeting = 'Hello, ' + myName;
```

Fig. todo

This script does not actually evaluate to *anything*, because although the last statement in the script is doing work, that statement does not return a value; therefore, it does not "evaluate to" anything. You can test this yourself, by opening up your web browser (preferably Chrome), pressing F12, navigating to the Console, and entering this code there. Upon pressing Enter, you'll probably see a line show up that says something like `undefined`. That's because your script didn't evaluate to anything!

If we add a third line, however, we can cause our little script to evaluate to the value of `greeting`:

```
var myName = 'Tim';
var greeting = 'Hello, ' + myName;
greeting;
```

Fig. todo

That third line "evaluates to" the value of `greeting`, which – since it's the last line of our script – means that *our script evaluates to* the value of `greeting`!

Now, applying this lesson to our scripted records, you might be able to see how our script evaluates to whatever value the function returns. The function, being an IIFE, is immediately invoked (or *run*), and since the combination declaration-and-invocation of that function is the *last statement* in our script, that means that *our script evaluates to* whatever that function returns!

***Pro-tip**: If we were to add another line of code below our function – even something like* `var a = 3;` *- it would mean that our script would no longer evaluate to the value returned by the function!*

INVOKING SCRIPTED RECORDS

Now that we've got our **Incident Export Map** table defined, and our **Transform script** field is set up with a useful default value, it's time to write the code that will use these potentially-scripted records.

First, we need to get an Incident record to do the transformation on. Once we have that, we need to loop through all our active export map records for that Incident and evaluate each of them. We'll focus on the function responsible for doing the actual transformation.

The API class we're going to be making use of to execute these scripts, is called `GlideScopedEvaluator`[4]. This class has the methods `.putVariable()`, `.getVariable()`, and `.evaluateScript()`. It's the `.evaluateScript()` method that we're going to use, since it has optional parameters allowing us to specify the variables to pass into the script in the export map's script field, so we don't need to use the `.putVariable()` method (although we could, if we wanted).

Now comes the big chunk of code that ties everything we've done so far together. Read through it yourself once and see if you can identify what all it's doing, then I'll walk you through the code line-by-line, below.

```
function getTransformedFieldVals(grIncident) {
    var sourceFieldName, targetFieldName, sourceFieldVal;
    var mappedValues = {};
    var gsEval = new GlideScopedEvaluator();
    var grExportMap = new GlideRecord(
        'x_13231_hb3_rec_sc_inc_export_map'
    );
    grExportMap.addActiveQuery();
    grExportMap.query();
    while (grExportMap.next()) {
        sourceFieldName = grExportMap.getValue('source_field_name');
        targetFieldName = grExportMap.getValue('target_field_name');
        sourceFieldVal = grIncident.getValue(sourceFieldName);
        //If scripted transform isn't necessary, just set
        // target value from source value & continue.
        if (grExportMap.getValue('use_transform') != '1') {
            mappedValues[targetFieldName] = sourceFieldVal;
            continue; //Continue to next loop iteration.
        }
        mappedValues[targetFieldName] = gsEval.evaluateScript(
            grExportMap, //GR for map record
            'transform_script', //Script field,
            { //Variables to be accessible within the script
                'current': grIncident,
                'sourceFieldName': sourceFieldName,
                'sourceFieldValue': sourceFieldVal,
                'targetFieldName': targetFieldName
```

[4] API docs for `GlideScopedEvaluator` can be found here: https://developer.servicenow.com/dev.do#!/reference/api/quebec/server/no-namespace/c_GlideEvaluatorScopedAPI

```
                }
            );
    }
    return mappedValues;
}
```

Fig. todo: Getting all transformed field values for a provided Incident.

Let's go line-by-line, and analyze what's happening here.

Line 2: "Hoisted" variable declarations, to keep our code clean and to make sure our code is *written* as close to how the JavaScript engine will actually *run* it as possible.

Ln 3-7: Declaring a variable containing an empty object (which is where our transformed field values will end up), our instantiation of `GlideScopedEvaluator` (which we'll use to evaluate the scripts in our export map records), and a `GlideRecord` object we'll use to query the Export Map table.

Ln 8-10: Querying the Export Map table; getting only **active** export map records.

Ln 11-13: Getting the values that we'll use to determine the target field value (whether by running a transform script or just getting the original field value).

Ln 16-19: Checking if the **Use transform** field is **not** set to `true`. If it isn't set to true, then just set the mapped value to the original value from ServiceNow, and `continue;` - meaning stop the current iteration of the `while` loop, and skip on to the next one.

Ln 20-29: Call the `.evaluateScript()` method of `gsEval` (our instance of the `GlideScopedEvaluator` class). For the first argument, we're passing in the GlideRecord corresponding to the specific export map record we're working on. The second argument is the name of the script field within that record. The third and final argument is an object containing a list of key:value pairs, where the *key* is the name of a variable to be available within the script as it runs, and the *value* is the actual value of that variable when the script executes.

Ln 31: Finally, once we've looped through all export map records and finished adding all mapped fields and mapped (or transformed) values to the `mappedValues` object, return that object, which should now contain a complete representation of the record in the target system, including all mapped fields and values.

The beauty of this approach is that in order to add new fields to the mapping, we don't need to add code! We simply need to add a new export map record! We *can* write some code in the export map record, but if no transformation is necessary, we can just leave the **Use transform** field unchecked, and the original value from the source field in ServiceNow will be sent along for the target field.