Version: Tokyo

NOW PLATFORM GUIDES

Technical Best Practices

Technical Best Practices are a guide to proven strategies for developing applications by writing efficient, easy-to-read scripts, creating re-usable logic, and avoiding performance pitfalls.

Scripting Technical Best Practices

Building strong functionality in ServiceNow begins with writing high-quality code. Take steps to make your code manageable, efficient, and scalable from the early stages. This helps to ensure good performance, reduces the chances of problems, and simplifies the debugging process.

Covered in this guide:

- Make Code Easy to Read
- Create Small, Modular Components
- Variables
- Interacting with the Database
- Use Self-Executing Functions
- Avoid Coding Pitfalls

Make Code Easy to Read

When writing code, remember that others may work with it in the future. Whenever possible, ensure your code is easy to read and understand. Follow any formatting standards used by your organization.

Covered in this section:

- Comment Your Code
- Use White Space
- Write Simple Statements

Comment Your Code

What may seem obvious today may not be clear in the future, especially on a complicated section of code. Comments should be well-written and clear, just like the code they are annotating.

Single line comments start with a double forward slash (//) and can be placed anywhere on the line. Everything after the // and up to the new line is considered a comment. For example:

```
// Initialize variables
 var count = 0;

 if (count > MAXVAL) // Check if we are over the limit
```

Block comments start with a forward slash and one asterisk (/*) and end with the reversed combination (*/). Everything between the markers is considered a comment. Adding extra asterisks, like those shown in this example, can make it easier to locate blocks of comments.

```
/***
  * getRiskEnvironment - determine the environment of the CIs to calculate the risk
  *
  * @param: chg - GlideRecord of change record
  * @return: result - Highest environment value of CIs: 1=production, 2=QA, 3=test, 4=dev, 0=error
  *
  ***/
```

Clearly describe the purpose of functions and all inputs and outputs. For example:

```
/***
 * putUserPref - save a user preference/value pair for this user
 *               If the preference already exists, check the current value and update if necessary
 *
 * @param prefName - name of the user preference
 * @param prefVal - string value to save
 * @return - None
 *
 ***/
```

Keep comments as accurate as possible. Erroneous comments can make programs even harder to read and understand. Also be sure to keep comments up to date as the scripts are updated. The comment in this code section does not clearly describe its function.

```
//
// Send notifications to the user's manager
//
function get5RecentRecords() {

    var list = [];
    var task = new GlideRecord('task');

    task.addQuery('active', true);
    task.orderByDesc('sys_updated_on');
    task.setLimit(5);
    task.query();

    while (task.next()) {
        list.push(task);
    }

    return list;
}
```

Write meaningful comments. Focus on what is not immediately visible. For example:

```
// Unhelpful comment:
    var i = 0; // Set i to 0

// Better comment:
// Set this record to never remind again
     gr.return_reminder = gs.nowDateTime();
     gr.update();
```

Use White Space

Use empty lines and spaces to make code more readable. The easier it is to read code, the easier it is to identify and correct issues. Empty lines help visually group blocks of code together so the reader can see the logical organization. Spaces on each line help make the items on an individual line easier to read.

The format code button in the ServiceNow syntax editor toolbar is a useful tool for adjusting indentation without altering other spacing.

```
// Example of poor white space usage
function createRelationship(relType,childCI,parentCI){
var newRel=new GlideRecord('cmdb_rel_ci');
var newRelType=new GlideRecord('cmdb_rel_type');
if(childCI==parentCI)
return;
var relTypeID;
if(newRelType.get(relType)){
relTypeID=newRelType.getValue('sys_id');
newRel.initialize();
newRel.setValue('type',relTypeID);
newRel.setValue('child',childCI.getValue('sys_id'));
newRel.setValue('parent',parentCI.getValue('sys_id'));
newRel.insert();
}
}
```

The lack of indentation and spacing in the previous example makes it difficult to determine the flow of logic. By adding a few spaces, the code becomes more readable:

```
// Example of good white space usage
function createRelationship(relType, childCI, parentCI) {
// Put spaces between parameters and before curly brace

    var newRel     = new GlideRecord('cmdb_rel_ci');
// Put spaces around '='.
    var newrelType = new GlideRecord('cmdb_rel_type');
// Optionally, put additional spaces to align '=' w/previous line.
    var relTypeID;
// Group variables together where appropriate

    if (childCI == parentCI)
// Put a space after 'if' and around operators ('!=')
        return;

    if (newRelType.get(relType)) {
// Use extra blank lines to help break up chunks of code and improve readability

        relTypeID = newRelType.getValue('sys_id');

        newRel.initialize();
// Group similar statements together
        newRel.setValue('type', relTypeID);
        newRel.setValue('child', childCI.getValue('sys_id'));
        newRel.setValue('parent', parentCI.getValue('sys_id'));
        newRel.insert();
    }
}
```

Write Simple Statements

Remember that less experienced developers may work with your code in the future. Make it as easy to maintain as possible. In general, it is the compiler's job to make code fast, so minimize the use of fancy tricks. For example, experienced programmers may not have an issue with the JavaScript ternary operator:

```
var result = (x == y) ? a : b;
```

Less experienced developers may find it challenging to understand. Instead, the statement could be written more clearly as:

```
var result;

if (x == y) {
    result = a;
}
else {
    result = b;
}
```

Create Small, Modular Components

Break your code into modular components or specialized functions. Script Include functions are excellent examples of this technique. Script Includes are essentially libraries of functionality that can be implemented in other server-side scripts, such as Business Rules, UI actions, and Script Actions. Some of the benefits of specialized functions include:

- They are easy to create because they are small and simple with limited functionality.
- They are generally simpler and shorter, so it is easy to understand the logic, inputs, and outputs. This makes it easier for the next person who works with the code to make modifications.
- They are easier to test. As a related note, when you test the code, be sure to test both valid and invalid inputs to ensure the script is as robust as possible.

As you create small specialized functions, keep in mind how the small bits fit together in a larger construct. For example, you may find that doing a query of the same data inside ten separate functions is not database-friendly, and that a different approach is necessary.

Construct Reusable Functions

If you see the same or similar logic repeated in a process, consider creating a specialized function. This improves the quality of code, saves you the trouble of searching through multiple blocks of similar code when a problem arises, and makes code easier to maintain.

For example, assume a script adds a user to one watch list, adds another user to a different watch list, and then adds a configuration item (CI) to a user-defined glide_list field. The only things which are different between the code for each command are the field name and the element being inserted; the logic is the same. The script could be constructed as a single function addGlideListElement(listName, elementID), and called repeatedly, as in this Script Include:

```
var AcmeIncidentFunctions = Class.create();
AcmeIncidentFunctions.prototype = {
    initialize: function() {
    },

    /*
     * addGlideListElement - add a sys_id to a specified glide list field
     *
     * @param grField - field element to add a value (e.g. current.watch_list)
     * @param id - sys_id to add (if it doesn't already exist)
     * @return - new list values
     *
     */
    addGlideListElement : function(grField, id) {

        // Get the list of values
        if (JSUtil.notNil(grField))
            var idList = grField.split(',');
        else
            var idList = [];

        // Determine if the id passed is already in the list
        var found = false;
        for (var i = 0; i < idList.length; i++) {
            if (id == idList[i]) {
                found = true;
                break;
            }
        }

        // If it was not in the list, then add it
        if (!found)
            idList.push(id);

        // Return the comma separated list of ids
        return idList.join(',');
    },

    type: 'AcmeIncidentFunctions'
}
```

Script Includes can be used in Business Rules, workflow script activities, or other server-side scripts:

```
var aif = new AcmeIncidentFunctions();
current.watch_list = aif.addGlideListElement(current.watch_list, gs.getUserID());
```

or

```
current.watch_list = new AcmeIncidentFunctions().addGlideListElement(current.watch_list, gs.getUserID());
```

Variables

Covered in this section:

- Use Descriptive Variable and Function Names
- Use Variables to Store Function Results
- Verify Values Exist Before Using Them
- Return Values

- Avoid Dynamic JEXL Expressions in an Evaluate
- Avoid the eval Function

Use Descriptive Variable and Function Names

Meaningful names for functions and variables better indicate to readers the purpose of code. Consider this code excerpt:

```
// Poor function and variable names
function del(r, d, s) {

    var a=0;

    if (s == 13) // 13=cancel/delete
        r.deleteRecord();
    else
        a = d;

    return a;
}
```

It is difficult to determine the purpose of function del(r, d, s). A better way to write this is:

```
function deleteIfCanceled(glideRec, defaultAnswer, stateValue) {

    var answer = 0;

    if (stateValue == 13)
        glideRec.deleteRecord();
    else
        answer = defaultAnswer;

    return answer;
}
```

Though it is helpful to be descriptive, there are cases where shorter variable names are acceptable. In some programming languages, it is common practice to use the variable i for counting in a looping statement (such as for loops). The following code is perfectly acceptable:

```
for (var i = 0; i < myArray.length; i++)  {
     // Do some important processing on each record here
}
```

Use Variables to Store Function Results

Avoid situations where the same function is repeatedly called with the same result. This is common when you call functions with no parameters. Depending on the function, repeated calls may negatively affect performance.

When possible, use a descriptive variable to store a value and refer to the variable instead of calling the same function repeatedly. Using variables can also make the code easier to understand. Consider this code sample:

```
if (gs.getUserID() == current.assigned_to ||
    gs.getUserID() == current.u_coordinator ||
    gs.getUserID() == current.caller_id ||
    gs.getUserID() == current.caller_id.manager) {
    // Do something important here
}
```

This code can be written more efficiently:

```
var currentUser = gs.getUserID();
var isOwner = (currentUser == current.assigned_to);
var isCoordinator = (currentUser == current.u_coordinator);
var isCaller = (currentUser == current.caller_id);
var isManager = (currentUser == current.caller_id.manager);

if (isOwner || isCoordinator || isCaller || isManager) {
   // Do some important processing here
```

Verify Values Exist Before Using Them

To avoid unpredictable results and warning messages, verify that variables and fields have a value before using them. Consider the following code:

```
var table = current.cmdb_ci.installed_on.sys_class_name;
gs.print('Table is: ' + table);
```

Any additional statements which use the variable table may throw warning messages in the system log if the value is undefined. This can happen if the cmdb_ci field is empty or if the installed_on field in the CI is empty. The following code demonstrates a better way to verify that the table variable has a value before using it:

```
var table = current.cmdb_ci.installed_on.sys_class_name;

if (table)
    gs.print('Table is: ' + table);
else
    gs.print('Warning: table is undefined');
```

Return Values

Get in the practice of returning some type of value when you create new functions. The value returned can tell the calling code useful information about how the function executed. Examples of return values include:

- The number of records read from a table (return 0 to indicate an error).
- The success status of a particular operation (return true to indicate success).
- A JavaScript object (return NULL to indicate a failure).

Examples of values returned are:

```
if (!saveRecord(current))
    gs.addErrorMessage('Save Error');

/*
 * saveRecord - save a record
 *
 * @param rec - GlideRecord
 * @return - boolean (true=successful save)
 *
 */
function saveRecord(rec) {

    var id = rec.update();

    if (!id.nil())
        return true;

    return false;
}
```

Avoid Dynamic JEXL Expressions in an Evaluate

When writing Jelly code, avoid using dynamic JEXL expressions inside the Jelly tag (or <g2:evaluate> for phase two). While the code appears to work, it affects a memory resource (called PermGen) in the Java Virtual Machine, which can lead to performance issues and even system outages over time. The exception to using JEXL expressions inside <g:evaluate> tags is with static values, including: ${AMP}, ${AND}, ${GT}, ${LT}, and ${SP} (and their phase two counterparts: $[AMP], $[AND], and so on).

A better way to use Jelly variables inside <g:evaluate> tags is to include the attribute jelly="true", then reference a copy of the variable with the jelly. prefix.

Incorrect:

```
<j:set var="jvar_userid" value="46d44a23a9fe19810012d100cca80666" />
<g:evaluate>
    var inc = new GlideRecord('incident');

    inc.addQuery('assigned_to', '${jvar_userid}');
    inc.addQuery('priority', ${sysparm_priority});
    inc.query();
</g:evaluate>
```

Correct:

```
<j:set var="jvar_userid" value="46d44a23a9fe19810012d100cca80666" />
<g:evaluate jelly="true">
    var inc = new GlideRecord('incident');

    inc.addQuery('assigned_to', jelly.jvar_userid);
    inc.addQuery('priority', jelly.sysparm_priority);
    inc.query();
</g:evaluate>
```

Avoid the eval Function

The eval() function evaluates or executes an argument. Improper use of eval() opens up your code for injection attacks and debugging can be more challenging, as no line numbers are displayed with an error, for example.

Consider the following code to achieve the same outcome:

```
GlideEvaluator.evaluateString("gs.log('Hello World');");
```

Interacting with the Database

Covered in this section:

- Avoid Complex GlideRecord Queries
- Use GlideAggregate for Simple Record Counting
- Avoid Complex Queries on Large Data Sets
- Let the Database Do the Work

Avoid Complex GlideRecord Queries

Rather than creating a series of addQuery() and addOrCondition() calls to obtain a result, use addEncodedQuery() to make the query easier to create and maintain.

Consider a requirement to obtain a list of all active Apple printers and computers in a company's Santa Ana office. Creating the proper combination of addQuery() and addOrCondition() queries to get the proper solution may sound simple. However, if the requirement changes and you are asked to add another office location and a different hardware manufacturer to the list, the task of maintaining the query can become challenging.

For complex GlideRecord queries, it is easier to create a query string by generating encoded query strings through a filter and using that string with addEncodedQuery. As requirements change, you can create a new query string using the list filter, verify the results with the requirement author, and use the query in the same script.

Use GlideAggregate for Simple Record Counting

If you need to count rows, you have two options: the getRowCount() method from GlideRecord, or GlideAggregate. Using GlideRecord to count rows can cause scalability issues as tables grow over time, because it retrieves every record with the query and then counts them. GlideAggregate gets its result from built-in database functionality, which is much quicker and doesn't suffer from the scalability issues that GlideRecord does.

Bad example:

```
/*
 * countInactiveIncidents - return the number of closed incidents
 *
 * @param - none
 * @returns integer - number of records found
 *
 */
function countInactiveIncidents() {

     var inc = new GlideRecord('incident');

     inc.addInactiveQuery();
     inc.query();

     var count = inc.getRowCount();
     gs.print(count + ' inactive incidents found');

     return count;
}
```

Good example:

```
/*
 * countInactiveIncidents - return the number of closed incidents
 *
 * @param - none
 * @returns integer - number of records found
 *
 */
function countInactiveIncidents() {

     var inc = new GlideAggregate('incident');

     inc.addAggregate('COUNT')
     inc.addInactiveQuery();
     inc.query();

     var count = 0;
     if (inc.next())
          count = inc.getAggregate('COUNT');

     gs.print(count + ' inactive incidents found');

     return count;
}
```

Avoid Complex Queries on Large Data Sets

Limit the number of times you search large tables. As your instance grows, these searches can affect performance. Assume you have a requirement to search the CMDB for the importance of all upstream services related to a specific server when that server is added to the Incident form. Running a query on the Relationship [cmdb_rel_ci] table is not a problem for a simple CMDB with a few hundred or thousand CIs. However, for a CMDB with three million CIs and hundreds of thousands of relationships, the query could take hours.

One solution is to create a related list for the CI that lists affected services. The list can be updated by a business rule as relationships for the CI are updated. When a CI is added to an incident, the affected services list can be quickly retrieved from the related list on the CI, rather than launching a long search on the Relationship table.

Let the Database Do the Work

Whenever possible, leverage the power of the database to retrieve the proper records. For example, if you are checking 1,000,000 incidents to see if at least one record is active, your first solution may look like this:

```
var inc = new GlideRecord('incident');

inc.addQuery('active', true);
inc.query();

if (inc.hasNext())
   // There is at least one active record
```

However, if there are 250,000 active records, the query() method has to retrieve all those records. That can take time. Instead, use the setLimit() method to instruct the database to return only one record. Returning one record is much faster than returning all the records.

```
var inc = new GlideRecord('incident');

inc.addQuery('active', true);
inc.setLimit(1);  // Tell the database to only retrieve a maximum of 1 record
inc.query();

if (inc.hasNext())
   // There is at least one active record
```

Use Self-Executing Functions

A self-executing function is both declared and invoked within the same script field. Whenever writing a script that only needs to run in a single context, such as a transform map script, use this type of function. For functions that must run in multiple contexts, consider reusable functions instead.

By enclosing your script in a self-executing function you can ensure that the script does not impact other areas of the product, such as by overwriting global variables. A self-executing function follows this format:

```
(function functionName() {

  //The script you want to run

})(); //Note the parenthesis indicating this function should run.
```

You can declare functions within the self-executing function. These inner functions are accessible only from within the self-executing function.

```
(function functionName() {

  function helperFunction(){
    //return some value
  }

  var value = helperFunction(); //Valid function call.

  //perform any other script actions

})();

var value2 = helperFunction(); //Invalid. This function is not accessible from outside the self-executing functior
```

Avoid Coding Pitfalls

Covered in this section:

- Experiment in a Sandbox
- Code in Stages
- Do Not Use Hard-Coded Values
- Avoid Dot-Walking to the sys_id of a Reference Field
- Use getDisplayValue() Effectively

Experiment in a Sandbox

Trying out new coding ideas or concepts in a development instance can lead to messy results later on. If you make changes, then alter your approach while using an update set, you may find unwanted changes getting in the update set and being promoted to other instances. If you do not use an update set for your proof of concept, your development instance may behave differently than your other instances, making it difficult to verify defects and fixes.

If you do not have a sandbox environment available, use a ServiceNow demo instance. When you understand how to implement the new concept in a sandbox, build the solution in development.

Code in Stages

Do not attempt to write hundreds of lines of code in one sitting. This is especially true if you are learning a new technology. Write a few lines of code at a time and test as you go to ensure it works as expected.

Although this process may seem slow and tedious at first, it is ultimately more effective. If you try to write too much code at one time, it becomes difficult to locate the source of a problem. Instead of tracking back through lines and lines of code to find the defect, save time and effort by incrementally writing functional code.

Do Not Use Hard-Coded Values

Avoid using hard-coded values in scripts, as they can lead to unpredictable results and can be difficult to track down later. Hard coding sys_ids is not recommended, as they may not be the same between instances. Instead, try looking up a value by reference or by creating a property and retrieving the value with gs.getProperty().

Hard-coding group, user, or other names often leads to problems when an organizational change occurs. For example, assume you assign a value of Service Desk to the script variable for a group name:

```
var taskID = '26c811f06075388068d07268c841dcd0';
var groupName = 'Service Desk';
```

The script will not function as expected when the group name changes from Service Desk to Help Desk. Instead, use a property:

```
var taskID = gs.getProperty('acme.default.task');
var groupName = gs.getProperty('acme.group.name');
```

Also consider the following example in which a workflow needs an approval from the IT director of Acme Corp. Sara, the former IT director, has been replaced by Dale. See why the initial approach of hard-coding Sara's user information is not preferable.

Initial Approach:

1. Create a user approval activity for Sara.
2. When Dale becomes the new IT director, update the workflow.

Better Approach:

This method eliminates the need to add hard-coded data in the workflow.

1. Create a group, IT Director, and make Sara a member.
2. Use the Group Approval Activity.
3. When Dale becomes the new IT director, simply update the group membership instead of the workflow.

Avoid Dot-Walking to the sys_id of a Reference Field

It is not necessary to include the sys_id of a reference field when dot-walking, as in the following example:

```
var id = current.caller_id.sys_id; // Wrong
```

The value of a reference field is a sys_id. When you dot-walk to the sys_id, the system does an additional database query to retrieve the caller_id record, then retrieves the sys_id. This can lead to performance issues. Instead, use the statement.

```
var id = current.getValue('caller_id'); // Right
```

Use getDisplayValue() Effectively

When scripting, become familiar with the tables and fields you are using. Instead of using name, number, or other explicit field name used for the display value, use the method getDisplayValue(). Consider the following example:

```
var parent = current.parent.number;
var myCI = current.cmdb_ci.name;
```

You would be required to update this code if the display value requirement changes on either of these two tables. For example, the business now wants to display the CI's serial number field instead of name. When someone changes the dictionary attribute on the Configuration Item [cmdb_ci] table, from name to serial_number, this code no longer reflects the current requirements. Instead, consider using the following code:

```
var parent = current.parent.getDisplayValue();
var myCI = current.cmdb_ci.getDisplayValue();
```

To determine which field on a table is used as the display value, inspect the dictionary entry for the table and note which field has a Display field value of true.

Client Scripting Technical Best Practices

A Client Script is JavaScript code which runs on the client, rather than the server. Well-designed Client Scripts can reduce the amount of time it takes to complete a form and improve the user experience. However, improperly implemented Client Scripts can significantly slow down form load times. With the exception of the onCellEdit Client Script, Client Scripts apply to forms only.

Follow these best practices to ensure Client Scripts work efficiently.

Covered in this guide:

- Client Scripting Considerations
- Run Only Necessary Client Scripts
- Enclose Code in Functions
- Minimize Server Lookups
- Client Scripting Practices to Avoid

Client Scripting Considerations

Covered in this section:

- Use Client Scripts to Validate Data
- Set Client Script Order
- Restrict List Editing

Use Client Scripts to Validate Data

An excellent use for Client Scripts is validating input from the user. Validation improves the user experience because the user finds out if there are data issues before submitting the information. An example of validation is to verify that the Impact field value is valid with the Priority field value. In this example, Low impact is not allowed with High priority.

```
if (g_form.getValue('impact') == '3' && g_form.getValue('priority') == '1') {
   g_form.showErrorBox('impact', 'Low impact not allowed with High priority');
}
```

Set Client Script Order

Client Scripts are not executed in a specific order, however you do have the ability to set an order of execution, very similar to UI Policies.

- Add the baseline Order field to the Client Script form.
- Scripts are executed in order from low to high. A script with an Order value of 100 executes before a script with an Order value of 300. Use UI Policies Instead of Client Scripts to Set Field Attributes

When possible, consider using a UI Policy to set field attributes to mandatory, read-only, or visible. No scripting is required to set field attributes in UI Policies.

Restrict List Editing

With the exception of onCellEdit Client Scripts, UI policies and Client Scripts apply to forms only. If you create UI policies or Client Scripts for fields on a form, you must use another method to ensure that data in those fields is similarly controlled in a list. You can:

- Disable list editing for the table.
- Create appropriate business rules or access controls for list editing.
- Create data policies.
- Create a separate onCellEdit Client Script.

Run Only Necessary Client Scripts

Client Scripts have no Condition field. This means onLoad() and onChange() scripts run in their entirety every time the appropriate form is loaded. To avoid running time-consuming scripts unnecessarily, make sure Client Scripts perform only necessary tasks.

This example is an inefficient onChange() Client Script set to run when the Configuration item field changes.

```
//Set Assignment Group to CI's support group if assignment group is empty
function onChange(control, oldValue, newValue, isLoading) {

    var ciSupportGroup = g_form.getReference('cmdb_ci').support_group;

    if (ciSupportGroup != '' && g_form.getValue('assignment_group') != '')
        g_form.setValue('assignment_group', ciRec.support_group.sys_id);
}
```

The following steps provide examples showing how this example can be improved to prevent the Client Script from running unnecessary code.

Covered in this section:

- General Cleanup
- Keep the isLoading Check (onChange Client Scripts Only)
- Keep the newValue Check
- Add the newValue != oldValue Check
- Bury the GlideAjax Call

General Cleanup

Look for performance optimizations. In the previous example, the getReference(), or GlideRecord lookup can be replaced with an asynchronous GlideAjax call.

```
//Set Assignment Group to support group if assignment group is empty
function onChange(control, oldValue, newValue, isLoading) {

    var ga = new GlideAjax('ciCheck');

    ga.addParam('sysparm_name', 'getSupportGroup');
    ga.addParam('sysparm_ci', g_form.getValue('cmdb_ci'));
    ga.getXML(setAssignmentGroup);
}

function setAssignmentGroup(response) {

    var answer = response.responseXML.documentElement.getAttribute("answer");

    g_form.setValue('assignment_group', answer);
}
```

Keep the isLoading Check (onChange Client Scripts Only)

The isLoading flag is the simplest way to prevent unnecessary code from taking up browser time. The isLoading flag should be used at the beginning of any script which is not required to run when the form is loading. There is no need to run this script on a form load because the logic would have already run when the field was last changed. Adding the isLoading check to the script prevents it from doing a cmdb_ci lookup on every form load.

```
//Set Assignment Group to CI's support group if assignment group is empty
        function onChange(control, oldValue, newValue, isLoading, isTemplate) {

            if (isLoading)
                return;

            var ga = new GlideAjax('ciCheck');

            ga.addParam('sysparm_name', 'getSupportGroup');
            ga.addParam('sysparm_ci', g_form.getValue('cmdb_ci'));
            ga.getXML(setAssignmentGroup);
        }

        function setAssignmentGroup(response) {

            var answer = response.responseXML.documentElement.getAttribute("answer");

            g_form.setValue('assignment_group', answer);
        }
```

If the onChange script should run during form load, use the following convention:

```
function onChange(control, oldValue, newValue, isLoading, isTemplate) {

    if (isLoading) {}; // run during loading

    // rest of script here

}
```

Keep the newValue Check

The newValue check tells this script to continue only if there is a valid value in the relevant field. This prevents the script from running when the field value is removed or blanked out. This also ensures that there will always be a valid value available when the rest of the script runs.

```
//Set Assignment Group to CI's support group if assignment group is empty
function onChange(control, oldValue, newValue, isLoading, isTemplate) {

    if (isLoading)
        return;

    if (newValue) {
        var ga = new GlideAjax('ciCheck');

        ga.addParam('sysparm_name', 'getSupportGroup');
        ga.addParam('sysparm_ci', g_form.getValue('cmdb_ci'));
        ga.getXML(setAssignmentGroup);
    }
}

function setAssignmentGroup(response) {

   var answer = response.responseXML.documentElement.getAttribute("answer");

   g_form.setValue('assignment_group', answer);
}
```

Add the newValue != oldValue Check

To have the script react to a value which changes after the form loads, use the newValue != oldValue check.

```
//Set Assignment Group to CI's support group if assignment group is empty
function onChange(control, oldValue, newValue, isLoading, isTemplate) {

    if (isLoading)
        return;

    if (newValue) {
        if (newValue != oldValue) {
            var ga = new GlideAjax('ciCheck');

            ga.addParam('sysparm_name', 'getSupportGroup');
            ga.addParam('sysparm_ci', g_form.getValue('cmdb_ci'));
            ga.getXML(setAssignmentGroup);
        }
    }
}

function setAssignmentGroup(response) {

   var answer = response.responseXML.documentElement.getAttribute("answer");

   g_form.setValue('assignment_group', answer);
}
```

Bury the GlideAjax Call

In this example, the GlideAjax call is buried one level deeper by rearranging the script to check as many things available to the client as possible before running the server calls. The script checks the assignment before executing the GlideAjax call. This prevents the server lookup when the assignment_group field is already set.

```
//Set Assignment Group to CI's support group if assignment group is empty
function onChange(control, oldValue, newValue, isLoading, isTemplate) {

    if (isLoading)
       return;

    if (newValue) {
        if (newValue != oldValue) {
            if (g_form.getValue('assignment_group') == '') {
                var ga = new GlideAjax('ciCheck');

                ga.addParam('sysparm_name', 'getSupportGroup');
                ga.addParam('sysparm_ci', g_form.getValue('cmdb_ci'));
                ga.getXML(setAssignmentGroup);
            }
        }
    }
}

function setAssignmentGroup(response) {

   var answer = response.responseXML.documentElement.getAttribute("answer");

   g_form.setValue('assignment_group', answer);
 }
```

Enclose Code in Functions

Client Scripts without a function cause issues with variable scope. This is why Client Scripts are enclosed in a function by default. When code is not enclosed in a function, variables and other objects are available and shared to all other client-side scripts. If you're using the same variable names, it is possible they could collide. This can lead to unexpected consequences that are difficult to troubleshoot. Consider this example:

```
var state = "6";

function onSubmit() {

   if(g_form.getValue('incident_state') == state) {
               alert("This incident is Resolved");
   }
}
```

Because the state variable is not enclosed in a function, all client-side scripts, have access to it. Other scripts may also use the common variable name state. The duplicate names can conflict and lead to unexpected results. These issues are very difficult to isolate and resolve. To avoid this issue, ensure all your code is wrapped in a function:

```
function onSubmit() {

   var state = "6";

   if(g_form.getValue('incident_state') == state) {
               alert("This incident is Resolved");
   }
}
```

This solution is much safer because the scope of the variable state is limited to the onSubmit() function. Therefore, the state variable does not conflict with state variables in other client-side scripts.

Minimize Server Lookups

Client scripting uses either data available on the client or data retrieved from the server. Use client data as much as possible to eliminate the need for time-consuming server lookups. The top ways to get information from the server are g_scratchpad, and asynchronous GlideAjax lookup.

The primary difference between these methods is that g_scratchpad is sent once when a form is loaded (information is pushed from the server to the client), whereas GlideAjax is dynamically triggered when the client requests information from the server.

Other methods, GlideRecord and g_form.getReference() callback are also available for retrieving server information. However, these methods are no longer recommended due to their performance impact. Both methods retrieve all fields in the requested GlideRecord when most cases only require one field. The GlideRecord API is not available for scoped applications.

Covered in this section:

- Example: g_scratchpad
- Example: Asynchronous GlideAjax
- Use setValue()'s displayValue Parameter with Reference Fields

Example: g_scratchpad

The g_scratchpad object passes information from the server to the client, such as when the client requires information not available on the form. For example, if you have a Client Script which needs to access the field u_retrieve, and the field is not on the form, the data is not available to the Client Script. A typical solution to this situation is to place the field on the form and then always hide it with a Client Script or UI Policy. While this solution may be faster to configure, it is slower to execute.

If you know what information the client needs from the server before the form is loaded, a display Business Rule can create g_scratchpad properties to hold this information. The g_scratchpad object is sent to the client when the form is requested, making it available to all client-side scripting methods. This is a very efficient means of sending information from the server to the client. However, you can only load data this way when the form is loaded. The Business Rule cannot be triggered dynamically. In those cases, use an asynchronous GlideAjax call.

For example, assume you open an incident and need to pass this information to the client:

- The value of the system property css.base.color
- Whether or not the current record has attachments
- The name of the caller's manager

A display Business Rule sends this information to the client using the following script:

```
g_scratchpad.css = gs.getProperty('css.base.color');
g_scratchpad.hasAttachments = current.hasAttachments();
g_scratchpad.managerName = current.caller_id.manager.getDisplayValue();
```

Example: Asynchronous GlideAjax

This script compares the support group of the CI and the assignment group of the incident by name:

```
//Alert if the assignment groups name matches the support group
function onChange(control, oldValue, newValue, isLoading) {

    if (isLoading)
        return;

    var ga = new GlideAjax('ciCheck');

    ga.addParam('sysparm_name', 'getCiSupportGroup');
    ga.addParam('sysparm_ci', g_form.getValue('cmdb_ci'));
    ga.addParam('sysparm_ag', g_form.getValue('assignment_group'));
    ga.getXML(doAlert); // Always try to use asynchronous (getXML) calls rather than synchronous (getXMLWait)
}

// Callback function to process the response returned from the server
function doAlert(response) {

    var answer = response.responseXML.documentElement.getAttribute("answer");

    alert(answer);
}
```

This script relies on an accompanying script include, such as:

```
var ciCheck = Class.create();

ciCheck.prototype = Object.extendsObject(AbstractAjaxProcessor, {

    getCiSupportGroup: function() {

        var retVal = ''; // Return value
        var ciID   = this.getParameter('sysparm_ci');
        var agID   = this.getParameter('sysparm_ag');
        var ciRec  = new GlideRecord('cmdb_ci');

        // If we can read the record, check if the sys_ids match
        if (ciRec.get(ciID)) {
            if (ciRec.getValue('support_group') == agID)
                retVal = 'CI support group and assignment group match';
            else
                retVal = 'CI support group and assignment group do not match';

            // Can't read the CI, then they don't match
        } else {
            retVal = 'CI support group and assignment group do not match';
        }

        return retVal;
    }

});
```

Use setValue()'s displayValue Parameter with Reference Fields

When using setValue() on a reference field, be sure to include the display value with the value (sys_id). If you set the value without the display value, ServiceNow does a synchronous Ajax call to retrieve the display value for the record you specified. This extra round trip to the server can leave you at risk of performance issues.

Incorrect:

```
var id = '5137153cc611227c000bbd1bd8cd2005';

g_form.setValue('assigned_to', id);
// Client needs to go back to the server to
// fetch the name that goes with this ID
```

Correct:

```
var id = '5137153cc611227c000bbd1bd8cd2005';
var name = 'Fred Luddy';

g_form.setValue('assigned_to', id, name); // No server call required
```

Client Scripting Practices to Avoid

Covered in this section:

- Avoid Global Client Scripts
- Avoid DOM Manipulation

Avoid Global Client Scripts

A global client script is any client script where the selected Table is Global. Global client scripts have no table restrictions; therefore they will load on every page in the system introducing browser load delay in the process. There is no benefit to loading this kind of scripts on every page.

As an alternative, and for a more modular and scalable approach, consider moving client scripts to a base table (such as Task [task] or Configuration Item [cmdb_ci]) that can be inherited for all the child/extending tables. This eliminates the system loading the scripts on every form in the UI - such as home pages or Service Catalog where they are rarely (if ever) needed.

Avoid DOM Manipulation

Avoid Document Object Model (DOM) manipulation if possible. It can cause a maintainability issue when browsers are updated. The only exception is when you are in charge of the DOM: in UI Pages, and the Service Portal.

Instead, use the GlideForm API or consider a different approach for the solution. In general, when using DOM manipulation methods, you have to reference an element in the DOM by id or using a CSS selector. When referencing out-of-box DOM elements, there is a risk that the element ID or placement within the DOM could change thus causing the code to stop working and/or generate errors. It is recommended to review these objects and reduce the use of DOM manipulation methods as much as possible.

Business Rules Technical Best Practices

A Business Rule is JavaScript code which run when a record is displayed, inserted, updated, or deleted, or when a table is queried. Follow these guidelines to ensure that Business Rules work efficiently and to prevent unpredictable results and performance issues.

Covered in this guide:

- Know When to Run Business Rules
- Use Conditions in Business Rules
- Keep Code in Functions
- Prevent Recursive Business Rules
- Use Business Rules to Double-Check Critical Input
- Use Script Includes Instead of Global Business Rules~~~internal-javascript

Know When to Run Business Rules

The When field on the Business Rule form indicates whether the Business Rule script runs before or after the current object is saved to the database. The most commonly used Business Rules are before and after rules.

You can use an async Business Rule in place of an after Business Rule. Async Business Rules are similar to after rules in that they run after the database commits a change. Unlike after rules, async rules run in the background simultaneously with other processes. Async Business Rules allow ServiceNow to return control to the user sooner but may take longer to update related objects.

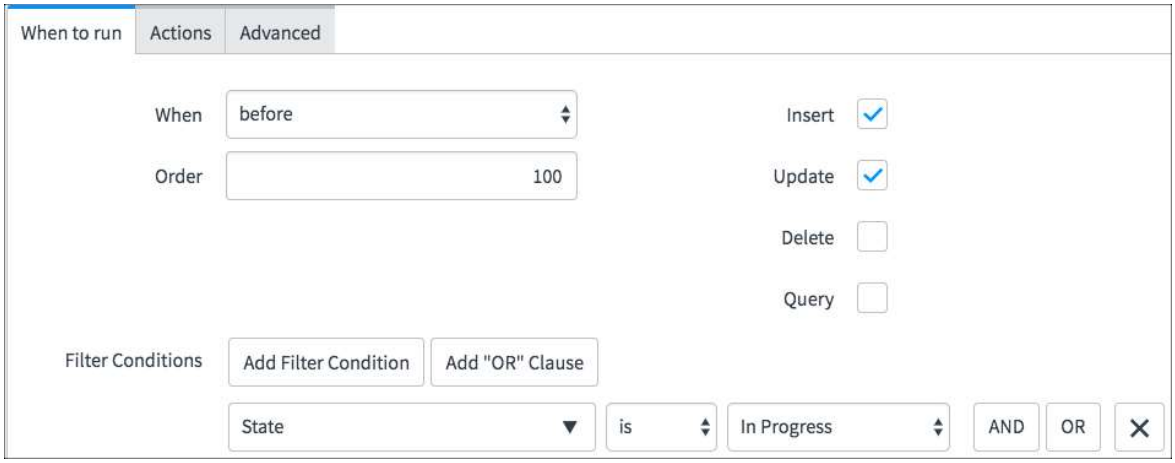
Follow these guidelines to determine which value to choose for the When field.

Value	Use Case
display	Use to provide client-side scripts access to server-side data.
before	Use to update information on the current object. For example, a Business Rule containing current.state=3; would set the State field on the current record to the state with a value of 3.
after	Use to update information on related objects that need to be displayed immediately, such as GlideRecord queries.
async	Use to update information on related objects that do not need to be displayed immediately, such as calculating metrics and SLAs.

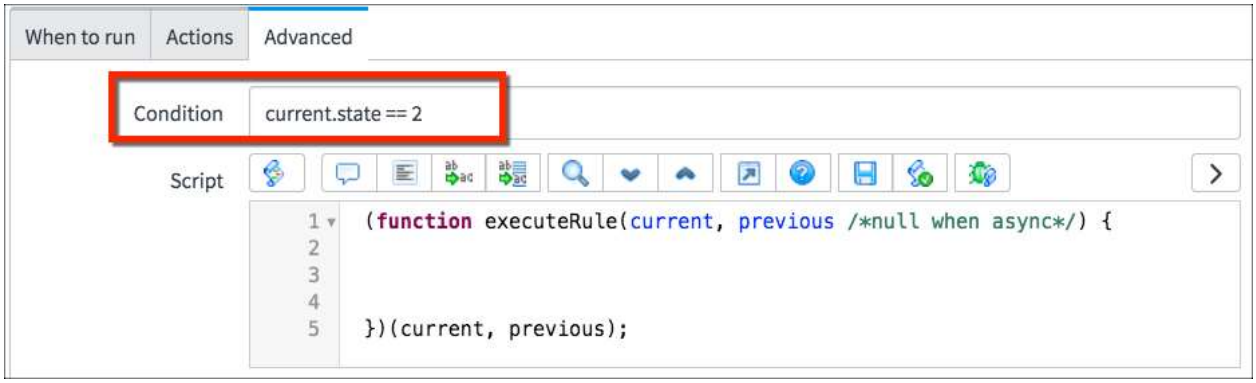
Use Conditions in Business Rules

Since Business Rules are evaluated whenever an insert, update, delete or query action is made to a record, it is important to ensure you are using conditions. Conditions are evaluated before the rule is executed, if the condition is met, the script is evaluated and executed. If there is no condition, the system assumes that the Business Rule should be evaluated and executed for every action to a record on the specified table of the Business Rule. It is easier to debug Business Rules when you can see which one meet a particular condition and which do not.

In the Business Rule form, use the Filter Conditions field in the When to Run section to specify the condition.



In the advanced Business Rule form, use the Advanced section to specify the condition. Specify the condition using the Condition field.



Keep Code in Functions

By default, an advanced Business Rule will wrap your code in a function, and it is important that this guideline is followed. When code is not enclosed in a function, variables and other objects are available to all other server-side scripts. This availability can lead to unexpected consequences that are difficult to troubleshoot. Consider this example:

```
var gr = new GlideRecord('incident');

gr.addQuery('active', true);
gr.query();

while (gr.next()) {
   // do some processing here
}
```

Because the gr object is not enclosed in a function, all server-side scripts, including script includes and other Business Rules, have access to it. Other scripts may also use the common GlideRecord variable name gr. The duplicate names can conflict and lead to unexpected results. These issues are very difficult to isolate and

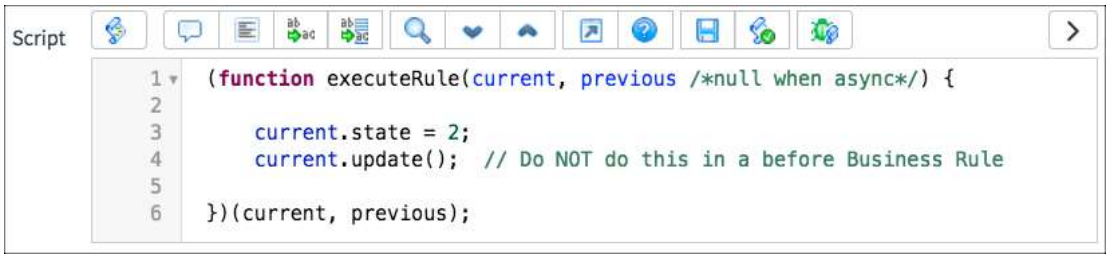
resolve because typical debugging methods identify the Business Rule giving erroneous results rather than the Business Rule containing global variables. To avoid this issue, keep the default function with the same code snippet:

```
(function executeRule(current, previous /*null when async*/) {
    var grInc = new GlideRecord('incident');

    grInc.addQuery('active', true);
    grInc.query();

    while (grInc.next()) {
        // do some processing here
    }
})(current, previous);
```

This solution is safer because the scope of the variable grInc is limited to the Business Rule since it is wrapped in the default function. If every script was wrapped, it would not matter what the variables were called. But it is good defensive coding to minimize the possibility of collisions even further, and avoid using a variable called gr at all. This makes the possibility of namespace conflicts even more remote.

Prevent Recursive Business Rules

Do not use current.update() in a Business Rule script. The update() method triggers Business Rules to run on the same table for insert and update operations, potentially leading to a Business Rule calling itself over and over. Changes made in before Business Rules are automatically saved when all before Business Rules are complete, and after Business Rules are best used for updating related, not current, objects. When a recursive Business Rule is detected, ServiceNow stops it and logs the error in the system log. However, this behavior may cause system performance issues and is never necessary.



You can prevent recursive Business Rules by using the setWorkflow() method with the false parameter, current.setWorkFlow(false). This will stop Business Rules and other related functions from running on this database access. The combination of the update() and setWorkflow() methods is only recommended in special circumstances where the normal before and after guidelines mentioned above do not meet your requirements.

Use Business Rules to Double-Check Critical Input

If you use a Client Script to validate data or a reference qualifier to present a filtered list, data may change between the time the user fills in the fields and the time the user submits the form. This mismatch can cause a conflict or error. Business Rules are an effective way to double-check critical input.

For example, a request allows users to reserve items. When users fill out the request form, they can only select currently available items. If two people fill out aBusiness Rule form at once and select the same item, the item appears to be available to both people because neither of them has submitted the form yet. If there is no Business Rule to double-check availability, ServiceNow assigns the same item to both requests. By using a Business Rule to re-verify item availability when the form is submitted, the second person receives a warning or other notification indicating the item has already been taken.

In this example, the Condition is: **current.cmdb_ci.changes()**

The script is:

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

/************************
 *
 * Double-check to ensure that no one else has selected and
 * submitted a request for the same configuration item (CI)
 *
 ************************/
doubleCheckCiAvailability();

function doubleCheckCiAvailability() {

    var lu = new LoanerUtils();

    if (!lu.isAvailable(current.cmdb_ci, current.start_date, current.end_date)) {
        gs.addErrorMessage(gs.getMessage('Sorry, that item has already been allocated'));
        current.cmdb_ci = 'NULL';
    }
}

})(current, previous);
```

Use Script Includes Instead of Global Business Rules

A global Business Rule is any Business Rule where the selected Table is Global. Any other script can call global Business Rules. Global Business Rules have no condition or table restrictions and load on every page in the system. Most functions defined in global Business Rules are fairly specific, such as an advanced reference qualifier on one field of one form. There is no benefit to loading this kind of script on every page.

Script includes only load when called. If you have already written a global Business Rule, move the function definition to a Script Include. The name of the Script Include must match the name of the function for the Script Include to work properly. There is no need to modify any calls to the named function.

Consider this global Business Rule:

```
function BackfillAssignmentGroup() {
		var gp = ' ';
		var a = current.assigned_to;

		//return everything if the assigned_to value is empty
		if(!a)
			return;
		//sys_user_grmember has the user to group relationship
		var grp = new GlideRecord('sys_user_grmember');
		grp.addQuery('user',a);
		grp.query();
		while(grp.next()) {
			if (gp.length > 0) {
				//build a comma separated string of groups if there is more than one
				gp += (',' + grp.group);
			}
			else {
				gp = grp.group;
			}
		}
		// return Groups where assigned to is in those groups we use IN for lists
		return 'sys_idIN' + gp;
	}
```

This Script Include is a better alternative:

```
var BackfillAssignmentGroup = Class.create();
BackfillAssignmentGroup.prototype = {
	initialize: function() {
	},

	BackfillAssignmentGroup:function() {
		var gp = ' ';
		var a = current.assigned_to;

		//return everything if the assigned_to value is empty
		if(!a)
			return;
		//sys_user_grmember has the user to group relationship
		var grp = new GlideRecord('sys_user_grmember');
		grp.addQuery('user',a);
		grp.query();
		while(grp.next()) {
			if (gp.length > 0) {
				//build a comma separated string of groups if there is more than one
				gp += (',' + grp.group);
			}
			else {
				gp = grp.group;
			}
		}
		// return Groups where assigned to is in those groups we use IN for lists
		return 'sys_idIN' + gp;
	},
	type: 'BackfillAssignmentGroup'
}
```

To call the function, use a script like this:

```
var ref = new BackfillAssignmentGroup().BackfillAssignmentGroup();
```

Debugging Best Practices

Debugging can provide information to help you understand system processes. There are a number of different debugging features you can use within a ServiceNow instance. Many browsers also feature a console log, which you can use for additional debugging.

Debugging best practices can be classified into two main areas: Server-side vs. Client-side practices.

Before you start debugging, it is important to identify the source of the issue, and to be able to reproduce the defect or bug. Potential sources include user error, invalid test data, test script inaccuracy, and technical implementation. To help identify and remediate the issue, the tester should provide the following information:

- The record being worked on—for example, CHG0012513
- The user ID used for the test—for example, Change Test User 3
- The steps required to reproduce the issue
- The expected result of the test
- The actual result of the test

Covered in this guide:

- Server-side Debugging
- Client-side Debugging
- Service Portal and External User Debugging
- Disabling Debugging

Server-side Debugging

Covered in this section:

- Debug Log
- Debug Business Rule
- Debug Security Rules
- Stop Debugging
- Server-Side Debugging Persistence
- Log Messages Controlled by a Property
- Scoped Application Script Logging
- Other Methods

Before starting work on an implementation, consider using one or more of the system debugging modules in the System Diagnostics and System Security applications. Enter debug in the navigation filter to display all debugging modules. Click a debugging module to activate it. For most but not all debugging features, when the debugging feature is active, and area labeled Debug Output appears at the bottom of the content frame.

Debug Log

System Diagnostics > Debug Log displays gs.debug(), gs.info(), gs.print() and gs.log() statements as well as server logging information and error messages at the bottom of the content frame. This helps you avoid alternating between the record you are trying to debug and the log output.

However, gs.print() and gs.log() are not available in scoped applications, whereas gs.debug() and gs.info() work in both scoped applications and global and are often used instead.

Debug Business Rule

System Diagnostics > Debug Business Rule displays messages about business rules. These messages indicate which business rules are being run and when they are started (==>), finished (<==), or skipped (===). If a business rule is skipped, the failed condition is displayed.

Debug Security Rules

System Security > Debug Security Rules places a debug icon on each field of a form. Point to the icon to see if there are any debug messages for the associated element. Click the icon to expand details about read and write access. This module is very helpful when you are using ACLs to control access to records and fields. The debug security rules have enhanced functionality, allowing you to view a context parameter.

Stop Debugging

System Security > Stop Debugging disables all debugging processes.

Server-Side Debugging Persistence

When you activate a server-side debugging module, it remains active until one of the following occurs:

- You activate the Stop Debugging module, located in System Security.
- You log out from the instance.
- The session expires (for example, the session times out).
- You close the browser.

Log Messages Controlled by a Property

The recommended way to debug server-side scripts is to use gs.debug() statements controlled by system properties. It enables you to have distinct debugging in script includes so they can each be debugged independent of each other. They can be turned on and off the debug property in the script include with a property so you don't have to modify code in production to enable/disable debugging.

A common use case is a debug function in a script include that checks the value of a specific system property to determine if it should output the message indicated. For example:

```
var MyUtil = Class.create();

MyUtil.prototype = {
    initialize: function(){
        this.debug       = gs.getProperty('debug.MyUtil') == 'true';
        this.debugPrefix = '>>>DEBUG: MyUtil: ';
    },

    myMethod : function(id) {

        // Some logic here
        this._debug('myMethod(): Sample debug output');
    },

    _debug : function(msg) {

        if (this.debug) {
            gs.debug(debugPrefix + msg);
        }
    },

    type : "MyUtil"
}
```

Then call it in various places as:

```
this._debug(recordCount + ' records read');
```

If the system property debug.MyUtil is set to false, nothing will be output to the log. If it is set to true, the debug message will be displayed. When combined with Debug Log or Debug Business Rule, this property can be used to enable or disable debug information without changing code and without impacting the user experiences of others on the system.

If a tester or user reports that something is not behaving as expected in test, QA, production, or another instance, you can enable the property, enable debugging output, and investigate your results quickly.

Scoped Application Script Logging

gs.print() and gs.log() are older and not available in scoped applications, whereas gs.debug(), gs.info(), gs.warn(), gs.error() work in both scoped applications and global are therefore are more versatile going forward in future versions.

Another helpful function gs.isDebugging() and can be used to determine if session debugging is on OR the scoped property is debug.

Other Methods

Some developers use the gs.addInfoMessage() or gs.addErrorMessage() JavaScript methods, but these may have an impact on the user experience. Consider the following situation: another developer is trying to debug a business rule running on the Task table while you are trying to do a code review or demo on the Incident table. Suddenly several messages appear at the top of the content frame, leaving you to wonder if they will appear in production when you are not even sure where they came from in development.

Using gs.log() (and alert() for client-side scripts) statements in scripts may seem fairly anonymous to other users at first. They do not show up on the screen and you have to know where to look to see them. The drawback to using gs.log() and alert() statements to output debug messages is that they often get left in the code. If gs.log()and alert()statements are left in the code, they can have a considerable impact on the size of the system log file. Just one gs.log()and alert()in the right spot can be triggered hundreds or thousands of times by users, system imports, scheduled jobs, and so on. Large log files full of debug messages adversely impact back up and restore times for information with little value in production. Controlling log messages with a system property prevents gs.log()and alert()statements from filling up the log. Additionally, system back ups and restores can take longer with erroneous production messages.

Client-side Debugging

To debug processes on the client side, click the settings icon in the banner frame, select Developer from the menu and turn on JavaScript Log and Field Watcher. This will open the log viewer at the bottom of the window. Along with useful system information, you may choose to include jslog() statements in client scripts and UI policy scripts to help isolate and resolve issues. Unlike server-side debug statements, jslog() statements in client-side scripts do not consume disk space. This means you can leave them enabled, even in production. alert() statements should be avoided if possible.

Browser Debugging

If available, check the browser's console log for additional debugging information. For example, the Firefox and Chrome console logs often display error messages that do not appear anywhere else. Using these logs can help speed up the development and troubleshooting processes by locating undefined objects.

Service Portal and External User Debugging

The Service Portal is often the only way external users access ServiceNow in certain applications (e.g., Customer Service). Currently, in the Service Portal interface, the section for Debug Output does not appear at the bottom of the content frame and therefore, debugging logging messages are not viewable except in the log. In the case of Debugging Security Rules quickly via the graphical Debug Output, an alternate solution is required to see it at the bottom of the screen if you need to debug as an external user since these ACL rules do not display in the log.

You can impersonate the external user who should only go to the Service portal, then in the internal platform view it will say "Security constraints prevent this view..." and it hides the menu and the page is blank there is nothing to do. At this point you usually type the service portal page appended to your URL like "http://instance.service-now.com/csm (http://instance.service-now.com/csm)" and go to the Service Portal, but the Debug out will not show up at the bottom of the content frame.

Instead, as an admin, you can first go to the normal internal interface of ServiceNow and get the URL of a specific page you wish to view the Security Rules on. Then, impersonate the external user and in a new browser go to that URL – the view and or form will not look the same, but you can still see the correct Debug information at the bottom of the content frame. (It won't actually bypass any true ACL security or Before Query

Business Rules so of course the external user still can't see any data they shouldn't.)
However, it does give you a better view than the SP pages and I can see the full graphical ACL debugging there which is incredibly helpful.

Disabling Debugging

Before you close out an update set or complete testing of the production instance, be sure to disable all server-side debugging to save log space in production. To disable debugging, search for all system properties that contain the string debug in the name and ensure they are all set to false.

This is another good reason to use properties to control debugging in a script. It makes enabling or disabling debugging a simple task, such as when you need to validate use cases or data.

Logs and Queues Best Practices

Reviewing queues and logs during development can help identify any issues that may not be immediately apparent. Uncorrected issues can cause performance problems and unexpected results. The goal is to achieve zero error and warning log entries to ensure the highest quality development.

Covered in this guide:

- Reviewing Warnings and Errors
- Checking Log File Sizes
- Diagnosing Problems with Slow SQL Statements
- Managing Queues
- Managing Event Logs

Review Warnings and Errors

Review queues and logs frequently throughout the development process to ensure correct operation. Waiting until development is complete makes diagnosis and correction more difficult.

The log level gives an indication of severity:

- **Information**: ServiceNow is working as expected, but a log message has been added so you know when it happened.
- **Warning**: Something isn't quite right, but the functionality typically still works, or the error was recovered from.
- **Error**: The functionality has aborted. The instance will continue to work in other areas, but the reason behind the error should be investigated and dealt with.
- **Debug**: This is low-level information which provides details about how processing is going. This may be how a SAML token or REST message is dealt with.

Navigate to one of these modules to see messages that match the appropriate log level:

- **System Logs > System Log > Warnings**: shows all warning messages.
- **System Logs > System Log > Errors**: shows all error messages.
- **System Logs > All**: shows all messages together. Run the filter conditions Level is Warning or Level is Error to view all warnings and errors in the same list.

Each entry should either be corrected or documented as a known issue.

Tip: Right-click the Created column heading and select Sort (z to a) to show the most recent warnings or errors at the top of the list. Then, add the Created by column to the Log list. As you work, check the list for recent warnings or errors and see who generated them.

Tip: The Source column can also be used to further help identify the cause of errors/warnings.

Checking Log File Sizes

Check the size of log files regularly. Navigate to System Logs > Node Log File Download to review recent log files (files with names that start with localhost_log). If a log file you have been checking suddenly increases in size, there may be excessive errors, debugging may be enabled, or a new plugin may have been activated. Additional investigation is recommended.

Note: After several days, the log file is zipped. This must be taken into consideration when identifying an increase or decrease in node log file size.

Viewing Log Detail

To review a system log file in detail:

1. Log in to the instance with the admin role.
2. Navigate to System Logs > Node Log File Download.
3. Click the name of the log file you want to download.
4. Under Related Links, click Download log.
5. When the file is saved to your local machine, review the log in a text editor.

Search for these messages, which may indicate potential issues:

- Slow evaluate
- Slow Business Rule
- Recursive Business Rule
- Compiler exception
- Warning - large table limit
- Extremely large result

If you find any of these messages in the log file, determine the root cause of the issue, such as a poorly performing script, and correct it. For example, Extremely large result may be caused by a poorly designed query returning too many results. Likewise, a recursive Business Rule may be the result of a current.update() statement placed in a before or after Business Rule.

As its name suggests, the Node Log File Browser allows you to browse the Node Log Files from within ServiceNow. To access this, navigate to System Logs > Node Log File Browser.

Diagnosing Problems with Slow SQL Statements

You may be able to diagnose potential performance issues by reviewing the Slow SQL Statements log found in System Diagnostics > Slow Queries. This table stores queries whose total execution time is longer than one second (1000ms). Use the data in the Total Execution Time and the Example columns to get an idea of

which script may be triggering slow queries and look for potential performance improvement opportunities.

In addition to slow SQL statements, it is also possible to gain a view of the execution time of other elements of the system. Elements include:

- Events
- Mutex locks
- Scripts
- Transactions

Managing Queues

Covered in this section:

- Checking the Email Queue
- Checking the ECC Queue

Checking the Email Queue

ServiceNow can notify selected users automatically via email of specific activities in the system, such as updates to Incidents or HR requests. It is recommended that you occasionally inspect the email queues to ensure email is operating properly. To check the email queues, navigate to the System Mailboxes application and select any of the modules, such as Inbox, Outbox, or Junk.

The lists provide information about each email notification, such as the creation date, user ID of the sender, the recipients, subject, and the state of the event.

For more information, open the ServiceNow Knowledge Base and access the following two articles:

- For troubleshooting inbound email, see KB0524472 (https://hi.service-now.com/kb_view.do?sysparm_article=KB0524472).
- For troubleshooting outbound email, see KB0521382 (https://hi.service-now.com/kb_view.do?sysparm_article=KB0521382).

Checking the ECC Queue

The External Communication Channel (ECC) queue is a table primarily used to control the MID server. The MID Server aids in communication and movement of data between the ServiceNow platform and external applications, data sources, and services. Periodic review of the ECC queue can help determine if problems exist with the instance or MID Server.

To review the ECC queue:

1. Log in to the instance with the admin role.
2. Navigate to ECC > Queue.
3. Create a filter condition containing Queue is output and State is ready.
4. The filtered list displays records waiting to be picked up by the MID Server (displayed in the Agent column). Take note of the oldest created records.

5. Depending on your MID Server configuration, output records may be in the ready state for up to four minutes. If you see records in the ready state with creation times older than four minutes, check to ensure the MID Server is running and communicating with the instance. If necessary, restart the MID Server and review the logs.

Managing Event Logs

Covered in this section:

- Reviewing Event Logs
- Removing Unused Events

Reviewing Event Logs

The events log records all system events that occur within ServiceNow. Periodically review the events log by navigating to either of these locations:

- System Logs > Events
- System Policy > Events > Events Log

Sort the Processing duration in descending order (Z-A) to determine which events are taking the longest to process (the processing duration is listed in milliseconds). Events that take a long time to process could be the result of inefficient scripts or a system error.

What constitutes a "long" duration is a matter of observation. In general, if an event takes longer than 1000 milliseconds, you should understand what email notifications or script actions are triggered by the event and what they are doing. It may be reasonable for a script action to take one second. On the other hand, email notifications are generally quick to process (a few milliseconds) and it would be unreasonable for an event triggering an email notification to take 6000 milliseconds. If such a case is found, you may need to contact ServiceNow Technical Support for assistance.

Check the Processed column for empty entries by adding the Processed is Empty condition to the list filter. Notice how many unprocessed events there are by checking the counter in the upper right, just above the list heading. Every minute or two, refresh the list and notice the new count. If the total number of records steadily decreases, this indicates the system has an event backlog and is working to catch up. If the number of unprocessed events steadily increases, it could indicate the system is not processing events correctly. You may need to contact ServiceNow Technical Support for assistance.

Removing Unused Events

You can improve overall system performance by removing unused events. From the Events list, right-click the Name column and select Group by Name.

This view allows you to see how many times a particular event has been logged. Review the records for the events with the highest counts and determine what actions the event triggers. If there is no action responding to the event, consider disabling the script that logs the event.

For example, assume there is a business rule that runs on the Configuration Item [cmdb_ci] table and logs the event ci.updated whenever a record is updated. The event was put in place for future use, but does not currently trigger a notification or script action. When you import 100,000 CIs, the business rule is run and adds 100,000 events to the event log which are needlessly processed by the event processing engine. Disabling the business rules on the import set or commenting out the gs.eventQueue() call in the business rule can prevent unused events from being logged.

Update Set Technical Best Practices

A well-defined migration process is essential for successfully moving changes from one instance to another. The heart of the migration process is a document that identifies necessary steps to migrate update sets, as well as data not captured by update sets. In some organizations, developers pass the responsibility of the update set migration process to different people or groups. It is also fairly common for more than one developer to maintain changes in multiple update sets, which increases the challenge of tracking details and getting the changes together in each instance. So a document detailing the migration procedure helps the team ensure that all changes operate as expected.

A good migration procedure should address the topics described here, as well as other topics appropriate to the situation.

Covered in this guide:

- Update Set Application Order
- Additional Data Migration

Update Set Application Order

In the migration procedure, create an ordered list which details how grouped update sets should be applied. Many organizations group system changes in routine (weekly or monthly) updates rather than promoting each change immediately. As several update sets are completed, the order in which they are applied may become important to ensure dependencies are met. For example, if Update Set A creates a new table and Update Set B changes the list layout of that table, committing Update Set B first results in an error. It cannot apply the new list layout to the table because the table has not yet been created.

Sample update set list:

Order	Update Set Name	Description
1	R1-CHG-S5-CWT	Change process sprint 5 weekly update
2	R1-CHG-S5-CWT-P1	Change process sprint 5 patch 1
3	R1-CHG-D03-CWT	Change defect bundle #3
4	R1-INC-S5-BAR	Incident process sprint 5 weekly update

Additional Data Migration

The migration procedure should include information about any additional data or configuration changes required to accompany the update sets. For example, some applications and processes, such as approval lookups, require data that is not captured in the update set. Be sure the migration procedure:

- Indicates where the additional data files are located. For example, state that they are attached to the update set in the development instance or provide a network drive path.
- Provides instructions for using the export and import procedure to transfer the data using XML files.

You may find that certain items need to be updated before or after an update set is committed. The migration procedure should include instructions for:

- Activating plugins
- Configuring system properties or other instance-specific settings
- Setting up MID Servers and identity servers
- Ensuring that the target table is available

XML Data Transfer Technical Best Practices

To occasionally migrate data from one instance to another, you can export the XML data from one instance and import it to another. This method ensures that all fields and values are transferred exactly. Migrating data in an XML file saves time for unscheduled data imports since there is no need to build an import set or a transform map.

Exporting and importing data in XML files is commonly used for records created in a development instance that must be migrated with the update sets as part of a migration procedure. If that is the case, consider adding a comment in the update set description so that the user installing the update set is aware that an XML import is required. Examples of these records include lookup tables, unit test records, and other information required to support production. Typically, this information is only migrated once and the overhead of an import set is not justified.

Covered in this guide:

- XML Data Export Technical Best Practices
- Import Records as XML Data

XML Data Export Technical Best Practices

Covered in this section:

- Exporting a Single Record
- Exporting Multiple Records

Exporting a Single Record

It is often useful to export a single record, such as an incident, a user, a configuration item (CI), or a scheduled job, from one instance to another. For example, if a user has issues in a production instance and you want to do in-depth diagnostics on the development instance without impacting the user, you can export the user, CI, or incident record as XML data for later import into the development instance.

To export a single record as XML data (requires admin role):

1. Sign in to the instance that contains the source data.
2. Navigate to the record you want to export.
3. Right-click the form header and select Export > XML (This record).

4. Depending on browser and settings, a dialog box may prompt you to save the file, or the browser may automatically save the XML file to the downloads folder specified in the browser preferences.

Exporting Multiple Records

Suppose you created a table to look up approvers to support the problem management process. The table, fields, security, forms, and many other configuration parameters are captured by an update set; however, the data records are not. To promote the lookup records from the development instance to other instances, you must manually export all the records.

To export multiple records as XML data (requires admin role):

1. Sign in to the instance that contains the source data.
2. Navigate to the list you want to export.
3. [Optional] Filter the list, if desired.
4. Right-click the list header and select Export > XML (List v3: Click the list menu and select Export, you will be presented with a menu, then select XML.)
5. In the export progress dialog box, click Download when the export completes.

6. Depending on browser and settings, a dialog box may prompt you to save the file, or the browser may automatically save the XML file to the downloads folder specified in the browser preferences.

Import Records as XML Data

Covered in this section:

- Importing XML
- Relations to Other Records

Importing XML

After you have successfully exported data from the source instance, you can import the XML file directly to the target instance. Importing XML does not trigger business rules or update the instance cache.

To import an XML file containing one or more records:

1. Sign in to the instance which should receive the data.
2. In the banner frame, click the menu arrow next to your name and select the **Elevate privileges** option.
3. In the Activate an Elevated Privilege dialog box, select the **security_admin** check box and click **OK**.
4. Navigate to any list in ServiceNow. Any list can be used because the XML file contains the destination table name for the records.
5. Right-click the list header and select **Import XML**. (List v3: Click the list menu and select **Import XML**.)
6. In the import screen, click **Choose File** and select the previously exported XML file.
7. Click **Upload**.

Relations to Other Records

When performing an XML import, it is important to keep in mind that only the current record will be exported and not the records related to it. For instance, while exporting a User record (sys_user table), the groups it belongs to and the roles it has been given will not be part of the XML file. In this context, it might be necessary to export the records describing those relations in separate XML files. In the previous example, the tables User Role (sys_user_has_role table) and Group Member (sys_user_grmember table) will need to be exported too.

In this example, because business rules are not running during an XML import, it is important to also export and import the relations to the roles, since importing the relations to the group will not give the user record the roles the groups contain.