



ServiceNow

Scripting in ServiceNow

Fundamentals

participant guide

© COPYRIGHT 2019 SERVICENOW, INC. ALL RIGHTS RESERVED.

ServiceNow provides this document and the information therein "as is" and ServiceNow assumes no responsibility for any inaccuracies. ServiceNow hereby disclaims all warranties, whether written or oral, express or implied by law or otherwise, including without limitation, any warranties of merchantability, accuracy, title, non-infringement or fitness for any particular purpose.

In no event will ServiceNow be liable for lost profits (whether direct or indirect), for incidental, consequential, punitive, special or exemplary damages (including damage to business, reputation or goodwill), or indirect damages of any type however caused even if ServiceNow has been advised of such damages in advance or if such damages were foreseeable.

TRADEMARKS

ServiceNow and the ServiceNow logo are registered trademarks of ServiceNow, Inc. in the United States and certain other jurisdictions. ServiceNow also uses numerous other trademarks to identify its goods and services worldwide. All other marks used herein are the trademarks of their respective owners and no ownership in such marks is claimed by ServiceNow.

Scripting in ServiceNow Fundamentals

Table of Contents

| | |
|---|------------|
| Module 1: Scripting Overview..... | 7 |
| Lab 1.1: Using the Syntax Editor | 36 |
| Lab 1.2: Syntax Checking | 41 |
| Lab 1.3: Explore Scripting Resources | 60 |
| Module 2: Client Scripts..... | 69 |
| Lab 2.1: Two Simple Client Scripts | 85 |
| Lab 2.2: g_form and g_user | 103 |
| Lab 2.3: Debugging Client Scripts | 115 |
| Lab 2.4: Client Scripting with Reference Objects..... | 128 |
| Module 3: UI Policies | 141 |
| Lab 3.1: Incident State Resolved/Closed | 150 |
| Module 4: Catalog Client Scripts & Catalog UI Policies | 161 |
| Lab 4.1: Control Variable Choices Catalog Client Script | 171 |
| Lab 4.2: Control Out of State Shipping Catalog UI Policy..... | 177 |
| Module 5: Business Rules | 185 |
| Lab 5.1: Debugging Business Rules | 220 |
| Lab 5.2: Current and Previous | 229 |
| Lab 5.3: Display Business Rules and Dot-walking | 233 |
| Module 6: GlideSystem..... | 241 |
| Lab 6.1: Setting the CAB Date | 252 |
| Lab 6.2: Re-open Problem Date Validation..... | 255 |

| | |
|--|------------|
| Module 7: GlideRecord | 265 |
| Lab 7.1: Two GlideRecord Queries..... | 285 |
| Lab 7.2: RCA Attached: Problem and Child Incidents | 290 |
| Lab 7.3: addEncodedQuery() | 294 |
| Module 8: Script an Event..... | 301 |
| Lab 8.1: Tracking Impersonations..... | 311 |
| Lab 8.2: Incident State Event | 314 |
| Module 9: Script Includes | 321 |
| Lab 9.1: Classless Script Include..... | 330 |
| Lab 9.2: Create a New Class | 338 |
| Lab 9.3: HelloWorld GlideAjax | 353 |
| Lab 9.4: Number of Group Members..... | 356 |
| Lab 9.5: JSON Object..... | 365 |
| Module 10: UI Actions | 381 |
| Lab 10.1: Client UI Action | 395 |
| Lab 10.2: Server UI Action | 397 |
| Lab 10.3: Client and Server UI Action | 400 |
| Module 11: Flow Designer Scripting..... | 405 |
| Lab 11.1: Build a Flow | 415 |
| Lab 11.2: Trigger a Subflow | 425 |
| Lab 11.3: Add a Script to a Flow | 438 |

Module 1: Scripting Overview

now.

| Scripting Overview |
|--------------------|
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Answer the five W's of Scripting
 - **Who** can script?
 - **What** is Platform scripting?
 - **When** should you script?
 - **Where** do scripts execute?
 - **Why** should you avoid scripting?
 - **How** do you script in ServiceNow?
- Introduce Application Scope
- Review ServiceNow APIs
- Explore where to get Scripting help

Labs

- Lab 1.1 Using the Syntax Editor
- Lab 1.2 Syntax Checking
- Lab 1.3 Explore Scripting Resources

Scripting Overview: Topics

now.

Who, What, When, Where, Why and How?

The Syntax Editor

Locate Your Scripts Quickly

Application Scope

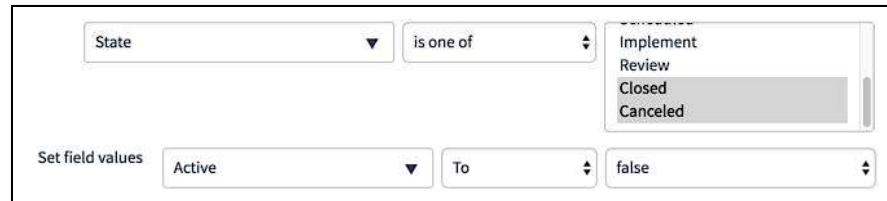
JavaScript in ServiceNow

Scripting Resources

What is Platform Scripting?

now.

- When making changes to your instance, the **Condition builder** should be used wherever possible to configure straightforward conditions and actions



- More complex configurations and behaviors can be scripted using **JavaScript**



ServiceNow provides the ability to **customize an instance using JavaScript** (based on Mozilla Rhino).

JavaScript, popularized by Yahoo, is the most prevalent scripting language on the web. It is object-oriented, runs within a browser, and does not need a license.

Rhino is an open-source implementation of JavaScript written entirely in Java. It is typically embedded into Java applications to provide scripting to end users.

The JavaScript engine used to evaluate scripts in ServiceNow supports the ECMAScript5 standard and is based upon Rhino version 1.7 R5.

Why Should You Avoid Scripting?

now.

- ServiceNow is continually evolving, what you scripted in the past might not need to be scripted now
- Easier to debug and fix configuration changes after an upgrade
- Make sure a script is really needed
 - How business critical is the requirement?
 - Will an ACL perform what you need instead?
 - Can you achieve 90% of your requirements via configuration changes instead of scripting?
- Consider ways of customizing **without scripting**



Remain current with ServiceNow's ongoing development:

- Check ServiceNow's Product and Developer documentation before developing anything new.
- Stay current on what has changed from release to release.
- Check the most recent version's release notes.
- Attend release meetings.

When Should You Script?

now.

- **Add new** functionality
- **Extend existing** functionality
- Guide users through messaging
- Automate processes
- Interact with 3rd party applications



Examples of when you should script:

- Update related record(s)
 - Cascade a comment from a master incident to its children.
 - Updating the ownership of a Configuration Item (CI) after a Change Request.
- Approval Strategies
 - One or all does not meet your business requirement.
 - Approvers need to be set dynamically.
- Show/hide a form section.
- Scan a list of CIs to dynamically determine risk based on criticality.
- Query database.
- Customize widgets.
- Change default behaviors.

The list of examples shown above is representative of typical uses for scripting. There are, of course, no limits as to what one can do with JavaScript in ServiceNow. It all comes down to your requirements!



TIP FROM THE FIELD

ServiceNow is always enhancing the platform. For example, the **Embedded Help** has been updated and a new feature called **Guided Tours** has been added. Research these options before you script a solution to guide users. If you have already scripted a solution, revisit your code and see if new baseline options can replace the script.

When Should You Script?

now.

A Word of Caution!

- Modified scripts are considered customer-owned
- Skipped during ServiceNow upgrades
- System treats the **Active** field as **Update Exempt**

In the past, the best practice strategy for updating baseline scripts was:

- Make a copy of the record you want to update
- Make the original record inactive
- Make changes to the script in the copy

Problems with Copy, Deactivate, and Change the Copy

Twice the Records to Maintain

X2

Customized Records do not get functionality updates



Releases: H I J K L

A Better Process

- Carefully considered customization
- Will no code changes work?
- Is customization necessary?
- Modify the baseline record
- Review and revert the skipped file after an upgrade

ServiceNow is always trying to make the upgrade process easier. Adopting this process will help with future upgrades to your instance.

The phrase “Modify a baseline script and you own it.” strengthened the copy, deactivate, and change the copy process. The strategy was validated when the number of skipped files were reduced after an upgrade. The original record, other than changing the value of the Active field, was still the same and it was upgraded. Upgraded baseline files and fewer skipped files was seen as a good thing.

Problems with this strategy:

- There are two files, the original and the copy, that need to be maintained. Maintenance doubles each time a customization is made.
- With each release, the customized record becomes older. Customers do not receive the advancements of a new release. In some cases, a new release may rely on the original record being updated. Developers may make more changes to compensate for the original record being inactive.

The current better choice is:

- Understand that customization can be costly and should be carefully considered.
- Research ServiceNow’s APIs. Can a no-code approach do the same thing?.
- Determine if the customization is necessary.
- Modify the baseline record.
- Be ready to review and revert the skipped file, if needed, after an upgrade.

Note: Changes made to a record that are only the result of changing the **Active** field are excluded from update tracking. This allows you to change the field value without affecting the *Updated* and *Updated by system* fields.

If you are interested in learning more about **Excluding a field from an update**, see ServiceNow’s Product documentation for additional information.

Who Can Script?

now.



System Administrator

Manages all the features, applications, and data in the platform



System Definition Administrator

Manages a specific System Definition (e.g. *Can only manage Business Rules*)



Application Administrator

Manages all the features and data of an application

Baseline, the **admin** role has access to all platform features, functions, and data, regardless of security constraints. **Grant this privilege carefully!** If you have sensitive information such as private HR records, ServiceNow recommends granting a custom admin role to an administrator who will maintain that application.

More granular administrative roles such as *business_rule_admin*, *client_script_admin*, *script_include_admin*, *ui_policy_admin*, grant specific access rights without granting the broader privileges of the *admin* role. For example, an administrator can grant a user rights to change UI Policies, but not the rights to edit Client Scripts.



IMPORTANT

Specific admin roles do not change the access level and behavior of the *admin* role, which grants general administrative privileges across the platform.

Where Do Scripts Execute?

now.

Client side (browser)

- Auto-populate a field based on the value of another field
- Show/hide form sections



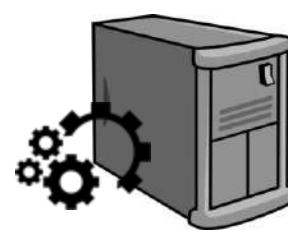
Server side

- Modify a database record
- Generate an event



On a MID server

- Integrate to a 3rd party application



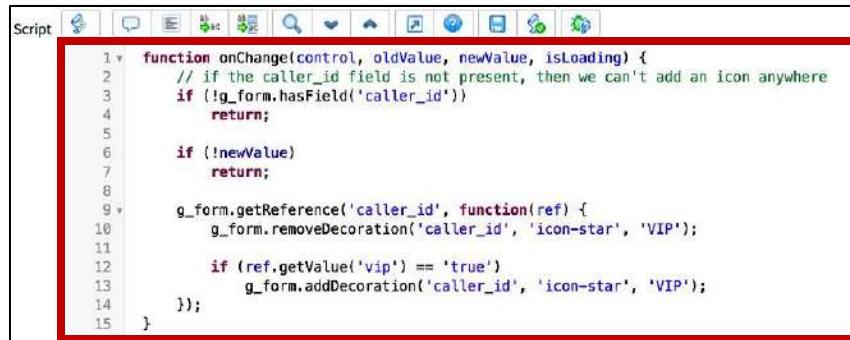
Where a script executes matters:

- Performance considerations.
- Access to objects and methods:
 - Client-side scripts have access to data on forms and in lists.
 - Server-side scripts have access to database records.

How Do You Script in ServiceNow?

now.

- The **Syntax Editor** is ServiceNow's built-in text editor
- Offers the following features as you script
 - Automatic JavaScript syntax coloring, auto-indentation, line numbers, and creation of closing braces and quotes
 - Context-sensitive help
 - Code editing functions
 - Editor macros for typing commonly used code
 - Syntax error checking



A screenshot of the ServiceNow Syntax Editor interface. The title bar says "Script". The editor window contains the following JavaScript code:

```
1 v
function onChange(control, oldValue, newValue, isLoading) {
    // if the caller_id field is not present, then we can't add an icon anywhere
    if (!g_form.hasField('caller_id'))
        return;

    if (!newValue)
        return;

    g_form.getReference('caller_id', function(ref) {
        g_form.removeDecoration('caller_id', 'icon-star', 'VIP');

        if (ref.getValue('vip') == 'true')
            g_form.addDecoration('caller_id', 'icon-star', 'VIP');
    });
}
15
```

The Syntax Editor is enabled baseline for new instances. For upgraded instances, a System Administrator must activate the **Syntax Editor [com.glide.syntax_editor]** plugin.

Scripting Overview: Topics

now.

Who, What, When, Where, Why and How?

The Syntax Editor

Locate Your Scripts Quickly

Application Scope

JavaScript in ServiceNow

Scripting Resources

The Syntax Editor: Syntax Coloring

now.

- Applies color coding to scripts for readability
 - **Green** = Comments
 - **Purple** = JavaScript commands
 - **Blue** = Strings, Reserved words

```
1+ function onLoad() {
2+   /**
3+    * For "maintenance" type schedules, locate the option for floating (value will be null)
4+    * as it is not a valid choice for this type of schedule. Also, hide the "type" and "pa
5+    * simplify the form.
6+    */
7+   var scheduleType = g_form.getElement("type");
8+   if (!scheduleType)
9+     return; // if it's not on the form, we cannot check it
10+
11  if (scheduleType.value != "maintenance")
12    return;
13
14  g_form.setDisplay("type", false);
15  g_form.setDisplay("parent", false);
16
17  var timeZoneSelect = g_form.getElement("time_zone");
18  if (!timeZoneSelect)
19    return; // if it's not on the form, we cannot change it
20
21  g_form.removeOption('time_zone','');
22}
```

Syntax coloring improves the readability of a script.

Comments will help you, future you, coworkers, and the next administrator understand the code.
Comments will not affect the code's execution.

JavaScript commands tell the platform what to do.

When writing scripts, you cannot choose a JavaScript reserved word as a variable name.

The Syntax Editor: Behaviors of Braces and Quotes

now.

- Closing parentheses, braces, and quotes automatically inserted as you type
- Matching character is highlighted when the cursor is beside a parenthesis, brace, or quote

```
1 v  function onChange(control, oldValue, newValue, isLoading, isTemplate) {  
2 v    if (isLoading || newValue == '') {  
3 v      return;  
4 v    }  
5 v  
6 v    var tableLabel = g_form.getValue('label'); ←  
7 v    autoFillField('user_role', generateUniqueCodeName(tableLabel, 'sys_user_role') + '_use  
8 v  
9 v  
10 v   function autoFillField(targetField, value, dynamic) { ←  
11 v     var targetDisplay = g_form.getDisplayBox(targetField);  
12 v     var targetValue = targetDisplay.value;  
13 v     if (targetValue && targetValue != '') {  
14 v       // Target is already set. We won't change it automatically here.  
15 v       return;  
16 v     }  
17 v  
18 v     g_form.setValue(targetField, '', value);  
19 v  
20 v     if (dynamic) {  
21 v       targetDisplay.title = new GwtMessage().getMessage("A new record with this value w:  
22 v       addClassName(targetDisplay, "ref_dynamic");  
23 v     }  
24 v   } ←
```

Locating matching characters is especially useful for debugging.



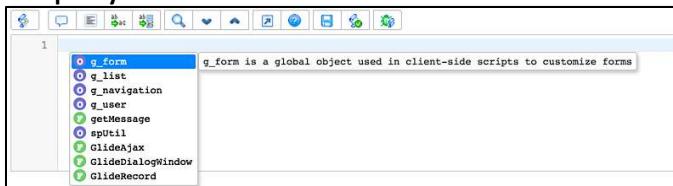
TIP FROM THE FIELD

If you do not have a corresponding parentheses, bracket, or quote, you may get unexpected results. Use the red highlights when you put your cursor next to one of these characters to help debug an issue. Did a closing character (parentheses, bracket, or quote) get highlighted when you put your cursor next to an open character? Did the correct open character highlight when you put the cursor next to the close character?

The Syntax Editor: Context-Sensitive Help

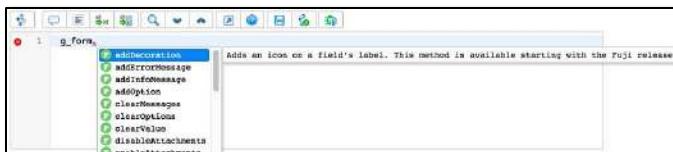
now.

- Displays a list of valid elements at the cursor's current position



Control + Spacebar at the beginning of a line

- Lists methods for a class



Period after a valid class name

- Lists expected parameters



Open parenthesis after a valid class, function, or method name

While editing a script, **Context-Sensitive Help** can be invoked by using the following keystrokes at the cursor location:

- Ctrl + Space (*both Mac/Windows*)
- . (*a period*)

Suggestions are context-sensitive relative to the cursor's current position and are filtered based on API type. For example, when working on a server-side script, only suggestions from server-side APIs display. When working on a client-side script, only suggestions from client-side APIs display.

Use any one of these strategies to insert a suggested element into your script.

- Highlight an element in the list using the arrow keys on your keyboard, then press the <Tab> or <Enter> key.
- Continue typing until the element becomes highlighted, then press the <Tab> or <Enter> key.
- Double-click an element in the list.

The Syntax Editor: Context-Sensitive Help

now.

- Properties assigned to a locally declared object are also included in the list of suggestions



Locally declared objects are only available to the current script.



IMPORTANT

- If the keyboard language is different from the instance language, you may experience unexpected behavior.
- Context-Sensitive Help is disabled for XML fields and UI Actions.
- Suggestions provided by Context-Sensitive Help are part of the Scoped Application API. If an element does not appear in the list of suggestions, simply continue typing the element's name. Context-Sensitive Help turns off when the text being entered no longer matches any suggestions in the list.

The Syntax Editor: Code Editing Functions

now.

- Syntax Editor toolbar provides standard JavaScript code editing functions
 - Toggle the Syntax Editor on/off
 - Comment/uncomment selected code
 - Code formatting
 - Replace
 - Replace All
 - Search
 - Display full screen editing mode
 - Keyboard shortcut help screen
 - Save
 - Syntax error checking
 - Script debugger

Script

Replace (Cmd-E)



Hover over an icon to see its keyboard shortcut.

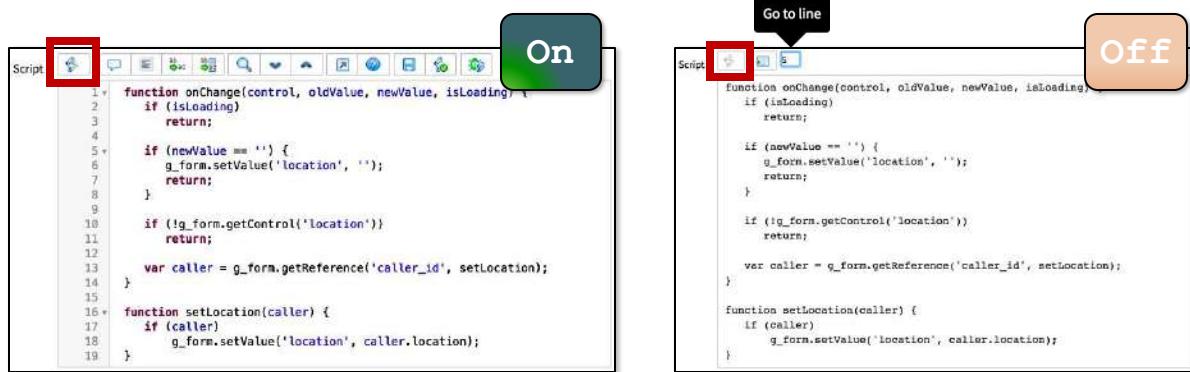
Terms and Definitions:

- Toggle Syntax Editor** – enables/disables the Syntax Editor.
- Comment Selected Code** – comments/uncomments out the selected code.
- Format Code** – applies the proper indentation to the script. Use the shortcut **Ctrl/Cmd A, Shift + Tab** to quickly format script.
- Replace** – replaces the next occurrence of a string in the script. Use the *Find Next* and *Find Previous* icons to jump to other occurrences of the string in the script. Confirmation will be needed.
- Replace All** – replaces all occurrences of a string in the script. No confirmation is needed.
- Search** – highlights all occurrences of a search term in the script. Locates the first occurrence.
- Toggle Full Screen Mode** – expands/contracts the script field to use the full form view for easier editing.
- Help** – displays the keyboard shortcuts help screen.
- Save** – saves changes without leaving the current view.
- Check Syntax** – checks the code for syntax errors.
- Script Debugger** – opens the Script Debugger in a new browser.

The Syntax Editor: Enable/Disable The Syntax Editor

now.

- Syntax Editor can be disabled by users preferring to develop in a basic text field
- Select **Toggle Syntax Editor** to enable/disable the Syntax Editor
- Choice is stored as a user preference



When the Syntax Editor is off, a **Go to line** button appears in the toolbar. To use the button:

1. Select it.
2. Enter a line number.
3. Press <Enter> on your keyboard. The cursor will appear at the beginning of that line in the script.

System Administrators can disable the editor for all users regardless of user preference by setting the *glide.ui.javascript_editor* property to **false**.

The Syntax Editor: Comment/Uncomment Selected Code

now.



Well-documented code is as important as having your code working!

To uncomment code:

1. Highlight the code to be made active.
2. Select the **Toggle Comment** button. The forward slashes at the beginning of each line of code will disappear.

Multiple lines of code can also be commented out by enclosing the selected block with the /* and */ characters.

The Syntax Editor: Format Code

now.

- Apply JavaScript standard indentation to your script

The screenshot shows two side-by-side code editors. The left editor has red annotations: a red arrow pointing up from the bottom of the code area with the text "Script not formatted", and a red arrow pointing down from the toolbar to the right with the text "Standard indenting applied after selecting Format Code". The right editor shows the same code with standard indentation applied. Both editors have identical toolbars at the top.

```
1 v function onChange(control, oldValue, newValue, isLoading) {  
2 v     var callerLabel = $('#label_incident.caller_id');  
3 v     var callerField = $('#sys_display.incident.caller_id');  
4 v     if (!callerLabel || !callerField)  
5 v         return;  
6 v  
7 v     if (!newValue) {  
8 v         callerLabel.setStyle({backgroundImage: ""});  
9 v         callerField.setStyle({color: ""});  
10 v        return;  
11 v    }  
12 v    g_form.getReference('caller_id', vipCallerCallback);  
13 v}
```

```
1 v function onChange(control, oldValue, newValue, isLoading) {  
2 v     var callerLabel = $('#label_incident.caller_id');  
3 v     var callerField = $('#sys_display.incident.caller_id');  
4 v     if (!callerLabel || !callerField)  
5 v         return;  
6 v  
7 v     if (!newValue) {  
8 v         callerLabel.setStyle({backgroundImage: ""});  
9 v         callerField.setStyle({color: ""});  
10 v        return;  
11 v    }  
12 v    g_form.getReference('caller_id', vipCallerCallback);  
13 v}
```

The Syntax Editor applies auto-indentation to code as it is written. Select the **Format Code** icon to re-apply indentation if needed.

Keyboard commands can also be used to quickly format your code. (*Can be used in all scripting fields across the platform.*)

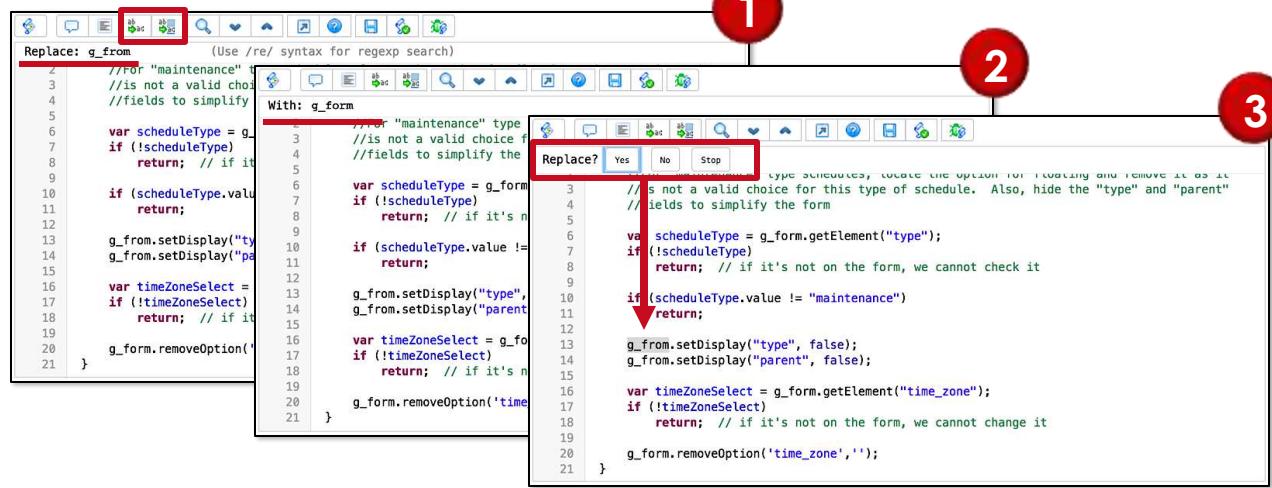
1. Ctrl-A/Cmd-A (*select all*)
2. Shift-Tab (*format code*)

Applying standard formatting is another strategy used to improve readability.

The Syntax Editor: Replace / Replace All

now.

- **Replace** a string with another string
- **Replace All** will **NOT** ask for confirmation



To replace a string in the script using the **Replace** button:

1. Select the **Replace** button.
2. Enter the text to replace in the Replace field and press the <Enter> key.
3. Enter the replacement text in the With field and press the <Enter> key.
4. Select **Yes** to replace, or **No** to skip for each match. Select **Stop** to discontinue the replacement process.

To replace a string in the script using the **Replace All** button:

1. Select the **Replace All** button.
2. Enter the text to replace in the Replace field and press the <Enter> key.
3. Enter the replacement text in the With field and press the <Enter> key.

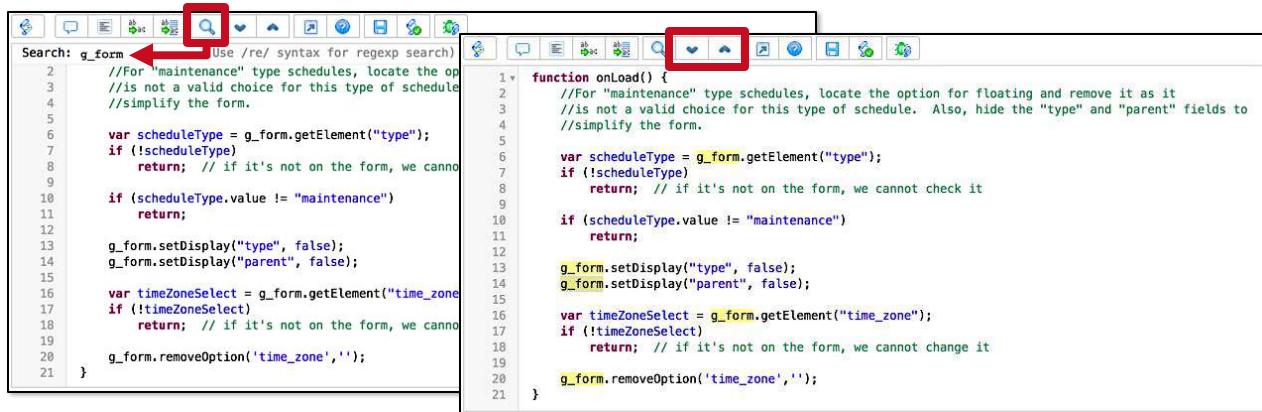
The *Replace* and *Replace All* features can also search for text specified with a regular expression.

Regular expressions must be bracketed by the '/' character. Example, the regular expression /g_[a-r] {4}/ would also find the string g_form.

The Syntax Editor: Search

now.

- Search for strings or regular expressions
- Every occurrence of the search term is highlighted
- Use **Find Next** and **Find Previous** to move between occurrences



To find a specific string in the Syntax Editor:

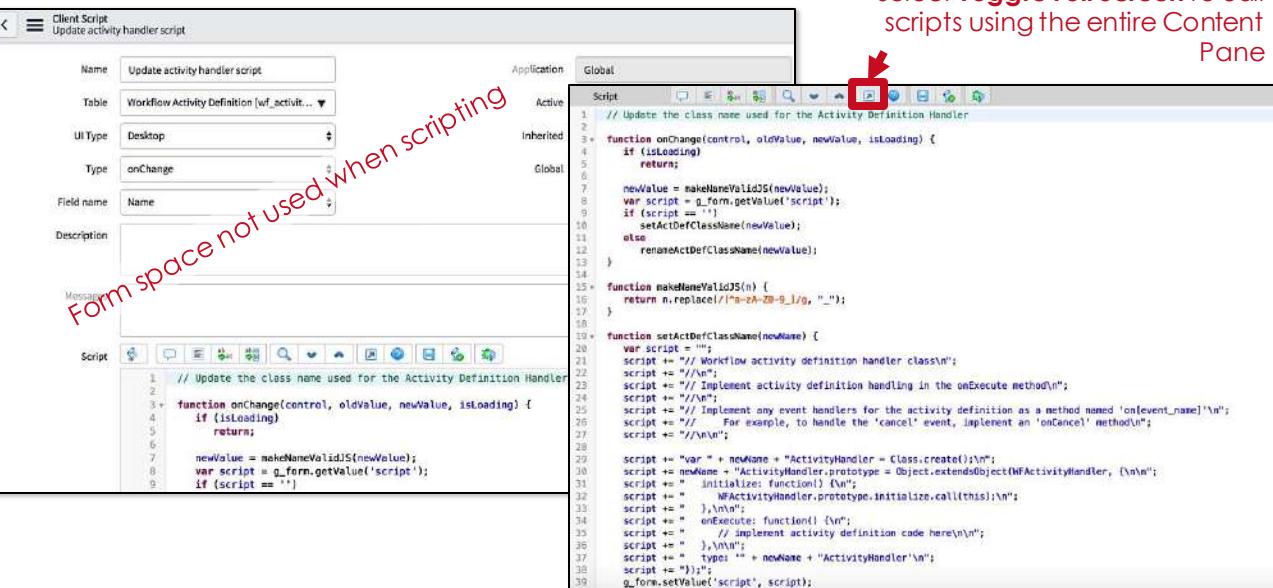
1. Select the **Search** button.
2. Enter the text to search for in the Search field.
3. Press the <Enter> key on the keyboard.

The *Search* feature can also locate text specified with a regular expression. Regular expressions must be bracketed by the '/' character. Example, the following regular expression /g_[a-r]{4}/, would also find the string g_form.

The Syntax Editor: Full Screen Editing

now.

Select **Toggle Full Screen** to edit scripts using the entire Content Pane



Every script has two parts:

1. Trigger (when to execute).
2. Script (what to do).

Once configured, the trigger seldom changes whereas scripts under development change frequently.

To have more space for the script, select the **Toggle Full Screen Mode** button to maximize the Syntax Editor. Select the button again to restore the Syntax Editor to its default size.

The Syntax Editor: Syntax Editor Macros

now.

- Provide shortcuts for commonly used code
- Insert a macro by typing the macro name and then pressing the <Tab> key
- Macro name is replaced with the full macro text
- Type the word **help** followed by the <Tab> key to see list of available macros

```
1 The Syntax Editor macros are:  
2 -----  
3 doc - Documentation Header  
4 for - Standard loop for arrays  
5 vargor - Example GlideRecord Or Query  
6 info -  
7 method - Standard JavaScript Class Method  
8 vargr - A common pattern of creating and querying a GlideRecord
```

For example, in the Syntax Editor, typing the word **for** followed by pressing the <Tab> key on your keyboard results in the following insertion at the cursor's position:

```
for (var i=0; i< myArray.length; i++) {  
    //myArray[i];  
}
```

The Syntax Editor: Creating New Syntax Editor Macros

now.

The screenshot shows the 'Editor Macro' creation screen. The 'Name' field contains 'object'. The 'Application' field is set to 'Global'. The 'Comments' field contains 'Create a new JavaScript object'. The 'Text' field contains the following code:

```
var objNameHere{  
    property1: value,  
    property2: value  
};
```

Below the text field, a help output window displays the following content:

```
1 The Syntax Editor macros are:  
2  
3 object - Create a new JavaScript object  
4 doc - Documentation Header  
5 for - Standard loop for arrays  
6 vargror - Example GlideRecord Or Query  
7 info -  
8 method - Standard JavaScript Class Method  
9 vargr - A common pattern of creating and querying a GlideRecord
```

A red arrow points from the 'Comments' field to the 'object' entry in the help output, with the annotation: 'Value in the Comments field used by the help macro'.

Navigate to **System Definition > Syntax Editor Macros** to create new or edit existing Editor Macros.

Name – macro keyword users type to insert the macro text. (Do not use special characters like period and quotes)

Application – identifies the scope of the macro.

Comments – provide a description of the macro. Will be used in the *help* macro output.

Text – full macro text that replaces the name in the editor when the macro is used.



IMPORTANT

Notice the Syntax Editor is not available for the Text field. You are responsible for the accuracy of your macros.

The Syntax Editor: Syntax Checker

now.

- The Syntax Checker finds basic JavaScript errors
 - Missing characters such as [and]
 - Missing ; at the end of JavaScript statements
 - Incomplete arguments in for loops
 - Bad function calls
- Does not find
 - Typos in variable names
 - Typos in function calls
 - Typos in method calls
- Cannot determine if the script works as expected

As with all programming editors and tools, the Syntax Checker cannot find all errors in a script.

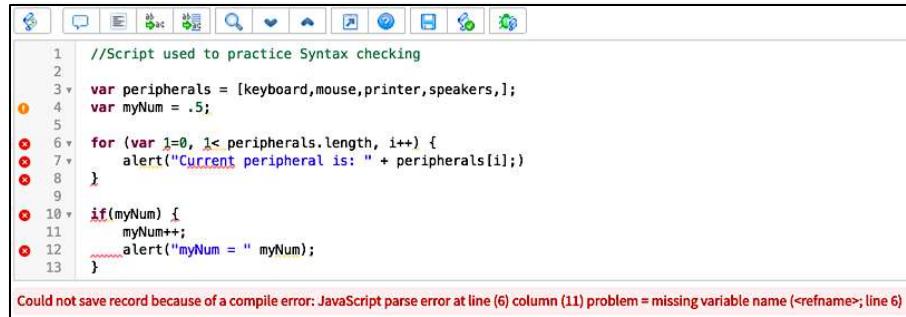
Any valid JavaScript will pass the Syntax Check even if the code does not do what you intended.

Note: Some script types will save with syntax errors (Workflow scripts) and others will not.

The Syntax Editor: Syntax Error Checking

now.

- Syntax is automatically checked as you type
 - Red circles and red underlines indicate errors
 - Orange circles and yellow underlines indicate warnings
 - Hover over warnings and errors for more details
- Error message appears below the field when a save is attempted on a script with errors



A screenshot of the ServiceNow Syntax Editor interface. The code area contains the following JavaScript:

```
//Script used to practice Syntax checking
var peripherals = [keyboard,mouse,printer,speakers,];
var myNum = .5;
for (var i=0, i< peripherals.length, i++) {
    alert("Current peripheral is: " + peripherals[i]);
}
if(myNum) {
    myNum++;
    alert("myNum = " myNum);
}
```

Red circles and underlines highlight several syntax errors: line 6 has a red circle under 'i<' and a red underline under 'i'; line 10 has a red circle under 'if(' and a red underline under 'myNum'; line 12 has a red circle under 'alert("myNum = ' and a red underline under 'myNum');'. A status bar at the bottom displays the error message: "Could not save record because of a compile error: JavaScript parse error at line (6) column (11) problem = missing variable name (<refname>; line 6)".

The system automatically checks for syntax errors as you type in any script field. To view error or warning details, hover over the red or yellow circles in the sidebar beside the line number, or the underlined script in the Syntax Editor.

If you cannot locate an error in the statement called out by the Syntax Checker, look at the preceding line(s) of code.

All errors must be corrected before a script can be saved.

This feature can be turned off by selecting the **Toggle Syntax Check** icon.

Scripting Overview: Topics

now.

Who, What, When, Where, Why and How?

The Syntax Editor

Locate Your Scripts Quickly

Application Scope

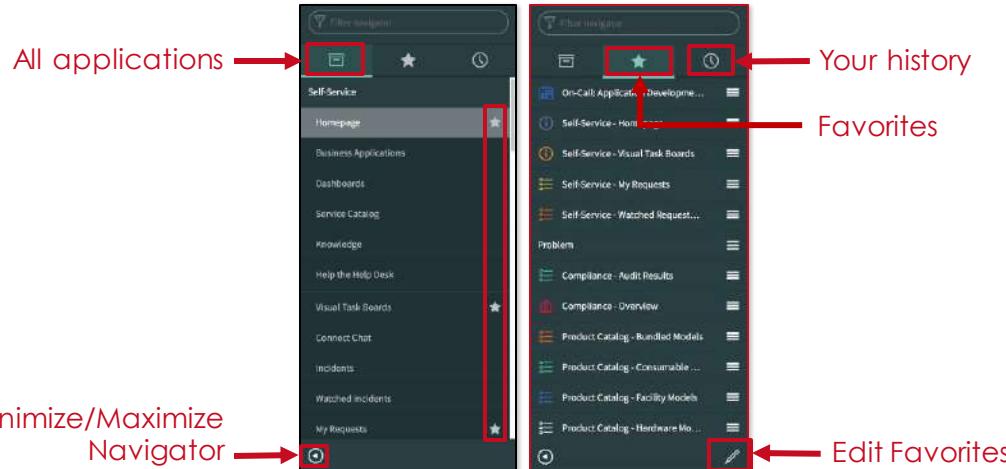
JavaScript in ServiceNow

Scripting Resources

Locate Your Scripts Quickly: Application Navigator

now.

- Use the Application Navigator to quickly locate your scripts
 - Select the star icon to the right of an Application or Module to identify it as a **Favorite**
 - Browse **Your history** to locate recently opened scripts



The Application Navigator provides links to all applications and modules in the ServiceNow platform.

- **All applications** – all applications and modules are displayed. Double-click the **All applications** icon for expand/collapse all functionality.
- **Favorites** – only items identified as favorites are displayed. Favorites are configured on a per user basis. When the star icon to the right of an application is selected, all the modules under the application are listed as favorites.
- **Your history** – list of most recently accessed items. System Administrators can increase/decrease the number of records pulled by setting the value of the `glide.ui.nav.history_length` system property. Baseline, the initial number of records is 30.

Select the **Minimize/Maximize Navigator** button to collapse or expand the Application Navigator. Hover over the icons to display an item's full name when the Navigator is minimized.

Select **Edit Favorites** to customize items in the Favorites list.



TIPS FROM THE FIELD

- Save a record as a Favorite by selecting **Create Favorite** on the record's Context menu or by dragging the record from a list to your list of favorites.
- Save a list as a Favorite by selecting **Create Favorite** on the list's Context menu or by dragging a breadcrumb condition to your list of favorites.
- If you forget to create a favorite, check **Your history** for recently accessed items.
- Add the **Updated** column to a list to sort the list by the most recently edited scripts.
- Add the **Updated by** column to a list to easily find scripts updated by a specific person.
- Search for the element by 'Name' using the **Go to** search field at the top of a list.
 - `mySearchString` (no leading or trailing characters) – does a "`>=`" search (like "starts with" but includes everything after the search string alphabetically also).
 - `*mySearchString` – does a "contains" search.
 - `mySearchString%` – does a "starts with" search.
 - `%mySearchString` – does an "ends with" search.

Locate Your Scripts Quickly: Navigate Directly to Table Configurations

now.

- Use commands in the Application Navigator's *Filter navigator* field to navigate directly to table elements
- Append **.config** after the table name to display all the configuration changes made to that table (e.g. **incident.config**)

| Name | Active | Table | Application | Order | Updated |
|------------------------------------|--------|-------------|-------------|-------|---------------------|
| Affected ci notifications | true | Task [task] | Global | 200 | 2018-10-25 14:46:50 |
| Affected cost center notifications | true | Task [task] | Global | 99 | 2018-11-14 13:44:28 |

A user can achieve the same result by selecting **Configure > All** from a form's Context Menu, but by using the command in the Application Navigator's *Filter navigator* field you do not have to be on that form to get to the lists of different configuration records.

Other available Application Navigator commands include:

- **<table_name>.list** – opens the list view of the table in the content pane.
- **<table_name>.LIST** – opens the list view of the table in a new window or tab.
- **<table_name>.form** or **<table_name>.do** – opens the form view of the table in the content pane.
- **<table_name>.FORM** – opens the form view of the table in a new window or tab.
- **<table_name>.CONFIG** – opens the configuration view of the table in a new window or tab.

Module Labs

now.

- **Lab 1.1**

- **Time:** 15-20m
- Using the Syntax Editor
 - Practice using the Syntax Editor to understand how scripts are recorded

- **Lab 1.2**

- **Time:** 5-10m
- Syntax Checking
 - Practice with the Syntax Checker to catch errors



Using the Syntax Editor

Lab
01.01
⌚15-20m

Lab Summary

You will achieve the following:

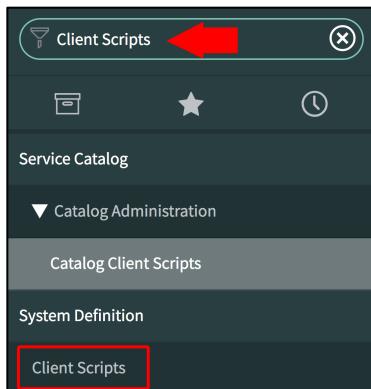
- Practice using the Syntax Editor in preparation for writing scripts in ServiceNow.

A. Preparation

1. Navigate to **Self-Service > Knowledge** on the Application Navigator.
2. Use the Search field to look for **Lab Answer Guide**.
3. Select the article, the file automatically downloads.
4. You are asked questions throughout the lab exercises to test your knowledge. Do your best to answer them on your own. Refer to the document if you require assistance or would like to simply confirm your answers.

B. Use the Syntax Editor

1. Type **Client Script** in the Application Navigator's *Filter navigator* field.
2. Select the **System Definition > Client Scripts** module.



TIP FROM
THE FIELD:

Typing the whole name of a module every time to open it can get tiresome. Identify and type the unique letters of a module name in the filter. Try **nt sc**. Did **Client Scripts** show up in the menu?

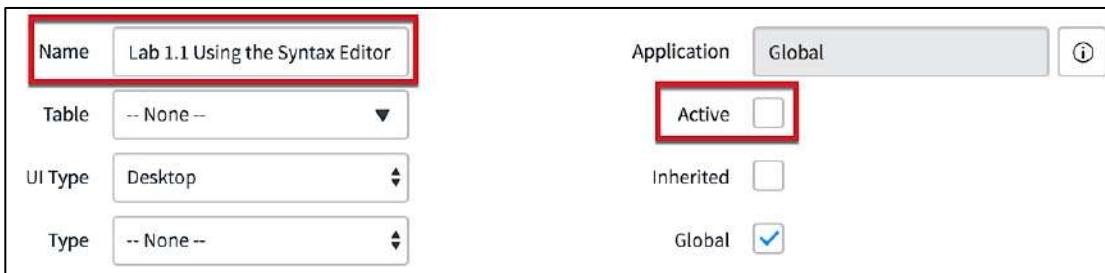
3. Select the **New** button ().

Note: When you create a Client Script, you will see this message:

New client-scripts are run in strict mode, with direct DOM access disabled. Access to jQuery, prototype and the window object are likewise disabled. To disable this on a per-script basis, configure this form and add the "Isolate script" field. To disable this feature for all new globally-scoped client-side scripts set the system property "glide.script.block.client.globals" to false.

This feature disables items that can cause issues with the platform. You can use these scripting techniques by following the instructions in the message. You should not need to make this change for this course.

4. Enter **Lab 1.1 Using the Syntax Editor** in the **Name** field.
5. De-select the **Active** checkbox to prevent the script from running.



The screenshot shows the 'Create Client Script' dialog. The 'Name' field contains 'Lab 1.1 Using the Syntax Editor' and is highlighted with a red box. The 'Active' checkbox, located in the top right section, is also highlighted with a red box and is unchecked. Other fields like 'Table', 'UI Type', 'Type', 'Application' (set to 'Global'), and 'Global' (with a checked checkbox) are visible but not highlighted.

6. In the Script field, enter the following text:

```
Code to test the Syntax Editor features
- Syntax coloring
- Special character highlighting
- Layout
- And more!
```

7. Convert the text to a comment.
- Select all of the text.
 - Select the **Toggle Comment** button () on the Syntax Editor toolbar

8. In the Script field, below the commented lines, enter the following script exactly as it appears here. As you enter the script, note the syntax coloring, special character highlighting, auto-indentation and the automatic creation of closing braces and quotes.

```
var myString = "Hello World";
var myNum = 32;
var myArray = ["Smartphone", "Tablet", "Laptop"];

var myObj = {
    property1: "first",
    property2: "second",
    property3: "Third"
};

for (var i=0; i< myArray.length; i++) {
    alert("The current value of myArray is: " + myArray[i]);
}
```

9. Save the script by selecting the **Save** button () on the Syntax Editor toolbar

10. Turn off syntax highlighting by selecting the **Toggle Syntax Editor** button ().
What differences do you notice in the Syntax Editor?

11. Select the same icon to turn syntax highlighting back on.

12. Use the **Replace** feature to change the number 2 in myObj.property2 to **myObj.propertyTwo**.

- Select the **Replace** button ().
- Type **2** in the *Replace* field then press **<enter>** on your keyboard.
- Type **Two** in the *With* field then press **<enter>** on your keyboard.
- Select **No** when prompted to replace the 2 in 32.
- Select **Yes** when prompted to replace the number 2 in property2 with the word Two.

13. Use the **Replace All** feature to change the value for myObj.property3 from Third to **third**.

- a) Select the **Replace All** button ().
- b) Type **Third** in the *Replace* field then press **<enter>** on your keyboard.
- c) Type **third** in the *With* field then press **<enter>** on your keyboard.

14. Select the **Update** button on the Header bar to save the script and exit the form.

C. Create a Syntax Editor Macro

1. Navigate to **System Definition > Syntax Editor Macros**.

2. Create a new macro:

Name: **try**

Comment: **Client-side try/catch**

Text:

```
try {  
}  
  
catch(err) {  
    g_form.addErrorMessage("A runtime error occurred: " + err);  
}
```

3. Select **Submit**.

4. Open the **Lab 1.1 Using the Syntax Editor Client Script**. Place the cursor on a new line below the existing script.

5. Test the 'try/catch' macro by typing **try** and pressing the **<tab>** key on your keyboard.

6. Did the macro display?

(If not, confirm the Name is correct in the Syntax Editor Macro record you created in step-2 and re-test.)



TIP FROM THE FIELD:

You can use the *Filter* navigator to locate the **Client Scripts** module. However, take a look at the **Your history** tab (). Is your Client Script available? This provides a direct link to the record and saves you some typing.

7. Place the cursor on a new line below the existing script, type **help** and press the **<tab>** key on your keyboard.

- a) Review the description for the **try** macro. Where did the value come from?
-

- b) Review the description for the **info** macro. Why is it blank?

Syntax

errors will prevent a save at this time, comment out or delete the **help** macro text.

8. Select **Update** to save the changes and close the record.

Challenge: *What Syntax Editor Macro records can you think of to create? What scripts do you write multiple times that you would want automated?*

Lab Completion

Good Job! You have practiced using some of the Syntax Editor features and have created and used Editor Macros.

Syntax Checking

Lab
01.02

5-10m

Lab Summary

You will achieve the following:

- Practice with the Syntax Checker to determine which types of errors the checker catches.

A. Syntax Checking in the Script Editor

1. Create a new Client Script.
2. Configure the trigger:

Name: **Lab 1.2 Syntax Checking**
Active: **Not selected (unchecked)**

3. Enter this script exactly as shown (including errors):

```
//Script used to practice Syntax Checking

var peripherals = [keyboard,mouse,printer,speakers,];
var myNum = .5;

for (var i=0, i < peripherals.length, i++) {
    alert("Current peripheral is: " + peripheral[i]);
}

if(myNum) {
    myNum++;
    alert("myNum" myNum);
}
```

4. Try to save your script. Did it save?
5. How can you tell?

6. Hover the mouse cursor over the orange (!) and red (✘) symbols to the left of the line numbers to read the warning and error messages
 7. Hover the mouse over the orange and red underlines in the script to read the warning and error messages.
 8. Correct the warnings and errors until no alerts remain.
 9. Did the Syntax Checker find all the errors in the script? Which error(s) were not found?
-
-
-

10. Select **Submit**.

Lab Completion

You have practiced Syntax Checking and understand not all errors are caught.

Scripting Overview: Topics

now.

Who, What, When, Where, Why and How?

The Syntax Editor

Locate Your Scripts Quickly

Application Scope

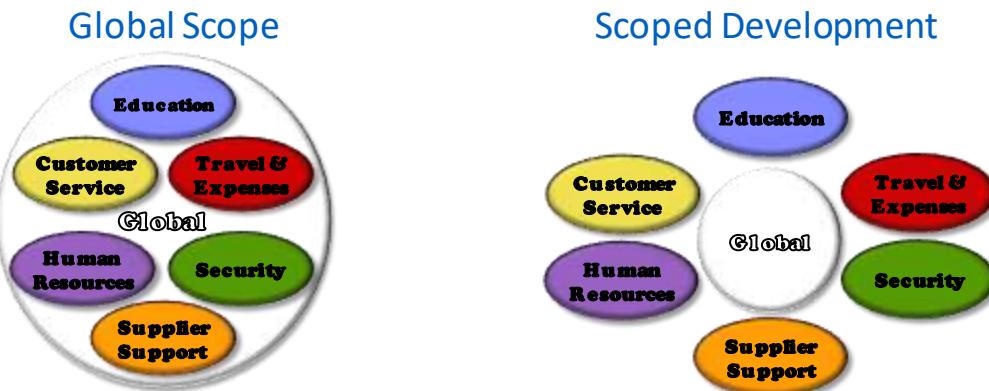
JavaScript in ServiceNow

Scripting Resources

Application Scope

now.

- Every application has a scope
- Determines which of its resources are available to other applications
- Once scope is assigned to an application, **it cannot be changed!**



Baseline applications provided by ServiceNow (*e.g. Incident, Service Catalog, Service Portal, etc...*), as well as any custom application built prior to the Fuji release are in the Global scope. It is difficult to protect/isolate application data in the Global scope.

There is no migration path to a custom or different scope.



TIP FROM THE FIELD

Turn on the **Show application picker in header** developer setting so you can easily switch between application scopes using a drop-down on the Header bar.

1. Select the **Settings** icon located on the right-side of the banner.
2. Select **Developer**.
3. Select **Show application picker in header**.
4. Use the **Application** drop-down list to switch between application scopes.

Application Scope

now.

- Scope protects an application and its artifacts from damage to, or from other applications
- Must be configured to grant other applications the ability to act on its records



Artifacts are the *application files* in an application. Examples include, but are not limited to, Tables, Access Controls, Email Notifications, Data Policies, Client Scripts, Business Rules, Script Includes, etc.

Application developers specify an application scope when they create a new application. They can also specify what parts of an application are accessible to other applications from:

- The Custom application record.
- Each application Table record.

For example, suppose you create a Travel & Expense Management application. By default, the application can access and change its own tables and business logic, but other applications in the platform cannot unless you grant them explicit permission to do so.

Application scope ensures:

- The application does not interrupt core business services.
- Other applications do not interfere with its normal functioning.

Application Scope: Scope Namespace Identifier

now.

- The system automatically prefixes a namespace identifier to scoped application artifacts (*including scripts*)
- Cannot be changed or removed to ensure they are always associated with the proper application

| Syntax | Description | Cloud Dimensions Example |
|----------------|--|--------------------------|
| | Scoped application artifacts always begin with x_ | x_ |
| Vendor Prefix | Unique ServiceNow generated prefix for each customer | cld |
| Application ID | Set when the application is first created | travel |
| Script name | Unique script name | ExpensesReqBy |

This example generates the namespace identifier: **x_cld_travel_ExpensesReqBy**

The application scope prevents naming conflicts and allows the contextual development environment to determine what changes, if any, are permitted.

Applications in the Global scope do not append a unique namespace identifier to the application name.

Application Scope: Updating Scripts in Another Scope

now.

- Out-of-scope scripts are read-only
- Select the **here** link to temporarily switch scopes

The screenshot shows a client script configuration page for a record named "Highlight VIP Caller". The application dropdown indicates "Global". A red arrow points to the "here" link in a tooltip message at the top of the page, which reads: "This record is in the Global application, but Benchmark Client is the current application. To edit this record click here."

| Name | Highlight VIP Caller | Application | Global |
|------------|----------------------|-------------|-------------------------------------|
| Table | Incident [incident] | Active | <input checked="" type="checkbox"/> |
| UI Type | Desktop | Inherited | <input type="checkbox"/> |
| Type | onChange | Global | <input checked="" type="checkbox"/> |
| Field name | caller_id | | |

Built-in limitations prevent developers from updating artifacts while in a different scope. This protects the application from inadvertent modifications.

Scripting Overview: Topics

Who, What, When, Where, Why and How?

The Syntax Editor

Locate Your Scripts Quickly

Application Scope

JavaScript in ServiceNow

Scripting Resources

- Provide classes and methods that can be used in scripts to define the functionality of the platform
- A few examples:

| Client-side Classes | Server-side Classes |
|---------------------|---------------------|
| GlideAjax | GlideAggregate |
| GlideForm | GlideDateTime |
| GlideList | GlideElement |
| GlideRecord | GlideRecord |
| GlideUser | GlideSystem |
| spModal | JSON |
| | Workflow |

Classes are grouped by those used for client-side scripts, REST APIs, global server scripts, and scoped server scripts.

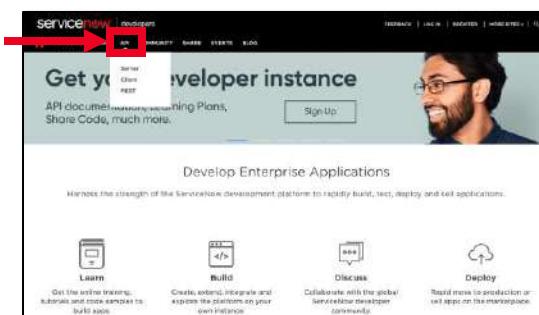
More classes are available for use in scripts and new ones are constantly being developed. It is important to stay current on what has changed from release to release.

- Visit ServiceNow's official **Developer** site to see the list of available API classes and methods, definitions on what they do, instructions on how to use them, as well as sample scripts written by ServiceNow developers.
- Visit ServiceNow's official **Documentation** site to review API release notes.

JavaScript in ServiceNow: API Documentation

now.

- The **API** section of ServiceNow's developer site (developer.servicenow.com) is the place to go for official API documentation
- Important to understand the server-side API category naming convention:
 - Scoped**: support for scoped applications, these APIs may behave differently in the Global scope
 - Legacy**: support for legacy applications in the Global scope



API Documentation is presented as Scoped or Legacy (Global) documentation. Understand what scope you are using, because you cannot call a global Glide API in a scoped application.

Another thing you want to consider is the version of your instance. Make sure the documentation you are looking at is the version of your instance.

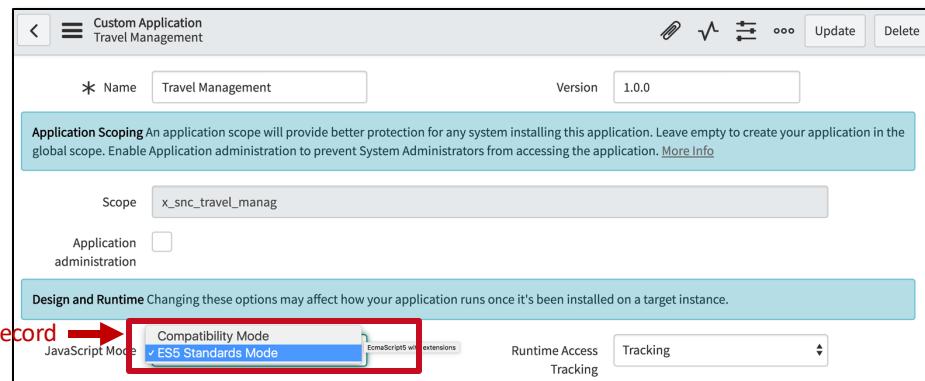
In the API documentation, you will find links to scope application, legacy application, and instance versions within the documentation.

A screenshot of the ServiceNow API documentation for the 'AbstractDBObject' class. The page has a dark header with the ServiceNow logo and a search bar. Below the header, there are two tabs: 'SCOPED' (which is highlighted in red) and 'LEGACY'. A red arrow points from the left towards the 'SCOPED' tab. On the right side of the page, there is a detailed description of the 'AbstractDBObject' class, its methods, and properties. A red arrow points from the right towards the 'Description' table. The table has columns for 'Type' and 'Description'. An example entry in the table is: 'Boolean True if the database record is valid, false otherwise.' At the bottom right of the page, there are buttons for 'ISTANBUL', 'JAKARTA', and 'KINGSTON' with a dropdown arrow.

JavaScript in ServiceNow: JavaScript Mode

now.

- The JavaScript engine ServiceNow uses to evaluate scripts supports the **ECMAScript5** scripting standard
- To support scripts written prior to the Helsinki release, the JavaScript engine has two modes
 - Compatibility Mode
 - ES5 Standards Mode (default)**



ES5 Standards Mode is the default mode when creating new scoped scripts. It supports ECMAScript5 syntax, extensions, and features, including:

- The 'use strict' declaration.
- Control over extensibility of objects.
- Get and set properties on objects.
- Control over write-ability, configurability, and innumerability of object properties.
- New Array and Date methods.
- Native JSON support.
- Support for modern third-party libraries such as lodash.js and moment.js.

Compatibility Mode supports ECMAScript3 with ServiceNow behaviors and extensions

- All Global scripts.
- All scripts developed prior to the *Helsinki* release.

Visit <https://en.wikipedia.org/wiki/ECMAScript> if you are interested in learning more about ECMAScript.

Scripting Overview: Topics

Who, What, When, Where, Why and How?

The Syntax Editor

Locate Your Scripts Quickly

Application Scope

JavaScript in ServiceNow

Scripting Resources

Scripting Resources: Where to Get Scripting Help

now.



- ServiceNow Scripting Resources
 - Community
 - ServiceNow User Groups (SNUG)
 - Product documentation
 - Developer documentation
 - Learning Library
- JavaScripting Resources
 - w3schools.com
 - codecademy.com
 - codeschool.com
 - developer.mozilla.org
 - lynda.com (requires a paid subscription)

There are many resources available for working with JavaScript.

Scripting Resources: ServiceNow Community

now.

Welcome to the Community

Search Content, Forums and People

Browse Community Forums

Service Management

Customer Service Management

IT Service Management

IT Operations Management

IT Business Management

Performance Analytics & Reporting

Now Platform

Governance, Risk and Compliance

HR Service Delivery

Security Operations

Software Asset Management

Developers

Architects

All Forums

All Topics

IT Business Management

Now Platforms

Performance Analytics and Reporting

Governance, Risk, and Compliance (GRC)

HR Service Delivery

Security Operations

Software Asset Management

Developer Community

community.servicenow.com

Membership in the ServiceNow community is free!

Select **LOG IN** at the top-right of the page to sign in or create an account.

With an account, you have access to all of these features:

- Post questions on our Forums.
- Comment on our Blogs.
- Rate content.
- Register for Local Events.
- Join Local User Groups.
- Get to know ServiceNow customers, partners, and employees.

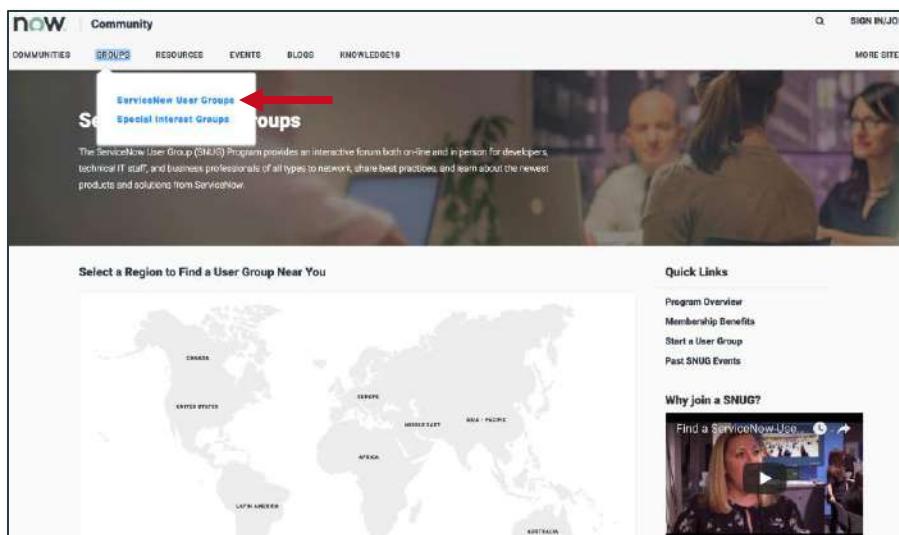


IMPORTANT

Get feedback not only from your internal sources, but from other customer sources as well.

Scripting Resources: ServiceNow User Groups (SNUG)

now.



Join a SNUG Near You!

Join a SNUG near you today. Attending an event is a fantastic opportunity to gain new knowledge and network with your peers that share a passion for ServiceNow. The benefits of attending a SNUG are numerous.

- Face-to-face interactions with ServiceNow subject matter experts, product evangelists, executives and partners.
- Knowledge transfer across diverse organizations, industries, and ServiceNow use cases.
- Connect with and learn from peers while developing professional relationships.
- Insight into upcoming product releases, roadmaps and company updates including sneak previews into new features and on-going product development.
- Help shape future product direction by providing real-time feedback.
- Access to live customer testimonials, demos and hands on workshops.
- Open forum to discuss ideas, industry trends, challenges and observations.

SNUGs are managed by ServiceNow users just like you. If there is no SNUG in your area, you can create one!

Scripting Resources: Product Documentation

now.

The screenshot shows the ServiceNow Product Documentation interface. At the top, there's a navigation bar with the ServiceNow logo, 'Product Documentation', 'More Sites', and 'Log in'. Below the navigation is a breadcrumb trail: Home > Madrid > Madrid Application Development > Now Platform Custom Business Applications > APIs and scripts > Scripts. To the right of the breadcrumb is a search bar and a red search icon. On the left, there's a sidebar titled 'Contents' with a tree view of the documentation structure, including 'Now Platform Custom Business Applications', 'Applications', 'Creating applications', 'Application management', 'Application tools', 'APIs and scripts' (which is expanded), and 'Extension points'. The main content area is titled 'Scripts' and includes a 'Madrid' tag. It describes what scripts are used for and provides technical details about APIs and Jelly scripts. A note at the bottom states: 'Note: When you are writing scripts, you cannot use reserved words'. There are also 'Subscribe' and '...' buttons.

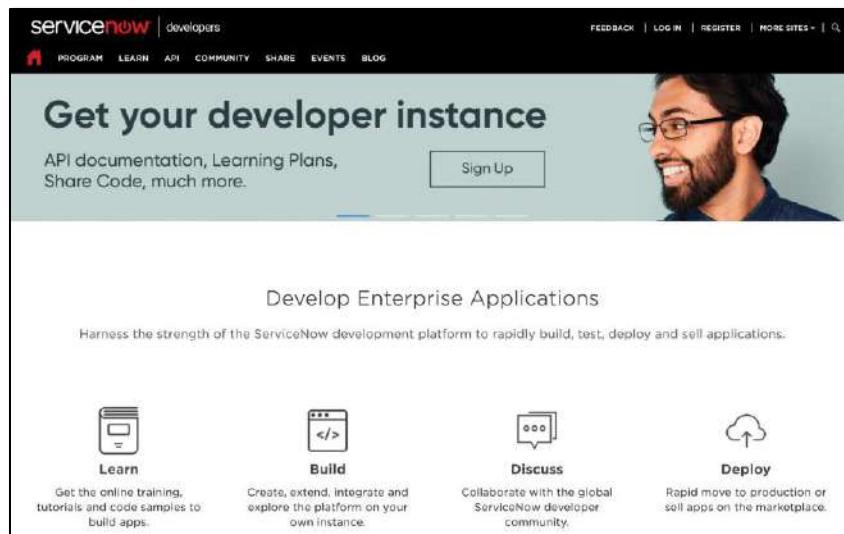
docs.servicenow.com

Product Documentation provides ServiceNow's official documentation about the platform. Use the **Search documentation** field to quickly navigate to the information you are looking for.

In addition to navigating directly to the website, you can also select the **Search Product Documentation** link from the Help icon on the banner frame in any instance.

Scripting Resources: Developer Documentation

now.



developer.servicenow.com

The **ServiceNow Developer Program** provides ServiceNow developers with a dedicated program of free content, technical resources, developer focused training, tools, events, and an online community for collaboration and sharing.

It is designed to maximize the productivity, effectiveness and efficiency of developers creating value-added applications and solutions on ServiceNow.

Register for a personal developer instance to practice building, extending and integrating new applications in a safe sand-box environment.

Scripting Resources: ServiceNow Learning Library

now.

The screenshot shows the ServiceNow Learning Library interface. On the left, a sidebar lists categories: CSM, ITBM, ITOM, ITSM, FULFILLERS, PLATFORM, REPORTING and PA, and SECURITY. The main area displays a grid of course thumbnails. The first row includes: 'Application Security Part 9 - Plugins and Mobile Security' by Dan Boena (0h 5m 49s, Beginner), 'Application Security Part 10 - Additional References' by Jason Lim (0h 30m 0s, All), and 'Event Management Module 3 - Service Mapping & Discovery' by Alex Darby (0h 45m 0s, Intermediate). The second row includes: 'Event Management Module 6 - CI Remediation' by Alex Darby (1h 10m 0s, Intermediate), 'ServiceNow Reporting' by Brisse Ramos (1h 0m 0s, All), and 'Module 2 - Importing in ServiceNow' by Glenn Pinto (1h 0m 0s, Intermediate). Below these rows, a message says 'Displaying courses 31 - 36 of 61 in total'. At the bottom, there are navigation links for 'Previous' and 'Next' pages, with the current page being 6. A 'Recently Played' section is also visible at the bottom.

<https://www.servicenowlearninglibrary.com/>

Learn on your own schedule, at your desired pace.

The ServiceNow Learning Library provides a cost-effective option to supplement our traditional training courses with 24/7 access to a wide variety of learning assets designed to enhance learning. Training ranges from short videos to detailed tutorials that can be paused at any point and continued later – allowing you to learn on a schedule that aligns with your availability. Mobile-ready content provides access at any time and from any location.

Module Labs

now.

- **Lab 1.3**

- **Time:** 15-20m
- Explore Scripting Resources
 - Explore the community website
 - Explore the documentation website
 - Explore the developer website



Explore Scripting Resources

Lab
01.03
⌚15-20m

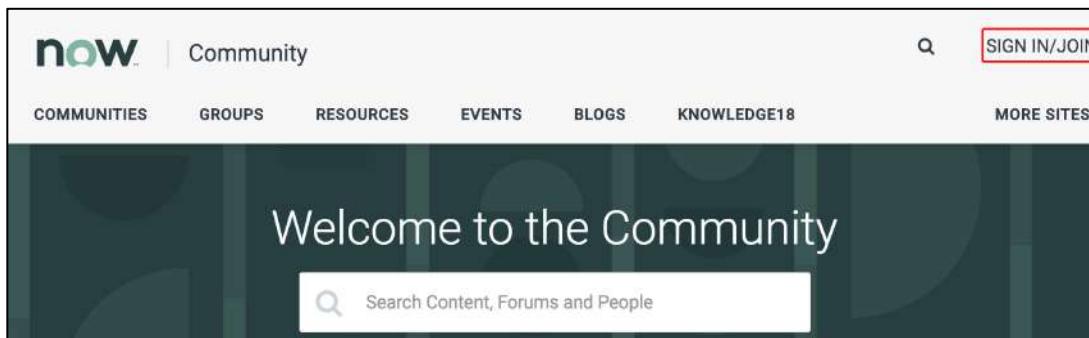
Lab Summary

You will achieve the following:

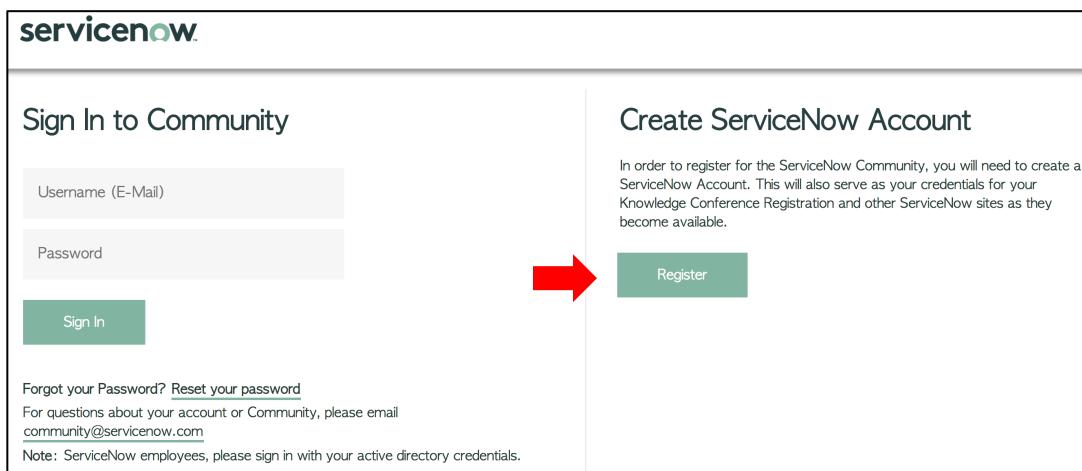
- In this lab, you will practice using various resources to get scripting help.

A. Explore Community

- In a web browser, navigate to community.servicenow.com.
- Sign in. If you do not have an account, create one by selecting the **SIGN IN/JOIN** link.



- Select the **Register** button. After requesting an account, continue with the lab.



4. Join the ServiceNow User Group (SNUG) nearest you.
 - a) From the Header bar, select **Groups > ServiceNow User Groups**.
 - b) Select a **Region** from the Map.

The ServiceNow User Group (SNUG) Program provides an interactive forum both on-line and in person for developers, technical IT staff, and business professionals of all types to network, share best practices, and learn about the newest products and solutions from ServiceNow.

Select a Region to Find a User Group Near You

Quick Links

- Program Overview
- Membership Benefits
- Become a Member
- Start a User Group
- Host a SNUG Meeting
- Roles and Responsibilities
- Past SNUG Events

Sponsor a SNUG!

Visit Your Success Center

Discover best practices for every phase of your ServiceNow journey.

[EXPLORE NOW](#)

- c) Select a **ServiceNow User Group** from the list that appears below the map.

United States User Groups

Search User Group

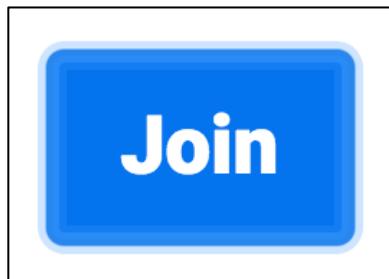
| | |
|--|--|
| | ServiceNow User Group - US - AL - Alabama |
| | ServiceNow User Group - US - AR - Arkansas |
| | ServiceNow User Group - US - AZ - Arizona |
| | ServiceNow User Group - US - CA - Bay Area |
| | ServiceNow User Group - US - CA - LA Area |

Find Expert Events on the Community

Browse, watch, and post questions! No registration required.

[BROWSE THE EVENTS](#)

- d) Select the **Request Access** button at the top-right corner of the page.



5. Where do you search for existing content in Community?



TIP FROM
THE FIELD:

*There are more regions and groups than shown here. Type a **country**, **state**, or **major city** followed by the words **user group** in the site's search box to find your user group.*

6. Search Community for the string **Ask Why**.

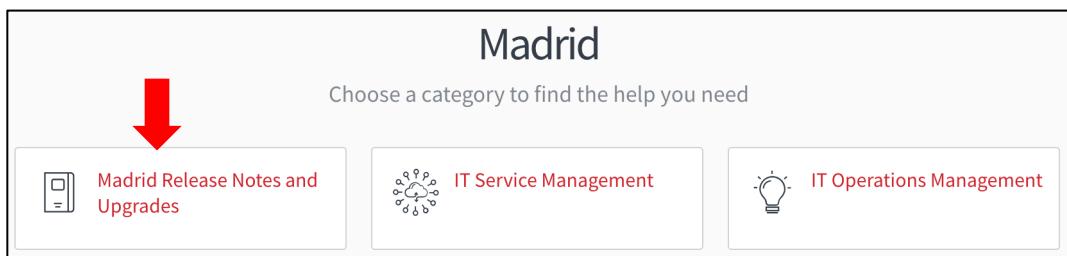
- Select the **Search (magnifying glass)** icon at the top-right corner of the page.
- Type **Ask Why** in the search field and press <Enter> on your keyboard.
- Open the article in **Chuck Tomasi's Blog**.

- d) Open and read the article.

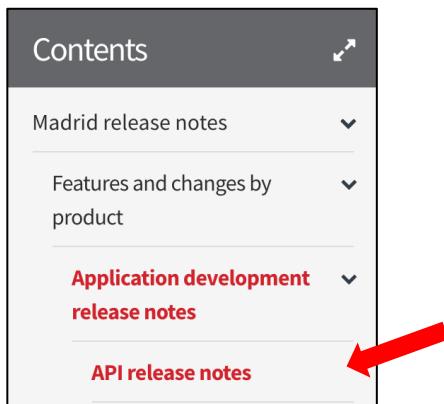
7. When you cannot find the information you are looking for in Community; how do you ask a question?
-
-

B. Explore Documentation

1. Navigate to ServiceNow's official Product Documentation website: docs.servicenow.com.
2. Select **Madrid**.
3. Open the **Madrid Release Notes and Upgrades** article.



4. On the left Navigation menu, expand the **Features and changes by product** category, expand the **Application development release notes**, and then expand the article titled **API release notes**.



Note: The **Now Platform release notes** category is where you locate information about new and updated features made to the platform itself, (this includes changes to Scripting as it is a function of the platform). The information and number of articles changes from release to release. This particular **API release notes** article is a good example of easy to locate information.

5. Review the article. What information is provided? Record your answer here:
-
-

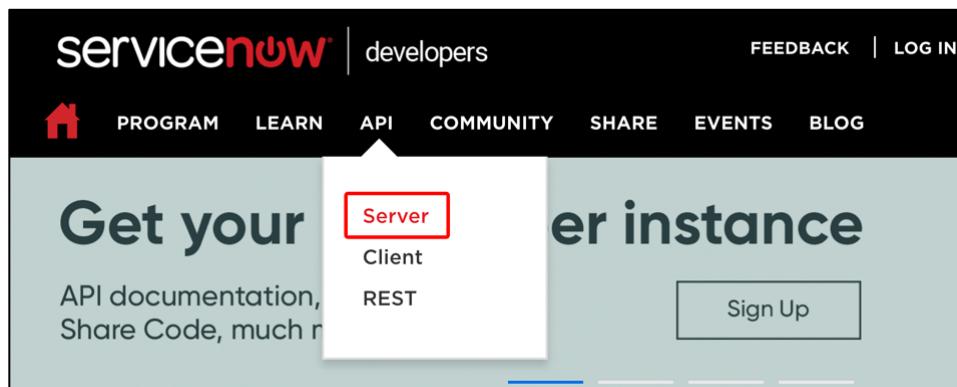
6. Use the **magnifying glass** to search Documentation for the string **Useful scripts**

- a) Type **Useful scripts** in the *Search* field at the top-right corner for the page and press <Enter> on your keyboard.
- b) Open the **Useful scripts** article.

Note: This article contains links to other articles offering scripts written by ServiceNow developers. You may want to favorite this webpage so you can explore it when you have more time.

C. Explore Developer

1. Select the **More Sites** drop-down menu, followed by **Developer Documentation** to navigate to ServiceNow's Developer website (developer.servicenow.com).
2. Select **API > Server**.



3. On the left Navigation menu, select **Legacy**.

Note: The **Scoped** menu lists documentation for applications in a private scope. These APIs may behave differently in the Global scope. The **Legacy** menu lists documentation for legacy applications in the Global scope.

4. Locate and expand the **GlideDateTime** category.

5. Select the **GlideDateTime()** method.

The screenshot shows the ServiceNow developers API documentation page. The search bar at the top contains the text "glidedatetime". On the left, there's a sidebar with categories like "SCOPE" and "LEGACY". Under "LEGACY", the "GlideDateTime" category is expanded, showing two sub-options: "GlideDateTime()" and "GlideDateTime(GlideDateTime gDT)". A red arrow points from the text "GlideDateTime()" to the "GlideDateTime()" option in the sidebar. The main content area displays the "GlideDateTime()" method documentation, which includes a brief description, an example code snippet, and another section for "GlideDateTime(GlideDateTime gDT)".

6. Review the documentation in the middle of the screen.
7. Notice the examples in each section.
8. What Date/Time format does the GlideDateTime object use?

Lab Completion

Good Job! You have really explored some of the ServiceNow resources that are at your fingertips when you need help scripting.

Scripting Fundamentals

now.

Good Practices

- Use a **Condition builder** whenever possible to configure your instance!!
- Create a **Syntax Editor Macro** to quickly insert commonly used syntax
- Expand the Script Editor when editing large scripts
- Identify the **Scope** of your script to ensure it only affects the resources in the same scope
- ServiceNow websites are more than just product documentation; make good use of the **Community**
- Join at least one SNUG

Module Recap: Scripting Overview

now.

Core Concepts

- Scripts can create new or extend the existing functionality of ServiceNow
- The answer is not always to create a new script
- Scripts execute in different locations
Client-side - Server-side - On a MID server
- ServiceNow's built-in Script Editor offers industry standard scripting capabilities
- Debug with the Syntax Checker
- Globally scoped scripts are shared resources
- Privately scoped scripts only affect resources in the same scope
- Know where to get scripting help

Real World Cases

- **Why** would you use these capabilities?
- **When** would you use these capabilities?
- **How often** would you use these capabilities?

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Module 2: Client Scripts

now.

| |
|-----------------------|
| Scripting Overview |
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Define what a Client Script is
- Know when to use a Client Script
- Write, test, and debug Client Scripts
- Use the `g_form` and `g_user` objects and methods
- Retrieve reference records from the database
- Compare and revert a script to a different version

Labs

- Lab 2.1 Two Simple Client Scripts
Lab 2.2 `g_form` and `g_user`
Lab 2.3 Debugging Client Scripts
Lab 2.4 Client Scripting with Reference Objects

Client Script Overview

Client-side APIs

Client-side Debugging

Reference Objects

Script Versions

Client Script Overview: What is a Client Script?

now.

- Client Scripts **manage the behavior** of forms, fields, and lists in real time
 - Make fields mandatory
 - Set one field in response to another
 - Modify choice list options
 - Hide/Show form sections
 - Display an alert
 - Hide fields
 - Prohibit list editing
- Client Scripts **execute client-side** (web browser)
 - Browsers may present items in different ways

“Well-designed Client Scripts can reduce the amount of time it takes users to complete a form.”
- ServiceNow Docs

Although modern browsers largely interpret JavaScript the same way, you may still observe browser-dependent behaviors in client-side scripts.

FAQ's

What Exists Baseline?

- More than 800 Client Scripts exist baseline
- Not all Client Scripts are *Active* baseline
- **Sample** Client Scripts can be used as starting points

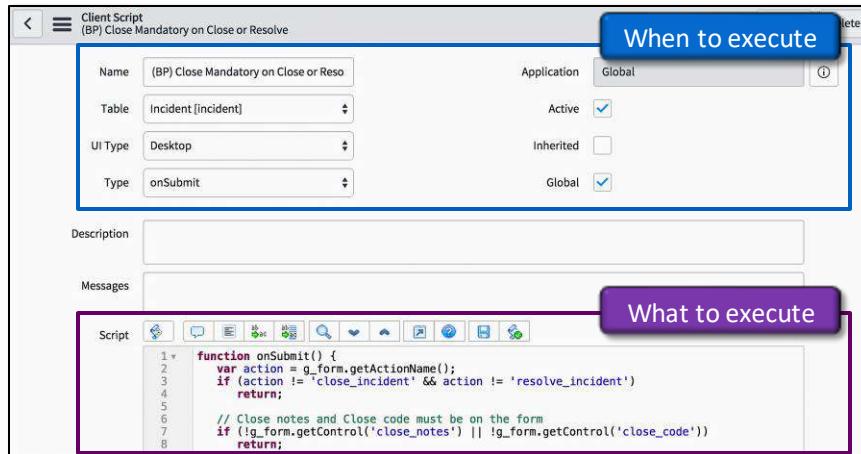
Where can this be found?

- Navigate to the **System Definition > Client Scripts** module to create or modify Client Scripts.

Client Script Overview: Client Script Execution

now.

- Trigger specified **when to execute**
- Scripts specifies **what to execute**



The **Description** field is for documenting the script. Include information like who wrote the script, what business requirement the script is for, and any other pertinent information.

The **Messages** field is used for internationalizing output to the user. For example, if the script creates an alert that says Hello World, the string "Hello World" would appear in the Messages field on its own line. If an entry exists in the *sys_ui_message* table with the same key but a localized language, the localized language version is presented to the user even though the script uses the version from the Messages field.

The **Script** field is used to script *what* needs to happen when the conditions in the trigger are met.

Client Script Overview: Client Script Trigger

now.

- Client Scripts execute when their trigger condition is met

| | | | | |
|---------|---------------------------|-------------|-------------------------------------|--|
| Name | (BP) Hide Choice - Closed | Application | Global | |
| Table | Incident [incident] | Active | <input checked="" type="checkbox"/> | |
| UI Type | Both | Inherited | <input type="checkbox"/> | |
| Type | onLoad | Global | <input checked="" type="checkbox"/> | |

Additional information is required if the Client Script is triggered by a field value change or if it applies to a specific view of the form

| | | | | |
|------------|----------------------|-------------|-------------------------------------|--|
| Name | Highlight VIP Caller | Application | Global | |
| Table | Incident [incident] | Active | <input checked="" type="checkbox"/> | |
| UI Type | Desktop | Inherited | <input type="checkbox"/> | |
| Type | onChange | Global | <input type="checkbox"/> | |
| Field name | Caller | View | ESS | |

While on the Client Scripts list, select **New** to create a new Client Script.

Name – name of the Client Script. Use a standard naming scheme to identify custom scripts.

Table – table the form or list is related to.

UI Type – select whether the script executes for *Desktop* only, *Mobile / Service Portal* only, or *All* environments.

Type – select when the script runs: *onChange*, *onLoad*, *onSubmit*, or *onCellEdit*.

Field name – used only if the script responds to a field value change (*onChange* or *onCellEdit*); name of the field to which the script applies.

Application – identifies the scope of the Client Script.

Active – if selected the script is executing in the runtime environment.

Inherited – execute the script for forms from any extending tables when selected.

Global – if Global is selected the script applies to all Views. If the Global field is not selected you must specify the View.

View – specifies the View to which the script applies. The View field is only visible when Global is not selected. A script can only act on fields part of the selected form View. If the View field is blank the script applies to the Default view.

Client Script Overview: Client Script Trigger – Order

now.

- Use the **Order** field when multiple Client Scripts for the same table have conflicting logic
- Executes in order from lowest to highest
- Does not appear on the Client Script form baseline
 - Configure the form to add the field

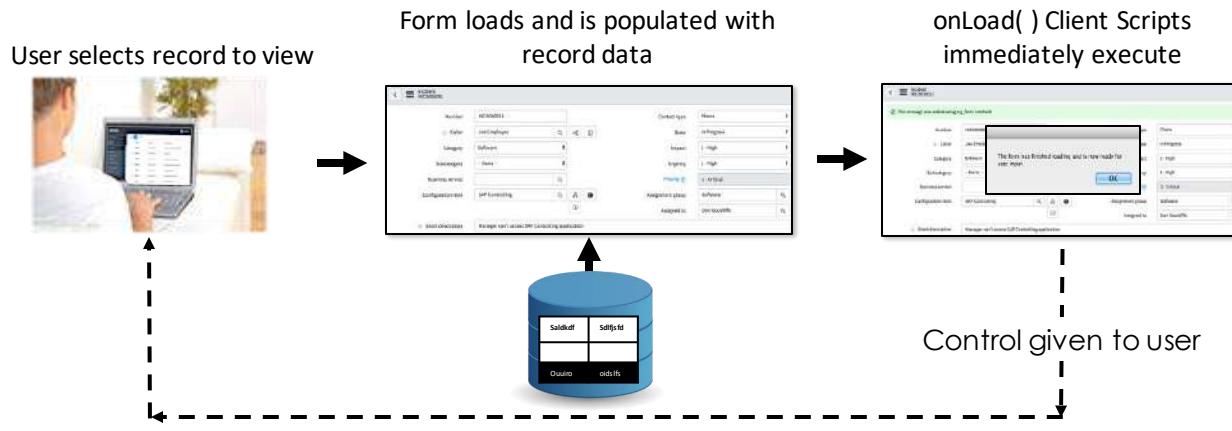
The screenshot shows the 'Client Script' configuration screen for a specific script named '(BP) Hide Choice - Closed'. The 'Table' is set to 'Incident [incident]'. The 'Type' is set to 'onLoad'. The 'Order' field is highlighted with a red border. Other fields shown include 'Name', 'Application', 'Active', 'Inherited', and 'Global'.

Order is the sequence in which the Client Scripts are executed, from lowest to highest. By convention, Order numbers contain three digits. For example: 150, 300, or 675.

Client Script Overview: Client Script Type – onLoad()

now.

- Script **runs when a form loads** and **before control is given to the user**
- Typically used to manipulate a form's appearance or content on screen

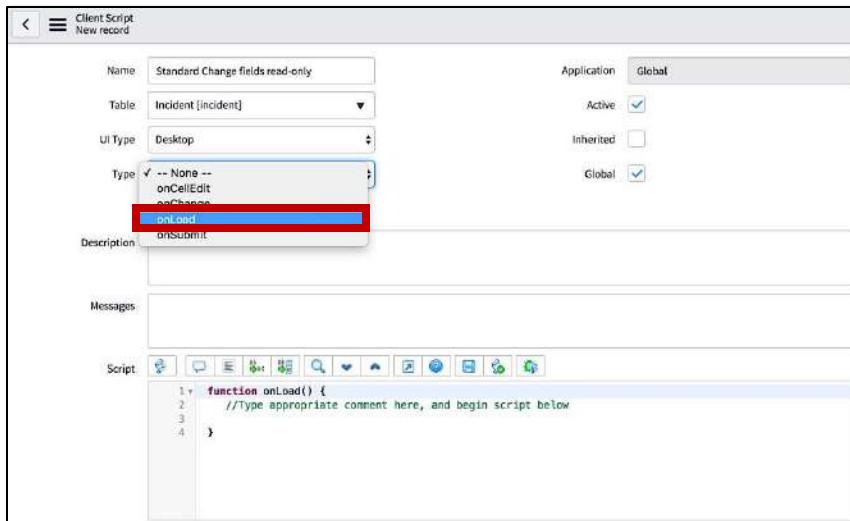


Users do not have the ability to modify form fields while an `onLoad()` Client Script executes.

Client Script Overview: Client Script Type – onLoad()

now.

- Select **onLoad** as Client Script Type



onLoad() function template automatically inserted in the Script

No arguments passed to it

For example, a baseline *onLoad()* Client Script on the Change form marks Standard Change fields read-only before control is given to the user.

Usually, *onLoad()* Client Scripts perform client-side-manipulation of the current form or set default record values.

Client Script Overview: Client Script Type – onSubmit()

now.

- Script **runs when a form is saved, updated, or submitted**
- Typically used for field validation

User saves, submits, or updates a record



onSubmit() Client Scripts execute



Record written to database



Script can prevent writing to database and return control to user

onLoad() Client Scripts run again and control is returned to the user

Users do not have the ability to modify form fields while an *onSubmit()* Client Script executes.

Client Script Overview: Client Script Type – onSubmit()

now.

- Select **onSubmit** as Client Script Type

The screenshot shows the 'Client Script' configuration page for a new record. The 'Type' dropdown is set to 'onSubmit'. The 'Script' pane contains a template for the onSubmit() function:

```
1+ function onSubmit() {
2+     //type appropriate comment here, and begin script below
3+
4+ }
```

onSubmit() function template automatically inserted in the Script

No arguments passed to it



TIP FROM THE FIELD

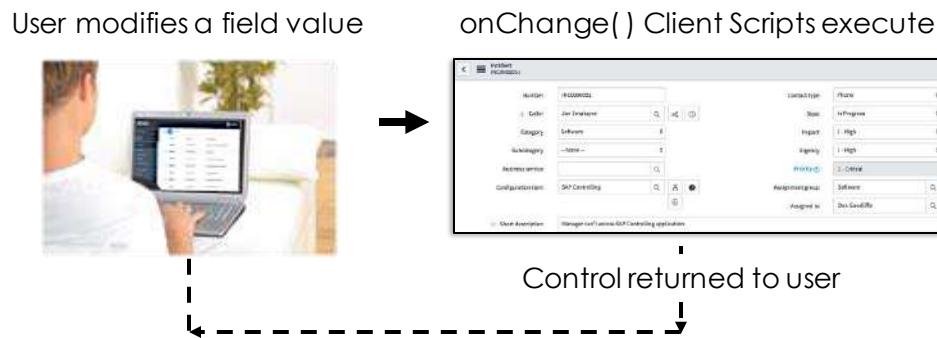
Typically, *onSubmit()* Client Scripts validate data on the form and ensure the submission makes sense. If the inputs the user submits do not make sense, an *onSubmit()* Client Script can cancel form submission by returning a value of **false**.

```
function onSubmit() {
    if(!myCondition) {
        return false;
    }
    else {
        //perform some logic here
    }
}
```

Client Script Overview: Client Script Type – onChange()

now.

- Script **runs when a particular field value changes**
- Typically used to
 - Respond to field values of interest
 - Modify one field value in response to another



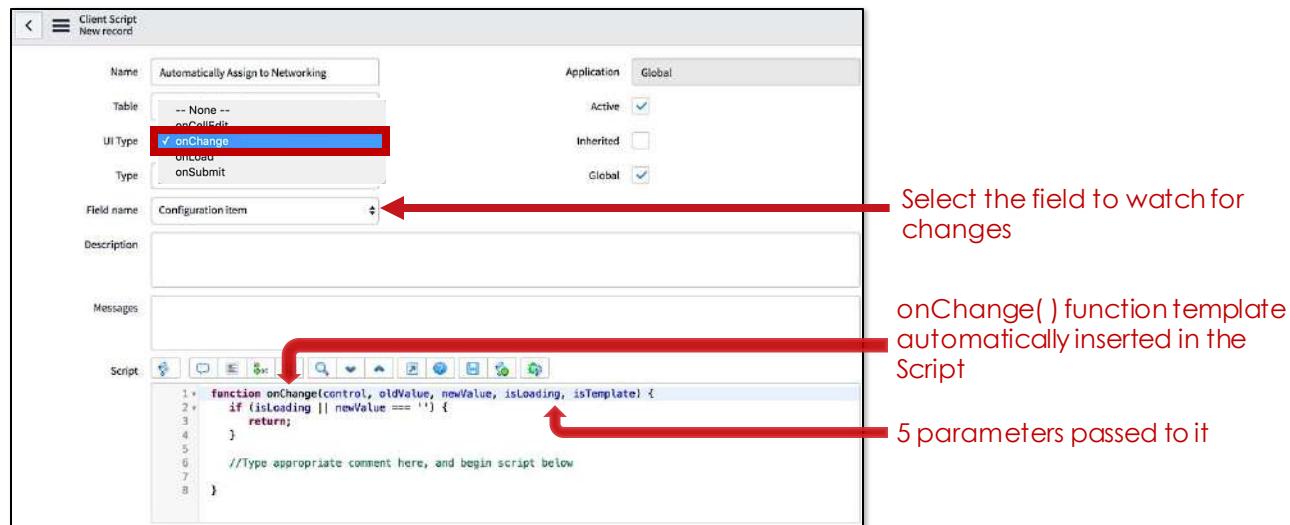
An `onChange()` Client Script watches one field. If two fields need to be watched, a second Client Script must be configured.

For example, one `onChange()` Client Script populates the 'Assignment group' field if the value in the `Configuration item [cmdb_ci]` field changes, while a second `onChange()` Client Script populates the Watch List if the value of the 'Priority' field changes to 1.

Client Script Overview: Client Script Type – onChange()

now.

- Select **onChange** as Client Script Type



An *onChange()* Client Script runs when a particular field value changes on the form.

The *onChange()* Client Script must specify these parameters:

- control** – name of the object (*field_name*) whose value just changed. The object is identified in the *Field name* field on the Client Script form.
- oldValue** – value of the control field when the form loaded and prior to the change. For example, if the value of *Assigned to* changes from Matt to Miranda the value of the parameter *oldValue* is Matt. *oldValue* will always be Matt (the value when the form loaded) no matter how many times the *Assigned to* value changes after the original form load.
- newValue** – value of the control field after the change. For example, if the value of *Assigned to* changes from Matt to Miranda, the value of the parameter *newValue* is Miranda.
- isLoading** – boolean value indicating whether the change is occurring as part of a form load. Value is true if change is due to a form load. A form load means all of a form's fields changed.
- isTemplate** – boolean value indicating whether the change occurred due to population of the field by a template. Value is true if change is due to population from a template.

Client Script Overview: Client Script Type – onChange() Template 'if' Statement

now.

- ALL field values change when a form loads
- onChange() template's **if** statement **aborts script execution if**
 - Field values change due to a form load
 - The newValue has no value

```
1 v function onChange(control, oldValue, newValue, isLoading, isTemplate) {  
2 v     if (isLoading || newValue === '') {  
3 v         return;  
4 v     }  
5 v  
6 v     //Type appropriate comment here, and begin script below  
7 v  
8 v }
```

Modify the script to change the behavior if necessary. For example, you might also check to see if the field value change was due to a template load.

```
function onChange(control, oldValue, newValue, isLoading, isTemplate) {  
    if (isLoading || newValue === '' || isTemplate) {  
        return;  
    }  
  
    //Type appropriate comment here, and begin script below  
}
```

Client Script Overview: Client Script Type – onCellEdit()

now.

- Script runs when a particular field value on a list changes
- Applies to all records selected

The screenshot shows a list of incidents in a ServiceNow interface. The first incident, INC0000055, has its 'State' field open, displaying a context menu with options: New, In Progress, On Hold, Resolved, Closed, and Canceled. The 'Resolved' option is highlighted with a blue background. To the right of the menu are two small icons: a green checkmark and a red X.

| Number | Opened | Short description | Caller | Priority | State | Category | Assignment group | Assigned to |
|------------|---------------------|---|-------------------|----------------|---|----------------|------------------|-------------|
| INC0000055 | 2016-10-02 21:47:23 | SAP Sales app is not accessible | Carol Coughlin | ● 1 - Critical | In Progress | Service Desk | Beth Anglin | |
| INC0000054 | 2015-11-02 12:49:08 | SAP Materials Management is slow or there is an outage | Christen Mitchell | ● 1 - Critical | On Hold | Software | Service Desk | |
| INC0000053 | 2016-10-02 13:48:46 | The SAP HR application is not accessible | Margaret Grey | ● 1 - Critical | In Progress | Inquiry / Help | Software | Beth Anglin |
| INC0000052 | 2016-10-02 13:48:40 | SAP Financial Accounting application appears to be down | Bud Richman | ● 1 - Critical | New ✓ In Progress On Hold Resolved Closed Canceled | Software | Fred Luddy | |
| INC0000051 | 2016-10-02 13:48:32 | Manager can't access SAP Controlling application | Joe Employee | ● 1 - Critical | In Progress | Software | Don Goodliffe | |

If you create a client-side script for fields on a form, an *onCellEdit()* Client Script can be used to ensure data in those fields is similarly controlled in a list.



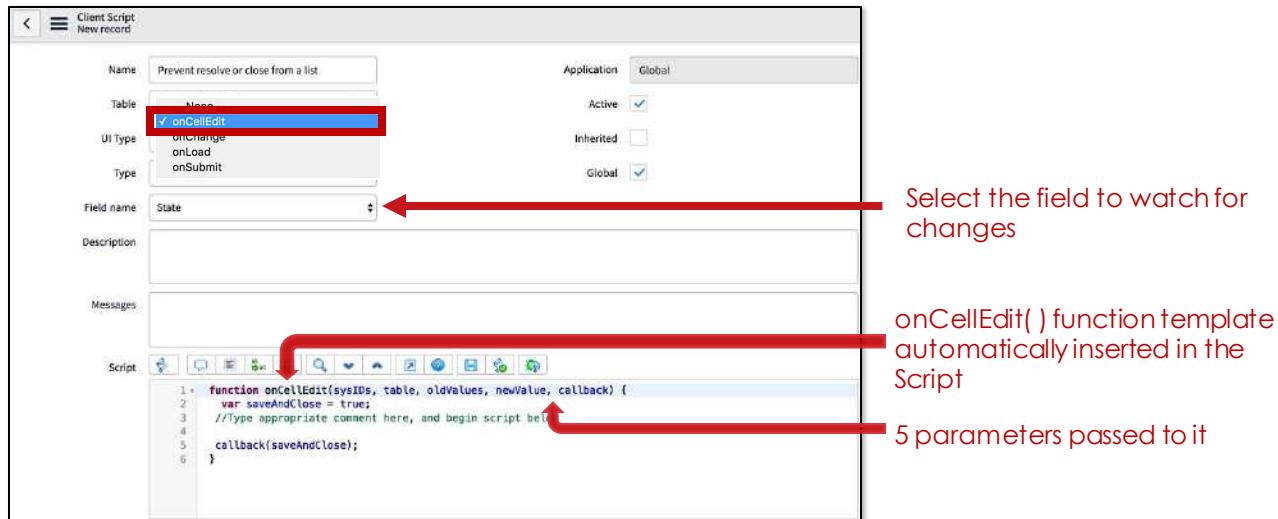
IMPORTANT

onCellEdit() scripts do not apply to List Widgets on homepages or dashboards.

Client Script Overview: Client Script Type – onCellEdit()

now.

- Select **onCellEdit** as Client Script Type



Parameters automatically passed to an *onCellEdit()* Client Script:

- sysIDs** – sys_id of the edited item(s).
- table** – the table name of the edited item(s).
- oldValues** – the old value of the edited cell(s).
- newValue** – the new value of the edited cell(s). Is the same for all edited items.
- callback** – a callback will continue the execution of other related cell edit scripts.

You must pass back either **true** or **false** in the callback function. If **true** is passed as a parameter, the other scripts are executed or the change is committed if there are no more scripts. If **false** is passed as a parameter, any further scripts are not executed and the change is not committed.

Example:

```
function onCellEdit(sysIDs, table, oldValues, newValue, callback) {
    var saveAndClose = true;
    if(newValue == 6) { //Resolved
        alert("You cannot change the state to 'Resolved' from a list");
        saveAndClose = false;
    }
    if (newValue == 7) { //Closed
        alert("You cannot change the state to 'Closed' from a list.");
        saveAndClose = false;
    }
    callback(saveAndClose);
}
```

Module Labs

now.

- **Lab 2.1**

- **Time:** 15-20m
- Two Simple Client Scripts
 - Practice writing Client Scripts
 - Understand the difference between an *onLoad()* and *onCellEdit()* Client Script



Two Simple Client Scripts

Lab
02.01

⌚15-20m

Lab Summary

You will achieve the following:

- Write and test two simple Client Scripts. Consider saving the Client Scripts module as a Favorite before beginning this lab.
- Create an *onLoad()* Client Script that generates an alert.
- Create an *onCellEdit()* Client Script to stop Incident records from being set to Resolved or Closed from a list view.

A. Create an *onLoad()* Client Script

1. Create a new Client Script.

Name: **Lab 2.1 onLoad Alert**
Table: **Incident [incident]**
UI Type: **Desktop**
Type: **onLoad**
Active: **Selected (checked)**
Inherited: **Not selected (not checked)**
Global: **Selected (checked)**
Description: **Lab 2.1 onLoad Client Script**

| | | | | |
|-------------|----------------------|-------------|-------------------------------------|--|
| Name | Lab 2.1 onLoad Alert | Application | Global | |
| Table | Incident [incident] | Active | <input checked="" type="checkbox"/> | |
| UI Type | Desktop | Inherited | <input type="checkbox"/> | |
| Type | onLoad | Global | <input checked="" type="checkbox"/> | |
| Description | Lab 2.1 onLoad Alert | | | |

2. Note the `onLoad()` function template populates the script field.

Replace the existing comment and add the following alert to the function.

```
function onLoad() {  
    //Type appropriate comment here, and begin script below  
    alert("The form has finished loading and is ready for user input");  
}
```

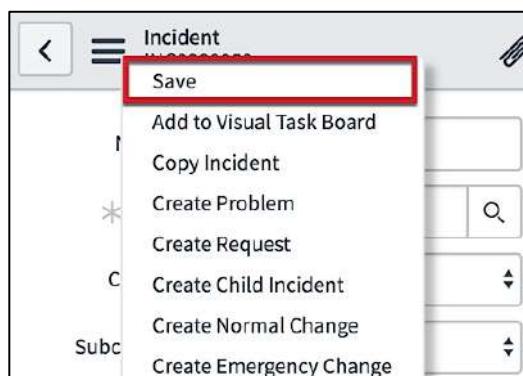
3. Select **Format Code** to re-align the script. (This is a good habit to get into.)
4. Select **Submit**.

TIP FROM THE FIELD:
The keyboard shortcut to **Format Code** starts with **Ctrl+A** for Windows and **Cmd+A** for Mac to select all code. Then **Shift+Tab** to format the code.

B. Test Your Work

1. Open any Incident record. What happens when the form loads?

2. Is this the behavior you expected? If not, debug and re-test.
3. Select **OK**.
4. Make a change to any field value.
5. Select **Save** on the form's Context menu to save the record and remain on the form.



6. Does the **Lab 2.1 onLoad Alert** Client Script execute again? Why or why not? Explain your reasoning:
-

7. Create a new Incident. What happens when the form loads for the new incident? Is this the behavior you expected?
-

8. Make the Client Script **inactive**.

- a) Open the **Lab 2.1 onLoad Alert** Client Script.
- b) Uncheck the **Active** field.
- c) Select **Update**.

C. Create an **onCellEdit()** Client Script

1. Create a new Client Script.

Name: **Lab 2.1 onCellEdit Alert**
Table: **Incident [incident]**
UI Type: **Desktop**
Type: **onCellEdit**
Field Name: **State**
Active: **Selected (checked)**
Inherited: **Not selected (not checked)**
Global: **Selected (checked)**
Description: **Lab 2.1 onCellEdit Alert**

2. Examine the pseudo-code for the script you will write:

- Create the saveAndClose variable with the value of true
- If the newValue of State is Resolved, alert the user they cannot change State to Resolved from a list and set the value of the saveAndClose variable to false
- If the newValue of State is Closed, alert the user they cannot change State to Closed from a list and set the value of the saveAndClose variable to false
- Execute the callback function returning the value of saveAndClose
 - If true is returned, the new value of the State field is committed
 - If false is returned, the new value of the State field is not committed

3. Write the following script:

```
function onCellEdit(sysIDs, table, oldValues, newValue, callback) {  
    var saveAndClose = true;  
  
    if(newValue == 6) { //Resolved  
        alert("You cannot change the State to 'Resolved' from a list");  
        saveAndClose = false;  
    }  
  
    if (newValue == 7) { //Closed  
        alert("You cannot change the State to 'Closed' from a list");  
        saveAndClose = false;  
    }  
  
    callback(saveAndClose);  
}
```

4. Select **Submit**.

D. Test Your Work

1. Navigate to **Incident > Open**.
2. Without opening a record, pick any Incident in the list. Record the current value of its **State** field:

3. Update the value of *State* to **Resolved** from the list view.
 - a) Double-click the **State** field.
 - b) Select **Resolved** from the drop-down list.
 - c) Select the **green checkmark** to save the change.



4. Did an alert appear advising you cannot change the State to 'Resolved' from a list? If not, debug and re-test your script.
 5. Select **OK**.
 6. Did the **Resolved** value remain in the *State* field? Why or why not?
-

7. Hold the **Shift** key down on your keyboard, then select **multiple State** fields. (*The background of the selected items changes to green.*)



8. Double-click any one of the selected **State** fields, change its value to **Closed**, and select **Save**.
 9. Did an alert appear advising you cannot change the State to 'Closed' from a list? If not, debug and re-test your script.
 10. Select **OK**.
 11. Did the **Closed** value remain in any of the State fields? Why or why not?
-
12. Select **OK**.
 13. Make the *Lab 2.1 onCellEdit Alert Client Script inactive*.

Lab Completion

Good job! You have successfully created *onLoad()* and *onCellEdit()* Client Scripts. Your testing efforts helped you to understand how each method works.

Client Scripts: Topics

now.

Client Script Overview

Client-side APIs

Client-side Debugging

Reference Objects

Script Versions

Client-side APIs: What Data Can You See in a Client Script?

now.

- Use ServiceNow's predefined client-side classes and methods to
 - Control how the platform looks and behaves in a web browser
 - Enhance the end user experience
- This class reviews the most popular Client-side APIs
 - **g_form** (GlideForm)
 - **g_user** (GlideUser)
 - **g_scratchpad** (in conjunction with a Display Business Rule, Module 5)



ServiceNow provides client-side JavaScript APIs:

g_form – object whose properties are methods used to manage form and its fields in the record.

g_user – object whose properties contain session information about the currently logged in user and their role(s).

g_scratchpad – object passed to a Client Script from a server-side script known as a Display Business Rule. The object's properties and values are determined by the server-side script.

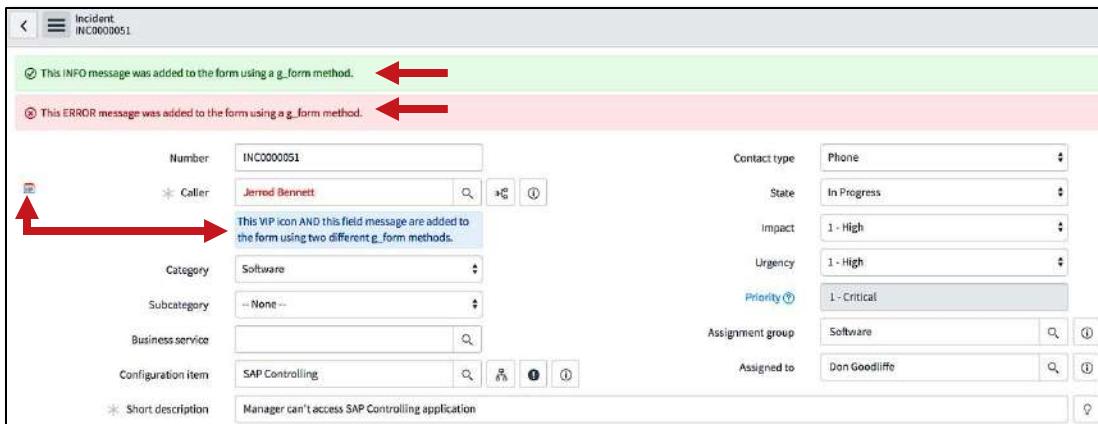
In addition to these pre-defined global variables, you also have any local variables you declare in a script.

Visit developer.servicenow.com for the complete list of available client-side classes and methods.

Client-side APIs: g_form Object

now.

- The **GlideForm** API provides useful methods to
 - Customize forms
 - Manage form fields and their data



Object **properties** are the fields from the currently loaded record.

Object **values** are the field values when the record is initially loaded.

Client-side APIs: g_form Methods

now.

- Access GlideForm methods using the **g_form** global object

`g_form.<method_name>(parameter information);` ← Syntax

- Examples

```
g_form.setValue('impact',1);
g_form.showFieldMsg('state','Change is waiting approval','info');
```

- Commonly used **g_form** method examples

- Draw attention to an area on the form: **flash()**, **showFieldMsg()**
- Get field information: **getValue()**, **getReference()**
- Change a field value: **setValue()**, **clearValue()**
- Change a choice list: **addOption()**, **clearOptions()**
- Get form information: **getSections()**, **isNewRecord()**
- Form actions: **addInfoMessage()**, **clearMessages()**

GlideForm methods are only used client-side.

Examples:

g_form.addInfoMessage() – displays an informational message at the top of a form.

g_form.addOption() – adds an option to the end of a Choice list.

g_form.clearMessages() – removes messages previously added to the form.

g_form.clearOptions() – removes all options from a Choice list.

g_form.clearValue() – clears a field's value.

g_form.flash() – flashes a field's label to draw attention to it.

g_form.getReference() – retrieves a reference object from the database.

g_form.getSections() – returns the elements of a form's section as an array.

g_form.getValue() – retrieves a field's value.

g_form.isNewRecord() – returns *true* if a record has never been saved.

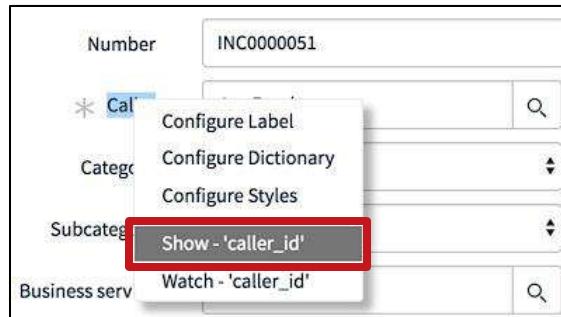
g_form.setValue() – sets a field's value.

g_form.showFieldMsg() – displays a message under a form field.

Client-side APIs: Referencing Field Names

now.

- g_form object methods **refer to fields by their field name**, not their label
- Easy way to locate a field name
 - Right-click a field's **label**
 - The field name appears on the Context menu



For example, the first parameter required by the addDecoration() method is a field name.

```
g_form.addDecoration('String field_name', 'String icon', 'String title');
```

Provide the field name **caller_id** vs. the field label **Caller**.

```
g_form.addDecoration('caller_id', 'icon-star', 'preferred member');
```



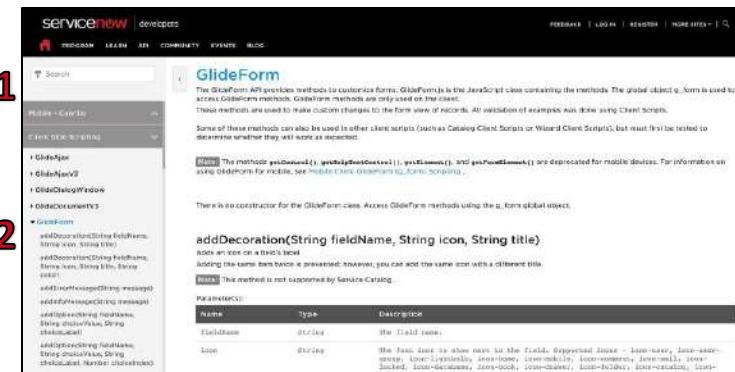
TIP FROM THE FIELD

Show – ‘field_name’ option on the Context menu works great to display the name of a field. A popup with a few of the field’s properties are displayed. Copy the field name for your script. Additional properties include Table, Type, Max Length, and Attributes.

Client-side APIs: g_form Method Syntax

now.

- Method use documented on the **Developers** portal
 - Two ways to search for methods:
 1. Use the Search field to search for a specific string, or
 2. Manually locate the method of interest on the page



To locate a `g` form method manually:

1. Select **API > Client**.
 2. Expand the **GlideForm** class.
 3. Locate and select the method of interest.
 - The method's syntax is documented in the middle-column of the page. Most methods have their parameters and return value documented.
 - If an example is included, it is located to the right of the method's documentation.

Client-side APIs: `g_form.getValue()`

now.

- `g_form.getValue()` is a very commonly used `g_form` method
- **Retrieves a field value from the form (*not the database*)**
- Pay close attention to the field's data type when using this method

```
var <varName> = g_form.getValue('<field_name>'); ← Syntax  
var incPriority = g_form.getValue('priority'); ← Example
```

| | |
|------------|--------------|
| Impact | 1 - High |
| Urgency | 1 - High |
| Priority ? | 1 - Critical |

Since JavaScript is weakly typed, it is not always necessary to verify data type when scripting. In the case of the `g_form.getValue()` method however, you must pay attention to data type or your script may have unexpected results.

The `g_form.getValue()` method always returns a string despite the data type of the field. If returning a number is important, use the `g_form.getIntValue()` or `g_form.getDecimalValue()` methods instead.



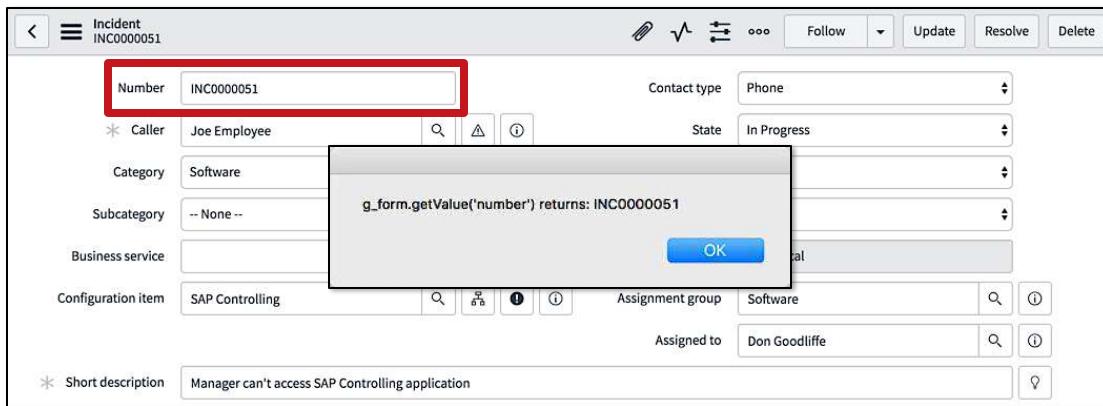
IMPORTANT

This method cannot retrieve values from the database, it retrieves values from forms even if the field is not visible in the current view or to the logged in user.

Client-side APIs: g_form.getValue() – Text Field

now.

- `g_form.getValue('number')`
 - Returns the field's content exactly as it appears on the form
 - The **Number [number]** field is a **text** field



Right-click a field's label and select **Show – '<field_name>'** from the Context menu to confirm a field's data type.

Client-side APIs: g_form.getValue() – Choice List

now.

- `g_form.getValue('impact')`
 - Returns the value of the selection, not the user friendly label
 - The **Impact [impact]** field is a **choice** list

The screenshot shows a ServiceNow incident form for record INC0000051. The 'Impact' field is highlighted with a red box, and its dropdown menu is open, displaying three options: '1 - High' (selected), '2 - Medium', and '3 - Low'. A tooltip window is also visible, stating 'g_form.getValue('impact') returns: 1'. Other fields on the form include 'Number' (INC0000051), 'Caller' (with a tooltip 'g_form.getValue('caller') returns: 1'), 'Category', 'Subcategory', 'Business service', 'Configuration item' (SAP Controlling), 'Assignment group' (Software), 'Assigned to' (Don Goodlife), and 'Priority' (1 - Critical). The 'State' is set to 'In Progress'.

To see a choice list's values, right-click the field's label and select **Show Choice List**.

Client-side APIs: g_form.getValue() – Reference Field

now.

- `g_form.getValue('caller_id')`
 - Returns the sys_id of the referenced record
 - The **Caller [caller_id]** field is a **reference** field

The screenshot shows an 'Incident' record with the sys_id INC0000051. The 'Caller' field is highlighted with a red box and contains the value 'Joe Employee'. A tooltip window is displayed over the 'Impact' field, showing the JavaScript code `g_form.getValue('caller_id')` and its return value: 681ccaf9c0a8016400b98a06818d57c7. The tooltip also includes an 'OK' button.

A sys_id is the database's unique key for a record.



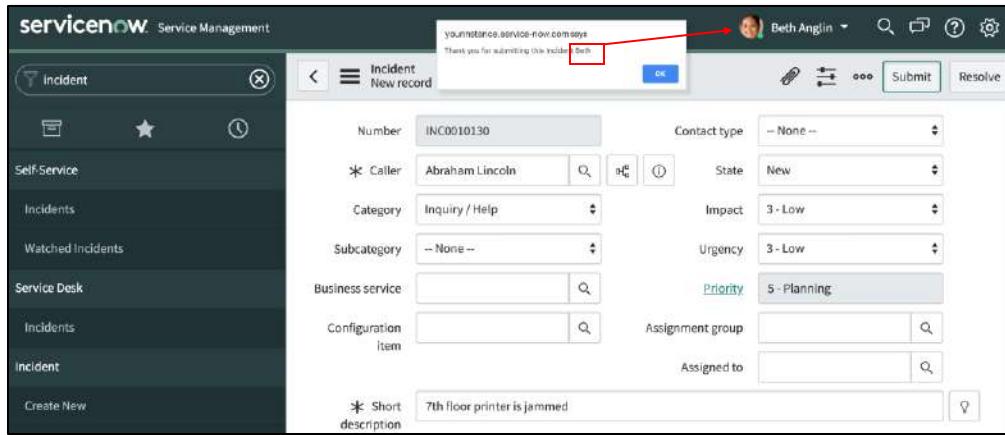
TIP FROM THE FIELD

The examples shown have used the `alert()` method in a Client Script to show the value of a single field. You can see several field values at once without a Client Script. If you have the 'admin' role, select **Show XML** on the form's Context menu. This will display the record's data in XML format.

Client-side APIs: g_user Object

now.

- The **GlideUser** API provides useful methods to access information about the currently logged in user
- Access GlideUser methods using the **g_user** global object



For the form shown, the `g_user` object contains the following properties and values for Beth Anglin:

```
g_user = {  
    userName: "beth.anglin",  
    userID: "46d44a23a9fe19810012d100cca80666",  
    firstName: "Beth",  
    lastName: "Anglin"  
}
```

Script used for the alert shown:

```
function onSubmit() {  
    if(g_form.isNewRecord()){  
        alert('Thank you for submitting this Incident ' +  
            g_user.firstName);  
    }  
}
```

Client-side APIs: g_user Object Properties and Methods

now.

- **g_user properties**

- firstName
- lastName
- userID
- userName

- **g_user property syntax**

```
g_user.<property>
```

Example:

```
alert("Logged in user: " +  
      g_user.userName);
```

- **g_user methods**

- getClientData()
- getFullName()
- hasRole()
- hasRoleExactly()
- hasRoleFromList()
- hasRoles()

- **g_user method syntax**

```
g_user.<method_name>(parameter  
information);
```

Example:

```
if (g_user.hasRole('itil')){  
    alert("Logged in user is a  
          Fulfiller");  
}
```

getClientData() – returns the session client value previously set with the *putClientData()* method.

getFullName() – returns the logged in user's first name and last name separated by a space.

hasRole() – returns true if the logged in user has the specified role or has the admin role.

hasRoleExactly() – returns true only if the logged in user has the specified role.

hasRoleFromList() – returns true if the logged in user has at least one role from the passed in list or has the admin role.

hasRoles() – returns true if the logged in user has any role.



IMPORTANT

DO NOT rely on *g_user* methods to apply security. Client-side security is easily defeated using developer tools built into browsers. Access Control or another server-side security strategy is recommended.

Module Labs

now.

- **Lab 2.2**

- **Time:** 20-25m
- **g_form and g_user**
 - Practice writing a Client Script using the `g_form` and `g_user` objects and their methods



g_form and g_user

**Lab
02.02**

⌚20-25m

Lab Summary

You will achieve the following:

- Practice writing a Client Script using the g_form and g_user objects and their methods. When this lab is completed you will be able to mark a Cloud Dimensions' Phase II requirement complete.



Business Problem

At Cloud Dimensions, a Major Incident Manager is responsible for submitting Priority-1 (P1) Incidents. This ensures the Incident adheres to strict process and communication guidelines.

Twice in the last 6 months, the on-duty Major Incident Manager was not available to perform this duty and the initial information provided in the record by the IT analyst was not complete. This created additional havoc when management received incomplete P1 Email Notifications.

Project Requirement

A P1 Incident record is expected to have the following fields populated before it is submitted:

- Category
- Configuration Item
- Assignment group
- Short description

The fields should not be mandatory in the event the information required to populate them is not known at the time of the initial submission. If this is the case, details as to why the information is missing should be added to the 'Additional comments' field.

Cloud Dimensions is requesting this functionality be built into the Incident form:

- If the person creating the record is NOT a Major Incident Manager, then the agent creating a P1 Incident should be prompted to confirm the minimum Major Incident information is included in the record before it is submitted.
- Instructions on how to handle unavailable mandatory details should be included as well.

A. Preparation

1. Create a new role for the Major Incident Manager.

a) Navigate to **User Administration > Roles**.

b) Select **New** to create a new Role.

Name: **major_inc_mgr**

Description: **Role required for Major Incident Managers**

c) Select **Submit**.

2. Create a new **Major Incident Manager** Group to identify the organization's Major Incident managers and give the Group the **major_inc_mgr** Role.

a) Open **User Administration > Groups**.

b) Select **New** to create a new Group.

Name: **Major Incident Managers**

Description: **Major Incident Managers**

c) **Save** the record, remain on the form.

d) Select the **Edit** button on the *Roles* Related List and add the **major_inc_mgr** role to the list.

e) Select the **Edit** button on the *Group Members* Related List and add **Beth Anglin**, **Christen Mitchell**, and **Don Goodliffe** to the list.

B. Use both g_form and g_user methods in a Client Script

1. Create a new Client Script.

Name: **Lab 2.2 Confirm Major Incident Details**

Table: **Incident [incident]**

UI Type: **Desktop**

Type: **onSubmit**

Active: **Selected (checked)**

Inherited: **Not selected (not checked)**

Global: **Selected (checked)**

Description: **Confirm initial P1 details are included if the Incident creator is not a Major Incident Manager**

2. Examine the pseudo-code for the script you will write:

When the form is submitted, saved or updated

- If Impact and Urgency are both high and the user does not have the major_inc_mgr role
- Create the ans variable to store the user's response to a confirmation box asking them to ensure base information is included in the record before submitting a Priority-1 incident
- If the user cancels the submission
 - Generate an alert stating the incident was not submitted and provide instructions to use the Additional comments field if Major Incident base information is missing
 - Add a field message below the Category, Configuration item, Assignment group and Short description fields identifying them as Major Incident base fields
- Return true or false based on the confirmation box response

3. Write the following script:

```
function onSubmit() {
    if(g_form.getValue('impact') == 1 && g_form.getValue('urgency') == 1 &&
       !g_user.hasRoleExactly('major_inc_mgr')){

        var ans = confirm("The customer is notified of all Priority-1 Incidents.  
Confirm base information is included before submitting this P1 incident.\n\nSelect  
Ok to submit, or Cancel to return to the record.");

        if(!ans) {
            g_form.addInfoMessage("Incident not submitted");
            g_form.addInfoMessage("If base information is unavailable, use the  
'Additional comments' field to document why it is missing.");
            g_form.showFieldMsg('category', "Major Incident base field");
            g_form.showFieldMsg('cmdb_ci', "Major Incident base field");
            g_form.showFieldMsg('assignment_group', "Major Incident base field");
            g_form.showFieldMsg('short_description', "Major Incident base field");
        }

        return ans;
    }
}
```

4. Select **Submit**.

C. Test Your Work

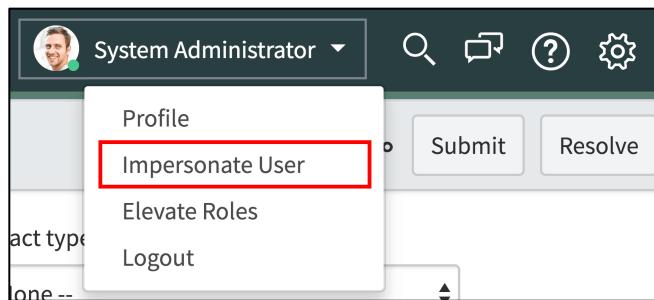
1. Create a new Incident, populate the mandatory fields, and set both the *Impact* and *Urgency* values to **1-High**.
 2. **Save** the record. Remain on the form.
 3. When the confirmation window opens, **cancel** the submission.
 4. What happened to the Incident form when you cancelled the submission? Was the Incident submitted?
-
-

5. Modify the confirmation dialog box in the script so the confirmation message includes the logged in user's first name. Which property do you need to use?

6. Test the modified script. Was the logged in user's first name included? If not, debug and re-test.

7. Impersonate a user and create a major Incident.

- a) Select the **System Administrator** avatar on the Header bar's top-right corner to display the Logged-in User's Context menu.
- b) Select **Impersonate User**.



TIP FROM THE FIELD:

As an administrator, you will use impersonate quite often. The shortcut to open impersonate is **control + option + i** on Mac and **Ctrl + Alt + i** on Windows.

- c) Enter **Beth Anglin**, **Christen Mitchell**, or **Don Goodliffe** in the *Search for user* field and select the record when it appears in the drop-down list. The page will refresh and their avatar will appear on the Header bar.



8. Create a new Incident, populate the mandatory fields, and set both the *Impact* and *Urgency* values to **1-High**.
9. **Save** the record, remain on the form. Did the confirmation message display? Explain why not:

10. Repeat steps 7a-7c to end the impersonation. *You should be logged in as the System Administrator when this step is complete.*

11. Make the *Lab 2.2 Confirm Major Inc Details Client Script* **inactive**.

Lab Completion

Good job! You have successfully used *g_form* and *g_user* methods and properties in a script, as well as practiced impersonating users to confirm the successful execution of a role based on specific role conditions.

Client Scripts: Topics

now.

[Client Script Overview](#)

[Client-side APIs](#)

[Client-side Debugging](#)

[Mobile Client Scripts](#)

[Script Versions](#)

Client-side Debugging: Debugging Client Scripts – Desktop

now.

- Many times your scripts do not work as expected
- You can debug them using the following strategies:
 - ServiceNow built-in client-side debugging tools
 - Script debug messages
 - JavaScript Log and jslog()
 - Response Time Indicator
 - JavaScript debugging tools
 - alert()
 - try/catch
 - Browser tools (*e.g. JavaScript console, Web Console*)
 - 3rd party tools (*e.g. Firebug*)



In some cases, more than one debugging strategy may need to be used.

The built-in and JavaScript debugging tools will be reviewed.

Client-side Debugging: Script Debug Messages

now.

- Include **addInfoMessage()** and **addErrorMessage()** methods in your script

Pros:

- Convenient strategy as messages appear on the form being tested
- Write any debugging information you want as messages are not restricted to field values

Cons:

- Statements must be removed before moving scripts to other instances
- Other users see the message(s)

The screenshot shows a ServiceNow incident form titled 'INC0000053'. At the top, there are three green status messages: 'The value of Caller is: 46d4a69ba9fe1981001af9616bf01185', 'Line 15 executed successfully!', and 'A runtime error occurred: TypeError: g_form.getVal is not a function'. Below these messages is a standard incident record with fields like Number, Contact type, State, Category, and Impact.

The *showFieldMsg()* method can also be used to display debugging output next to a specific field.

All users are affected when the *addInfoMessage()*, *addErrorMessage()*, and *showFieldMsg()* methods are used as everyone sees the output on the form. In a multiple administrator/developer environment this approach has the potential to be confusing. Consider including code in your debugging scripts to include your name in the output or to hide the output from other users.

- Include a *g_user* property in the script to identify yourself in the message. Example:
`g_form.addInfoMessage(g_user.firstName + ", the value of caller_id is: " + g_form.getValue('caller_id'));`
- Include a *g_user* property in combination with an IF statement to display the message only to you. Example:
`if(g_user.userName == 'admin') {
 g_form.addInfoMessage("The value of caller_id is: " +
 g_form.getValue('caller_id'));
}`



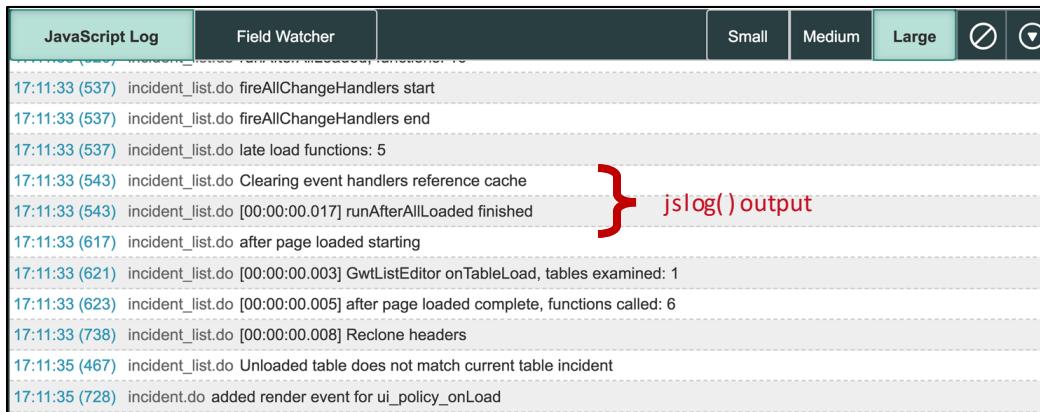
TIP FROM THE FIELD

Create a **Debug Syntax Editor Macro** to quickly insert commonly used debugging code.

Client-side Debugging: JavaScript Log and jslog()

now.

- Log file
- Runs on the browser
- Use the **jslog()** method to send information to the JavaScript Log



Follow these steps to open the JavaScript Log (*admin users only*):

1. Select the **System Settings** (gear) icon on the Header bar.
2. Turn the **JavaScript Log and Field Watcher** button on. The JavaScript Log opens at the bottom of the page (*Small, Medium, and Large buttons allow you to select your preferred window size*).
3. Create the conditions necessary to test your script.
4. View *jslog()* messages in the JavaScript Log.

The JavaScript Log appears only for the admin user who opened it; no other users are impacted.

If the JavaScript Log closes, use the **JavaScript Log and Field Watcher** button on the System Settings menu to re-open it. It must remain open in order to debug.

The JavaScript Log is only available in Desktop mode; it is not available in the Tablet or Mobile operating systems.

Any argument passed to the *jslog()* method appears in the JavaScript Debug window. Such as:

- Strings
- g_form or g_user properties or methods
- Variables
- JavaScript string escape characters such as "\n" (*new line*) and "\t" (*tab*).



TIP FROM THE FIELD

Prepend your *jslog()* messages with an easily identifiable string such as the script name or your initials so you can quickly search through the debug messages. In this example, the string "LOG MSG!!" is used.

```
jslog("LOG MSG!! The value of Priority is: " +  
g_form.getValue('priority'));
```

Client-side Debugging: Response Time Indicator

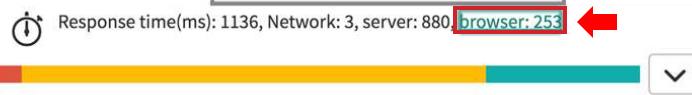
now.

- Useful for locating the cause of slow page loads

- Times are in milliseconds

- Network
- Server
- Browser

CSS and JS Parse: 108
Form Sections: 170
UI Policy - On Load: 7
Client Scripts - On Load: 4
Client Scripts - On Change (initial load): 2
Browser processing before onload: 25
DOMContentLoaded to LoadEventEnd: 29
addLoadEvent functions: 120



- Select the **browser** link to see a detailed breakdown of the browser processing times on forms

Use this strategy to look for Client Script issues causing long load times.

Administrators can disable the Response Time Indicator by setting the `glide.ui.response_time` property to **false**.

Client-side Debugging: JavaScript Debugging Tool – Try/Catch

now.

- JavaScript debugging strategy used to trap runtime errors
- This example passes the Syntax Check but has a runtime error

```
try {
    g_form.getVal('priority');
}

catch(err) {
    g_form.addErrorMessage("A runtime error occurred: " + err);
}
```

The screenshot shows a ServiceNow incident detail page for incident INCO000053. The notes section contains the following text:

- ② The value of Caller is: 46d4a09ba9fe1981001af016bf01185
- ③ Line 15 executed successfully!
- ④ A runtime error occurred: TypeError: g_form.getVal is not a function

A red arrow points from the bottom note to the code line in the previous code block. Another red arrow points from the bottom note to the error message in the notes section.

Try/Catch syntax:

```
try{
    //code to execute goes here
}
catch(err){
    //code to deal with error here
}
```

err is a JavaScript object with properties `description`, `message`, `name`, and `number`.

You can throw your own error messages for the catch function using the JavaScript `throw()` function. Try/catch only traps runtime errors. Using `throw()` you can also catch user errors such as entering an invalid data value in a field.

Module Labs

now.

- **Lab 2.3**

- **Time:** 20-25m
- Debugging Client Scripts
 - Practice debugging a Client Script



Debugging Client Scripts

Lab
02.03

⌚20-25m

Lab Summary

You will achieve the following:

- Debug a Client Script using ServiceNow and JavaScript strategies.

A. Preparation

1. Locate and open the **Lab 2.3 Client Script Debugging** Client Script for editing.
2. Select the **Active** checkbox to make the script active.
3. Examine the pseudo-code for the script:
 - When the value of Impact changes
 - Store the current value of State in the variable incState
 - If the value of incState is 1
 - Add a decoration to the State field
 - Have the State label flash the color teal
 - If Impact has the value High or Medium
 - Remove the State choice list options On Hold, Resolved, Closed, and Canceled
 - Log a message to the top of the form confirming the State options removal
 - Else if the form is not loading and Impact does not have the value High or Medium
 - Clear the State choice list
 - Add options back to the State choice list
 - Set the State field to the value of incState
4. Consider saving the record as a favorite as you will be opening it a few times throughout the lab. Select **Create Favorite** on the form's Context menu.
5. Select **Update**.

B. Observe the Current Execution of the Script

1. Create a new Incident.
2. Without changing the value of the State field, open the drop-down choice list and make a mental note of the available options.
3. Update the value of *Impact* to **1-High**.
4. Confirm the **Lab 2.3 Client Script Debugging** Client Script executed as expected.

| Expected Behavior | Occurred? |
|--|-----------|
| A decoration appeared to the left of the State field | |
| The State 's label flashed teal for four seconds | |
| The State choice list options now only include 'New' and 'In Progress' | ✓ |
| An Information Message appears at the top of the form confirming the State options removal | ✓ |

C. Use *jslog()* to confirm Variable Values and Script Execution

1. Open the **Lab 2.3 Client Script Debugging** Client Script.
2. Use the *jslog()* method in strategic places to debug your script. If you require assistance, you can use this script as an example of how to include *jslog()* messages in your script.

```
function onChange(control, oldValue, newValue, isLoading, isTemplate) {
    if (isLoading || newValue === '') {
        return;
    }

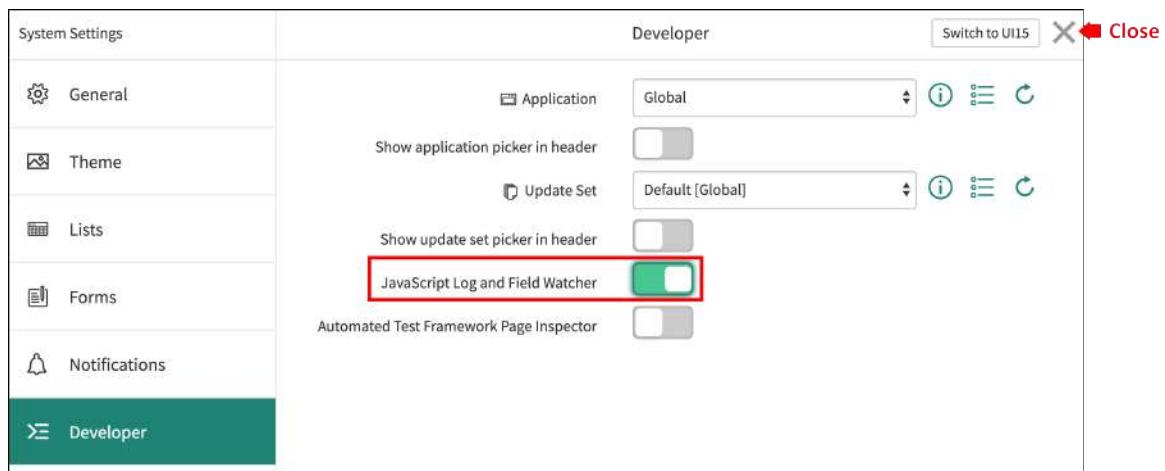
    //Document the current value of State, if New, use a decoration
    var incState = g_form.getValue('incident');
    jslog("<your_initials> - the value of incState is: " + incState);

    if (incState == 1) {
        jslog("<your_initials> LINE 11 EXECUTED!");
        g_form.addDecoration('state','icon-edit','Gathering initial details');
        g_form.flash('state', "teal", -4);
    }
}
```

3. Select **Update**.

D. Turn on the JavaScript Log and Field Watcher

1. Select the **System Settings** (gear) icon on the right-side of the banner frame.
2. Select **Developer**.
3. Turn the **JavaScript Log and Field Watcher** switch on.
4. Close the System Settings window when you are ready to proceed.



E. Force the Updated Script to Execute

1. Create a new Incident.
2. Select the **Clear log** icon (Ø) to remove any existing log messages.
3. Select your preferred view (*Small, Medium, or Large*) so the log messages are visible at the bottom of your screen.
4. Update the value of *Impact* to **1-High**.
5. Review the log messages in the JavaScript Log.
 - What is the value currently in the **incState** variable? _____
 - What should it be? _____
 - Does the string "LINE 11 EXECUTED!" appear as a logged message? _____
6. What can you conclude regarding the debugging output currently in the JavaScript Log?

7. At this point, you know something is wrong with the value in incState variable. Before leaving the Incident form, double-check the **State** field name by right-clicking the field's label. What is the exact spelling of the **State** field name? _____

8. Open the **Lab 2.3 Client Script Debugging** Client Script.

9. Locate the statement at the beginning of your script that reads:

```
var incState = g_form.getValue('incident');
```

Review the **State** field name in this statement, does it match the field name you documented in step-7? _____

10. Update the statement to

```
var incState = g_form.getValue('state');
```

11. Select **Update**.

12. Force your updated script to execute by creating a new Incident. If you closed the JavaScript Log, re-open it.

13. Clear the JavaScript log.

14. Update the value of *Impact* to **1-High**.

15. Confirm the **Lab 2.3 Client Script Debugging** Client Script executed as expected.

| Expected Behavior | Occurred? |
|--|-----------|
| A decoration appeared to the left of the State field | ✓ |
| The State's label flashed teal for four seconds | ✓ |
| The State choice list options now only include 'New' and 'In Progress' | ✓ |
| An Information Message appears at the top of the form confirming the State options removal | ✓ |

16. Review the messages in the **JavaScript Log**.

- Is the value of *incState* **1**? _____
- Does the string "LINE 11 EXECUTED!" appear as a logged message? _____

17. If you answered **No** to any of these questions in steps 15 or 16, debug and re-test.

18. Close the **JavaScript Log and Field Watcher** by select the **Close** () icon.
 19. Review the **InfoMessage** at the top of the form. Are you the only person who can see this script output?
-

20. Form messages are a good debugging strategy as the results are instantly presented the top of the form you are testing on. Would this type debugging strategy be best in a development or production instance?
-

F. Explore Built-in Debugging Messages

1. Open another Incident where the value of *Impact* is **1-High** or **2-Medium**.
 2. Update the value of Impact to **3-Low**.
 3. Examine the built-in debugging provided by the platform below the Impact field. Correct any errors in the script and re-test until no errors remain.
- Hint:** You will find the list of available GlideForm methods on developer.servicenow.com very helpful for this step.

G. Explore Debugging While Impersonating

1. Impersonate **Beth Anglin**.
 2. Can Beth turn the **JavaScript Log and Field Watcher** on? Why or why not?
-
3. Beth Anglin does not have the 'admin' role and therefore cannot impersonate other users. Return to the System Administrator, then impersonate **Fred Luddy**.

4. Can Fred turn the **JavaScript Log and Field Watcher** on? Why or why not?
-

5. Based on this test, what can you conclude regarding the **JavaScript Log and Field Watcher** while you are impersonating another user?
-

6. End the impersonation. You should be logged in as the System Administrator when this step is complete.

7. Open the **Lab 2.3 Client Script Debugging** Client Script.

- Remove any debugging statements.
- Make the Client Script **inactive**.

8. Select **Update**.

9. If you saved the Client Script as a Favorite at the beginning of the lab, you can remove it from the list now as you have completed debugging it.

Lab Completion

Well done! You have successfully debugged a script using client-side debugging methods.

Client Scripts: Topics

now.

Client Script Overview

Client-side APIs

Client-side Debugging

Reference Objects

Script Versions

Reference Objects

now.

- Reference Object records exist on a table other than a form's currently loaded table
- Reference Object data is not loaded into forms
- Client-side scripts only have access to data on forms

"Accounts Payable printer issues occur so infrequently, I always have to look up the correct Assignment group. It would be great if the 'Assigned to' field would automatically populate when I select the CI."



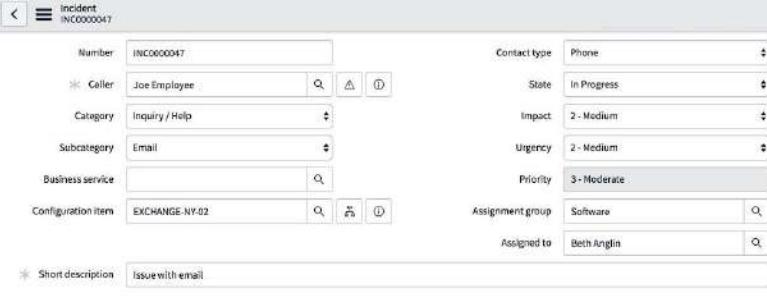
1st Level Support

When writing client-side scripts, you have access to all form fields and their values. Reference Object fields exist on forms but the Reference Object record itself is not loaded into the form.

Reference Objects: Scripting with Reference Objects

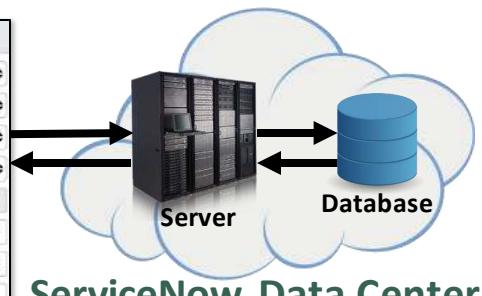
now.

- Forms only store a Reference Object's **sys_id**
- Reference Object fields cannot be directly referenced from a Client Script
- Retrieving Reference Object fields **requires a trip to the server and back**
 - Server trips take time
 - Make as few trips as possible!!



The screenshot shows a ServiceNow incident form with the following fields:

- Number: INC0000047
- * Caller: Joe Employee
- Category: Inquiry / Help
- Subcategory: Email
- Business service: EXCHANGE-NY-02
- Contact type: Phone
- State: In Progress
- Impact: 2 - Medium
- Urgency: 2 - Medium
- Priority: 3 - Moderate
- Assignment group: Software
- Assigned to: Beth Anglin
- Short description: Issue with email



A reference field stores a sys_id for the record it references in the database, but the sys_id is not shown. The reference field shows the record's *display value* on the form.

Reference Objects: `g_form.getReference()` Method

now.

- Returns a Reference Object's record client-side

```
1+ function onChange(control, oldValue, newValue, isLoading, isTemplate) {
2+   if (isLoading || newValue === '') {
3+     return;
4+   }
5+
6+   //Get the record identified in the Caller field from the database, call the callerInfo() function upon return
7+   var userRecord = g_form.getReference('caller_id', callerInfo);
8+
9+   //Pass the returned record into the callback function
10+  function callerInfo(userRecord) {
11+
12+    //Advise if the Caller is locked out or not
13+    alert("User ID: " + userRecord.user_name + "\nLocked Out: " + userRecord.locked_out);
14+
15+  }
16+}
```

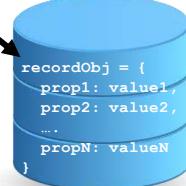
Step 3:

Execute logic after the object returns from the server

Step 1:
Request a record
from the database



Step 2:
Requested record returned
from the database



The following occurs when a script requests a record on another table in the database:

- A client-side script requests a record from the database.
- The server retrieves the requested record from the database and passes the record back to the calling client-side script as an object.
- The callback function logic executes when the object returns from the server.

The `g_form.getReference()` method runs asynchronously when a callback function is used. When the record is returned to the calling script, the logic in the callback function executes. If the callback function is omitted, the `g_form.getReference()` method runs synchronously.

Reference Objects: `g_form.getReference()` Method

now.

- **Syntax**

```
g_form.getReference(field_name,callback_function)
```

- **Example**

```
function onChange(control, oldValue, newValue, isLoading, isTemplate) {
    if (isLoading || newValue === '') {
        return;
    }

    //Get the record identified in the caller_id field from the DB, call the callerInfo() function upon return
    var userRecord = g_form.getReference('caller_id', callerInfo);

    //Pass the returned record into the callback function
    function callerInfo(userRecord) {
        alert("User ID: " + userRecord.user_name + "\nLocked Out: " + userRecord.locked_out);
    }
}
```

In the example shown, the `callerInfo()` function is called only after the `caller_id` record is returned from the server.

Reference Objects: Synchronous vs. Asynchronous

now.

- **Synchronous** (*bad practice and will not work on mobile*)

```
var userRecord = g_form.getReference('caller_id');
alert("User ID: " + userRecord.user_name + "\nLocked Out: " + userRecord.locked_out);
```

- **Also synchronous** (*bad practice and will not work on mobile*)

```
var callerEmail = g_form.getReference('caller_id').email;
alert("Caller email = " + callerEmail);
```

- **Asynchronous** (*Good, but not the best!*)

```
var userRecord = g_form.getReference('caller_id', callerInfo);

function callerInfo(userRecord){
    alert("User ID: " + userRecord.user_name + "\nLocked Out: "
        + userRecord.locked_out);
}
```

Synchronous `getReference()` method calls lock the form until the requested record is returned from the database. Asynchronous execution allows users to continue using the form while the method call executes.

The first example executes synchronously resulting in a poor user experience. `getReference()` without a callback function will not run on the Mobile platform.

The second example also executes synchronously and returns only one field's value. The performance penalty is the same as for the first example.

The third example demonstrates how to make an asynchronous call to the server. While this was considered to be a best practice for some time, using `g_form.getReference` retrieves all fields from a requested GlideRecord. Typically, the client may only need one field. This can have an impact on performance. This has been included so you are aware of its use. We will discuss a better way later.

Module Labs

now.

- **Lab 2.4**

- **Time:** 15-20m
- Client Scripting with Reference Objects
 - Practice writing, testing, and debugging a Client Script that works with Reference Objects



Client Scripting with Reference Objects

Lab
02.04
⌚15-20m

Lab Summary

You will achieve the following:

- Practice using the `getReference()` method to retrieve records for Reference Objects.
You will write a script to automatically set the Priority, Risk, and Impact to Low if the Configuration item is 3D Pinball.

A. Client Scripting with Reference Objects

1. Create a new Client Script.

Name: **Lab 2.5 Reference Objects**

Table: **Change Request**

UI Type: **Desktop**

Type: **onChange**

Field: **Configuration Item**

Active: **Selected (checked)**

Inherited: **Not selected (not checked)**

Global: **Selected (checked)**

2. Examine the pseudo-code for the script you will write:

- When the Configuration Item field on a Change Request is changed
 - Request the CI record from the server and execute the callback function.
 - In the callback function
 - If the CI is '3D Pinball'
 - Set the Priority to Low, Risk to None, and Impact to Low.
 - Make the Priority, Risk, and Impact fields read-only.

3. Write the script:

```
function onChange(control, oldValue, newValue, isLoading, isTemplate) {  
    if (isLoading || newValue === '') {  
        return;  
    }  
  
    var affectedCI = g_form.getReference('cmdb_ci',checkCI);  
  
    function checkCI(affectedCI) {  
        if(affectedCI.name == '3D Pinball') {  
            g_form.setValue('priority',4);  
            g_form.setValue('risk',5);  
            g_form.setValue('impact',3);  
  
            g_form.setReadOnly('priority',true);  
            g_form.setReadOnly('risk',true);  
            g_form.setReadOnly('impact',true);  
        }  
    }  
}
```

4. Select **Submit**.

B. Test Your Work

1. Create a new **Normal Change Request**.
2. Enter **3D Pinball** in the *Configuration Item* field.
3. Are the *Priority*, *Risk*, and *Impact* fields set correctly? Are they read-only? If not, debug and re-test.
4. Change the Configuration Item to anything *except* 3D Pinball.
5. Are the *Priority*, *Risk*, and *Impact* fields read-only? Should they be?

6. Modify the script so the *Priority*, *Risk*, and *Impact* fields are editable if the Configuration Item is not 3D Pinball. (*Script on the next page.*)

```

function onChange(control, oldValue, newValue, isLoading, isTemplate) {
    if (isLoading || newValue === '') {
        return;
    }

    var affectedCI = g_form.getReference('cmdb_ci',checkCI);

    function checkCI(affectedCI) {
        if(affectedCI.name == '3D Pinball') {
            g_form.setValue('priority',4);
            g_form.setValue('risk',5);
            g_form.setValue('impact',3);

            g_form.setReadOnly('priority',true);
            g_form.setReadOnly('risk',true);
            g_form.setReadOnly('impact',true);
        }
        else {
            g_form.setReadOnly('priority',false);
            g_form.setReadOnly('risk',false);
            g_form.setReadOnly('impact',false);
        }
    }
}

```

7. Select Update.
8. Test and if required, debug.

C. Make a Second getReference() Call in the Same Script

1. Modify the script to display the Requested by user's email address in the Description field by using a second getReference() method call and callback function. (Script is on the next page.)

```

function onChange(control, oldValue, newValue, isLoading, isTemplate) {
    if (isLoading || newValue === '') {
        return;
    }

    var desc = g_form.getValue('description');
    var reqBy = g_form.getReference('requested_by',checkUserInfo);
    var affectedCI = g_form.getReference('cmdb_ci',checkCI);

    function checkUserInfo(reqBy) {
        g_form.setValue('description',reqBy.email + "\n\n" + desc);
    }

    function checkCI(affectedCI) {
        if(affectedCI.name == '3D Pinball') {
            g_form.setValue('priority',4);
            g_form.setValue('risk',5);
            g_form.setValue('impact',3);

            g_form.setReadonly('priority',true);
            g_form.setReadonly('risk',true);
            g_form.setReadonly('impact',true);
        } else {
            g_form.setReadonly('priority',false);
            g_form.setReadonly('risk',false);
            g_form.setReadonly('impact',false);
        }
    }
}

```

2. **Save** the script.
3. Test and if required, debug.
4. Make the Lab 2.5 Reference Objects script **inactive**.

Lab Completion

Well done! You successfully created a Client Script that required two separate getReference() calls that executed two different actions when one field changed on the form.

Client Scripts: Topics

now.

Client Script Overview

Client-side APIs

Client-side Debugging

Reference Objects

Script Versions

Script Versions

now.

- Track changes to the record
- Created automatically every time a script is saved, submitted, or updated
- Can compare versions to see differences
- Can revert to a previous version of a script

Located at the bottom of the form

| Update Versions | | | |
|--------------------------|---|-----------------------|----------|
| | Name | Recorded at | State |
| <input type="checkbox"/> | sys_script_client_0be5ced4d21f403007f44690a6f7a7d34 | 08/24/17 13:01:30.351 | Current |
| <input type="checkbox"/> | sys_script_client_0be5ced4d21f403007f44690a6f7a7d34 | 08/24/17 13:01:26.379 | Previous |
| <input type="checkbox"/> | sys_script_client_0be5ced4d21f403007f44690a6f7a7d34 | 08/24/17 09:23:55.512 | Previous |
| <input type="checkbox"/> | sys_script_client_0be5ced4d21f403007f44690a6f7a7d34 | 08/24/17 08:32:59.687 | Previous |

Script versions are available for other script types and are not unique to Client Scripts.

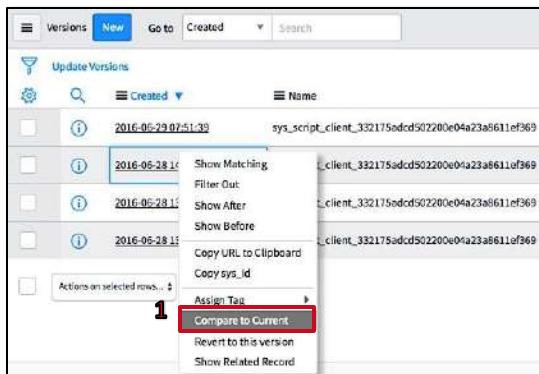
Update Sets do not include every version. If you migrate a script from one instance to another, only the most recent version of the script is included in an Update Set.

Script Versions: Two Ways to Compare Script Versions

now.

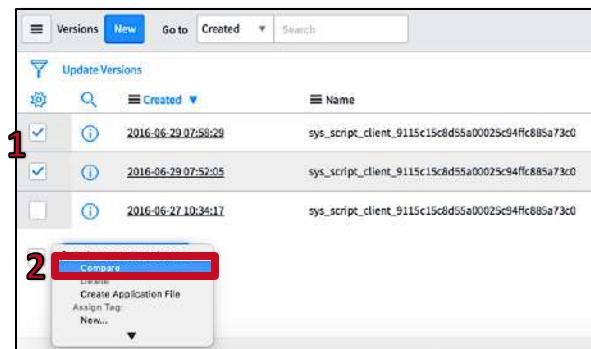
Compare to the current version

1. Right-click the version record and select **Compare to Current** on the Context menu



Compare any two versions

1. Select checkboxes beside two versions
2. Select **Compare** on the *Actions on selected rows...* menu



The **Versions** Related List is visible baseline. In the event it is not, open the record's Context menu and select **Configure > Form Layout**. Add **Versions** to the *Selected* column and save the change.

Script Versions: Compare to Current

now.

- Highlights the fields that differ
 - Update settings in the Current Version
 - Overwrite settings from the Selected Version to the Current Version
 - Revert to the Selected Version

The screenshot shows a 'Compare to Current' interface for a 'sys_script_client' record. It displays two columns: 'Selected Version' and 'Current Version'. The 'Selected Version' has a priority of 1 and a script containing an onLoad function. The 'Current Version' has a priority of 2 and a different script. Red annotations highlight specific fields:

- A red arrow points to the 'Priority' field in the 'Current Version' column with the text 'Select a different field'.
- A red arrow points to the 'Name' field in the 'Selected Version' column with the text 'Overwrite the current value of the field with value from the selected version'.
- A red arrow points to the 'Script' field in the 'Current Version' column with the text 'Select to edit script'.

| Compare to Current - [sys_script_client] - Testing Versions | |
|---|--|
| | |
| Selected Version | |
| Action | INSERT_OR_UPDATE |
| Active | <input checked="" type="checkbox"/> |
| Description | |
| Field name | active |
| Global | <input checked="" type="checkbox"/> |
| Name | Testing Versions |
| Order | |
| Script | <pre>function onLoad() { alert("The form has finished loading and is ready for use!"); }</pre> |
| Current Version | |
| Action | INSERT_OR_UPDATE |
| Active | <input checked="" type="checkbox"/> |
| Description | |
| Field name | Priority |
| Global | <input checked="" type="checkbox"/> |
| Name | Testing Versions |
| Order | |
| Script | <pre>function onChange(control, oldValue, newValue, isLoading, isTemplate) { if (isLoading newValue === "") { return; } }</pre> |

The **Compare to Current** page is customizable.

Select the **Revert to Selected Version** button at the top of the form to overwrite the Current Version record with the Selected Version record.

Select the **Save Merge** button to save changes made to the Current Version while on the Compare to Current page.

Script Versions: Version Form

now.

- Select the **Compare to Current** or **Revert to this version** Related Link

The screenshot shows the 'Script Versions: Version Form' page. At the top, there are fields for 'Action' (set to 'INSERT_OR_UPDATE'), 'File path', 'Instance ID', 'Instance Name', and 'Name'. Below these are sections for 'Protocol' (containing a script), 'Payload' (containing '334552461'), and 'Record name' (set to 'Lab 3.5 Reference Object'). A 'Reverted from' field is also present. At the bottom, there is a 'Related Links' panel with three buttons: 'Compare to Current', 'Revert to this version', and 'Show Related record'. The 'Revert to this version' button is highlighted with a red box.

You can easily undo recent changes to a script by selecting the **Revert to this version** Related Link to revert to a previous version.

Client Scripts: Cloud Dimensions Requirements

now.

User Interface Requirements

- 1 Confirm Major Incident process is followed before a P1 is submitted
- 2 Enforce mandatory Incident requirements if State is Resolved or Closed



Database Requirements

- 3 Identify Incident records with RCA documentation
- 4 Populate Change's CAB date field with next CAB meeting date
- 5 Prevent Problems from re-opening if closed for more than 30 days
- 6 Update Problem and Child Incidents with RCA details from parent Incident
- 7 Implement SLA targets and identify Incidents in danger of breaching them
- 8 Automatically assign Incidents to Assignment group members

Security Requirements

- 9 Track Impersonations

Lab 2.2 fulfills requirement 1.

Client Scripts

now.

Good Practices

- Use the `g_form` methods to manage the form and form fields
- Use the `g_user` properties and methods to access information about the currently logged-in user
- Make as few calls to the server as possible
- Include a callback to make `getReference()` calls asynchronous
- Use `try/catch` to find runtime errors
- Use methods and debugging strategies appropriate to Mobile, Desktop, or both
- **Comment your scripts!!**

Additional resources for Client Script Best Practices:

- Search for the **Client script best practices** article on docs.servicenow.com.
- Search for the **Six ways to Improve the Performance of Client Scripts** blog article on community.servicenow.com.

Module Recap: Client Scripts

now.

| Core Concepts | Real World Cases |
|--|--|
| Client Scripts execute in the browser | • Why would you use these capabilities? |
| The g_form object has access to a form's fields and data | • When would you use these capabilities? |
| The g_user object has access to information about the currently logged-in user | • How often would you use these capabilities? |
| Reference records are retrieved with the getReference() method and a callback function | |
| Design your scripts for Mobile, Desktop, or both | |
| Quickly undo changes by reverting to an older version of a script | |

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Module 3: UI Policies

now.

| |
|--------------------|
| Scripting Overview |
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Define what a UI Policy is
- Know when to use UI Policies
- Write, test, and debug UI Policies
- Discuss Client Scripts versus UI Policies

Labs

Lab 3.1 Incident Resolved / Closed UI Policy

In this module you will write, test, and debug UI Policies.

What is a UI Policy?

- Defines the behavior and visibility of fields on a form
 - Mandatory
 - Visible
 - Read-only
- Condition must be true for it to execute
- Execute *after* Client Scripts

Baseline Problem form

| | |
|--------------------|---------------------------------|
| Number | PRB0000005 |
| Configuration item | Windows XP Hotfix (SP2) Q817606 |
| Priority | 2 - High |
| Change request | CHG0000003 |

After UI Policy executes

- 'Number' is read-only
- 'Configuration item' is mandatory
- 'Priority' is not visible

| | |
|--------------------|---|
| Number | PRB0000005 |
| Configuration item |  Windows XP Hotfix (SP2) Q817606 |
| Change request | CHG0000003 |
| Known error | <input type="checkbox"/> |

In the basic case UI Policies do not require scripting. In the example shown, no scripting was required to:

- Make the Number field read-only.
- Make the Configuration item field mandatory.
- Hide the Priority field.

Client Scripts can also hide fields, show fields, and make fields mandatory. If you can, always use a UI Policy instead of a Client Script for faster load times and easier maintenance.

FAQ's

What Exists Baseline?

- More than 1300 UI Policies exist baseline

Where can this be found?

- Navigate to the **System UI > UI Policies** module to create or modify UI Policies.



IMPORTANT

UI Policies do not run unless their condition is met. In some cases, UI Policies do not have a condition – these UI Policies **execute every time** a form loads and/or is changed.

What is UI Policy Scripting?

now.

- Ability to script complex conditions and execute advanced behavior
 - Show/Hide sections (tabs)
 - Remove/add/change/validate data in fields
- Full power of JavaScript

The screenshot shows a ServiceNow approval interface. At the top, it says "Approval CHG0030001". Below that, "Approver" is listed as "Fred Luddy" and "State" is set to "Rejected". A red box highlights the "Rejected" dropdown. To the right, "Approving" is listed with "Change Request: CHG0030001". Below the state dropdown, there is a field labeled "Comments" with a red asterisk (*) indicating it is mandatory. A red arrow points from the text "Comments are required when rejecting an approval" to this field. At the bottom right is a "Post" button.

Example:
UI Policy ensures Comments
are mandatory when an
Approval is rejected

`g_form.showFieldMsg()`
method used to include
instructions

UI Policy scripts execute client-side.

This example occurs when the baseline *Comments mandatory on rejection* UI Policy executes.

UI Policy Execution

now.

- Trigger specifies **when to execute**
- UI Policy Actions** and/or **Scripts** specifies **what to execute**
- Every field in the record can be evaluated even if it is not visible on the form (*UI16 and UI15*)
- UI Policies do not have a *Name* field
 - Use the **Short description** field as a pseudo name
- Not all UI Policies require scripting

The screenshot shows the 'UI Policy' form in 'Advanced view'. At the top, there's a table dropdown set to 'Incident (incident)'. Below it, a 'Short description' field contains 'Subcategory Mandatory If category == software'. A 'When to Apply' section is highlighted with a purple box, containing a 'Script' tab and a condition 'Category Is Software'. A 'What to execute' section is also highlighted with a purple box, showing a table of UI Policy Actions. One row in the table is selected, showing 'Fieldname' as 'subcategory', 'Mandatory' as 'True', 'Visible' as 'Leave alone', and 'Read only' as 'Leave alone'.

| Fieldname | Mandatory | Visible | Read only |
|-------------|-----------|-------------|-------------|
| subcategory | True | Leave alone | Leave alone |

The **Advanced view** of the UI Policy form is presented baseline. Select the **Default view** Related Link to remove the advanced fields.

There is a basic four step strategy to create a new UI Policy:

- Configure the UI Policy trigger:

Table – table to which the UI Policy applies.

Application – identifies the scope of the UI Policy.

Active – select the checkbox to make the UI Policy active.

Short description – supply a short explanation of what the UI Policy is for. The value in this field appears in the debugging information.

Order – the sequence in which UI policies are applied, from lowest to highest. Use when there are multiple UI Policies for the same table that may contain conflicting logic.

- Configure the the **When to Apply** conditions (when the UI Policy will run).

- Save** (not Submit) the record.

- Configure the **UI Policy Actions** (what the UI Policy will do):

Field name - field for which the UI Policy applies an action.

Mandatory - how the UI Policy affects the mandatory state of the field.

Visible – how the UI Policy affects the visible state of the field.

Read Only – how the UI Policy affects the read-only state of the field.



IMPORTANT

Although a UI Policy can evaluate any field in a record regardless if it is visible by the end user, the field specified in a **UI Policy Action** does need to be present on the form. For UI11, ensure any field called by a UI Policy exists on the form.

UI Policy Trigger – When to Apply

now.

- UI Policies execute based on evaluation of the Conditions
 - Build conditions with the Condition Builder rather than scripting for better performance
 - If left blank, the UI Policy logic always executes

The **Condition Builder** is not unique to UI Policies; the feature is used across the platform.

Global – the UI Policy applies to all form views if selected.

View – name of the specific form view to which the UI Policies applies. Visible when the Global field is not selected. If the Global field is not selected and the View field is left blank, the script applies the default view.

Reverse if false – UI Policy actions are reversed and the *Execute if false* script executed when its UI Policy conditions evaluate to false.

On load – execute on form load *and* form change.

Inherit – apply this script to any extended tables when selected.



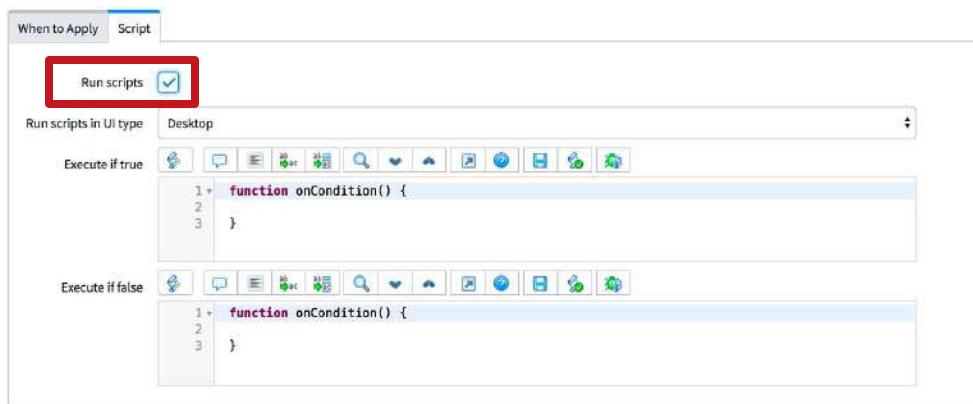
IMPORTANT

Conditions are only re-evaluated if a user manually changes a field on the form; if the change is made by the system it will not be rechecked. Use **Data Policies**, which are not scriptable, to manage the mandatory and read-only state of fields for records not changed on a form.

UI Policy Trigger – Script

now.

- Select **Run scripts** on the *Script* tab to access the scripting fields
- Can write a script in one or both fields
- UI Policy Scripts execute client-side



Run scripts in UI type – select whether the script executes for *Desktop* only, *Mobile/Service Portal*, only or *All* environments.

Execute if true – JavaScript executes when the UI Policy Condition tests true.

Execute if false – JavaScript executes when the UI Policy Condition tests false.

The *onCondition()* function in the appropriate UI Policy script fields is automatically called when the condition returns true or false.



IMPORTANT

The **Reverse if false** (*located on the 'When to Apply' tab*) field must be selected in order for the **Execute if false** script to execute.

What Data Can You See in a UI Policy?

now.

- Local variables declared in a script
- Predefined client-side Global variables such as
 - **g_form** (GlideForm)
 - **g_user** (GlideUser)
 - **g_scratchpad** (in conjunction with a Display Business Rule, Module 5)



ServiceNow client-side global variables:

g_form – object whose properties are methods used to manage the form and its fields in the record.

g_user – object whose properties contain session information about the currently logged in user and their role(s).

g_scratchpad – object passed to a UI Policy from a server-side script known as a Display Business Rule. The object's properties and values are determined by the server-side script.

Debug UI Policies

- Must be enabled/disabled
 - System Diagnostics > Session Debug > Debug UI Policies
 - System Diagnostics > Session Debug > Disable UI Policies Debug
- Output appears in the JavaScript Log window
- Shows evaluation of condition and script processing

JavaScript Log

Field Watcher

Small Medium Large

15:25:11 (886) incident.do dirty form focus

15:25:19 (608) incident.do>1

15:25:19 (609) incident.do GlideFieldPolicy: Running "Subcategory Mandatory IF category == software" UI Policy on "Incident" table

15:25:19 (609) incident.do GlideFieldPolicy: >>> evaluating conditions:

15:25:19 (610) incident.do GlideFieldPolicy: > category's value of "software" with the condition(= software) evaluates to TRUE

15:25:19 (610) incident.do GlideFieldPolicy: <<< condition exited with: TRUE

15:25:19 (612) incident.do GlideFieldPolicy: Setting "mandatory" to "true" on "subcategory" field

UI Policy Action Processing

UI Policy Short Description

Table

Condition Evaluation Result

Individual Condition Line Item Evaluation

All of the Client Script debugging strategies reviewed in the previous module also work for UI Policies. In addition, UI Policies have the **Debug UI Policies** option. This is the only debugging strategy for seeing the evaluation of a UI Policy's Condition.

Module Labs

now.

- **Lab 3.1**

- **Time:** 20-25m
- Incident Resolved / Closed UI Policy
 - You will write, test, and debug a UI Policy



Incident Resolved / Closed UI Policy

Lab
03.01
⌚20-25m

Lab Summary

You will achieve the following:

- Write, test, and debug a UI Policy
- The UI policy should:
 - Trigger when the State is set to Resolved.
 - Change Urgency and Impact fields to read-only.
 - Change the Resolved by field to mandatory.
 - Display an Info Message.
- A Cloud Dimensions' Phase II requirement will be complete.



Business Problem

1. When closing Incident records, IT analysts are changing the value of the Impact and Urgency fields in an effort to manipulate SLA results.
2. In addition to the baseline mandatory fields when a record is put in a Resolved or Closed state, the Incident Management team would also like to consistently capture who is resolving Incidents for reporting purposes.
3. IT Analysts do not always remember to populate the Closure Information fields as they are on a different tab, and are frustrated with the amount of times they are presented with the 'Mandatory fields were missed' alert after they attempt to save the record. They have requested a message on the form, reminding them which fields are mandatory before they attempt to save the record.

Project Requirement

Because every requirement is based on the same Incident condition, only one UI Policy is required to solve all of the business problems. When an Incident's state is Resolved or Closed:

- Make the Impact and Urgency fields read-only.
- Make the Closed by field mandatory.
- Configure an Info Message reminding users which fields are mandatory before they attempt to save a record.

A. Create a UI Policy

1. Navigate to System UI > UI Policies.
2. Create a new UI Policy.

Table: **Incident [incident]**

Active: **Selected (checked)**

Short Description: **Lab 3.1 Incident Resolved or Closed**

Order: **100**

Condition: **State | is one of | Resolved + Closed**

(Hold the 'Shift' key down to select both items in the list)

Global: **Selected (checked)**

On load: **Not selected (not checked)**

Reverse if false: **Selected (checked)**

Inherit: **Not selected (not checked)**

The screenshot shows the 'UI Policy' creation interface. The 'Table' is set to 'Incident [incident]'. The 'Short description' is 'Lab 3.1 Incident Resolved or Closed'. The 'Order' is '100'. The 'When to Apply' tab is selected, showing a 'Conditions' section. In the 'Conditions' section, 'State' is selected, and 'is one of' is chosen. A modal window displays the options 'In Progress', 'On Hold', 'Resolved', 'Closed', and 'Canceled', with 'Resolved' and 'Closed' being the ones highlighted in blue. Other tabs like 'Script' and 'Global' are also visible.

3. **Save** the record, remain on the form.

4. Add a UI Policy Action.

a) Scroll down to the *UI Policy Actions* Related List and select the **New** button.

b) Configure the UI Policy Action:

Field name: **Urgency**

Mandatory: **Leave alone**

Visible: **Leave alone**

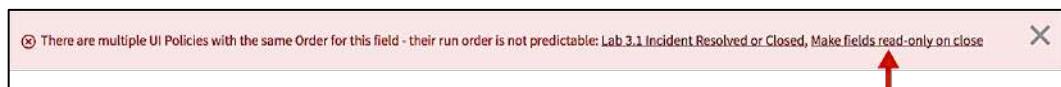
Read Only: **True**

c) Select **Submit**.

d) Examine the record you just created in the list of *UI Policy Actions*. Notice the red dot to the left of **urgency**? This indicates there is an error that requires attention. Select the **urgency** link to open the record.

| UI Policy Actions | | | | | |
|---|-------------|-----------------|-----------|--|--|
| | New | Search for text | Search | | |
| UI policy = Lab 3.1 Incident Resolved or Closed | | | | | |
| Field name | Mandatory | Visible | Read only | | |
| urgency | Leave alone | Leave alone | True | | |

e) An error message appears indicating there are multiple UI Policies with the same Order for this field. Select the **Make fields read-only on close** link in the error message to open the conflicting UI Policy record.



f) If the *Default* view of the form opens, select the **Advanced view** Related Link to display the Order field. Record the **Order** of the *Make fields read-only on close* UI Policy here:

g) Return to the **Lab 3.1 Incident State Resolved** UI Policy. If the *Default* view of the form opens, select the **Advanced view** Related Link.

h) Change the **Order** field value to a value greater than the conflicting UI Policy Order field's value.

i) **Save** the record, remain on the form.

j) Notice the red dot to the left of the urgency *UI Policy Action* is no longer visible.

- Add two additional UI Policy Actions:

Impact: Read only
Closed by: Mandatory

| UI policy = Lab 3.1 Incident Resolved or Closed | Field name | Mandatory | Visible | Read only |
|---|------------------------------------|-------------|-------------|-------------|
| | <input type="checkbox"/> urgency | Leave alone | Leave alone | True |
| | <input type="checkbox"/> impact | Leave alone | Leave alone | True |
| | <input type="checkbox"/> closed_by | True | Leave alone | Leave alone |

- On the **Script** tab, Select (check) the **Run scripts** field.
- Examine the following pseudo-code for the **Execute if true** script you will write:
 - When the condition is true
 - If the Closed code, Closed notes, or Resolved by fields do not have values
 - Display an Information Message reminding the analysts mandatory fields are required before saving the record.
- Write the **Execute if true** script:

```
function onCondition() {
    if(g_form.getValue('close_code')==='' || g_form.getValue('close_notes')==='' || g_form.getValue('resolved_by')==='') {
        g_form.addInfoMessage("REMINDER: Populate the Resolution Information fields before saving an Incident in a Resolved or Closed State");
    }
}
```

- Select **Update**.

B. Test Your Work

- Open an existing Incident.
- Set the value of *State* to **Resolved** or **Closed**.
- Did the Information Message appear at the top of the form? If not, debug and re-test.
- If the user updating a record selects one of these two states in error and immediately sets the value of *State* back to **In Progress**, does the Info Message at the top of the form disappear?
- Open the **Lab 3.1 Incident Resolved or Closed UI Policy**.

6. Write the **Execute if false** script:

```
function onCondition() {  
    g_form.clearMessages();  
}
```

7. Select **Update**.
8. Open an existing Incident.
9. Set the value of the State field to **Resolved** or **Closed**.
10. Once the Info Message appears at the top of the form, update the value of *State* to **In Progress**? Does the Info Message at the top of the form disappear? If not, debug and re-test.
11. Does the **Reverse if false** field need to be selected in order for the **Execute if false** script to execute?

-
12. When a UI Policy condition is not met and the **Reverse if false** field is selected, does the reverse of what is scripted in the Execute if true field occur?
-
-

C. Practice Debugging a UI Policy

1. Select **System Diagnostics > Session Debug > Debug UI Policies** to enable UI Policy debugging.
2. Open the **JavaScript Log and Field Watcher** pane.
3. Open an existing Incident.
4. Select the **Clear log** button () to remove any messages in the JavaScript Log.
5. Force the UI Policy to execute again by setting the value of *State* to either **Resolved** or **Closed**.
6. Locate the debugging information in the *JavaScript Log* for the **Lab 3.1 Incident Resolved/Closed** UI Policy.

7. Examine the debugging information for the evaluation of the condition of this lab's UI Policy. What did the condition evaluate to?
-

8. Are the correct fields read-only or mandatory?
-

9. Select the **Clear log** button () to remove any messages from the JavaScript Log.

10. Set the value of *State* to **In Progress**.

11. Locate the debugging information in the *JavaScript Log* for the **Lab 3.1 Incident Resolved/Closed** UI Policy.

12. Examine the debugging information for the evaluation of the condition. What did the condition evaluate to?
-

13. Examine the debugging information for the UI Policy Actions. Were the correct fields made read-only or mandatory?
-

14. Close the JavaScript Log and Field Watcher pane.

15. Disable UI Policy Debugging by selecting **System Diagnostics > Session Debug > Disable UI Policies Debug** on the Application Navigator.

16. Make the Lab 3.1 Incident Resolved/Closed UI Policy **inactive**.

Lab Completion

Great job! You successfully created a UI Policy that works everywhere in the platform.

UI Policies: Cloud Dimensions Requirements

now.

User Interface Requirements

- | | |
|--|---|
| 1 Confirm Major Incident process is followed before a P1 is submitted | ✓ |
| 2 Enforce mandatory Incident requirements if State is Resolved or Closed | ✓ |

Database Requirements

- | |
|--|
| 3 Identify Incident records with RCA documentation |
| 4 Populate Change's CAB date field with next CAB meeting date |
| 5 Prevent Problems from re-opening if closed for more than 30 days |
| 6 Update Problem and Child Incidents with RCA details from parent Incident |
| 7 Implement SLA targets and identify Incidents in danger of breaching them |
| 8 Automatically assign Incidents to Assignment group members |

Security Requirements

- | |
|------------------------|
| 9 Track Impersonations |
|------------------------|

Lab 3.1 fulfills requirement 2.

Client Scripts versus UI Policies

now.

| | Client Script | UI Policy |
|---------------------------------------|---------------|-----------|
| Execute on form load | ✓ | ✓ |
| Execute on form save/submit/update | ✓ | |
| Execute on form field value change | ✓ | ✓ |
| Have access to a field's prior value | ✓ | |
| Execute on list field value change(s) | ✓ | |
| Execute after Client Scripts | | ✓ |
| Require scripting | ✓ | |

Client-side scripts manage forms and their fields. Client Scripts and UI Policies both execute client-side and use the same API. Use this table to determine which script type is best suited to your application needs.

UI Policies

now.

Good Practices

- Set onLoad to false if you do not need to execute on page load
- Use as few UI Policies as possible to avoid long page load times
- Apply conditions using the Condition Builder whenever possible so unnecessary UI Policy scripts do not load
- Use the Short description field to document the UI Policy
- Add the Description field to the form to thoroughly document the UI Policy
- **Comment your scripts!!**

Module Recap: UI Policies

now.

Core Concepts

UI Policies are used for client-side form management

UI Policies take actions based on scripts and/or UI Policy Actions

UI Policies can be created for tables in the same scope

Debug UI Policies using logging, JavaScript, and UI Policy Debugging

Different actions are available if the condition returns true or false

Real World Cases

• **Why** would you use these capabilities?

• **When** would you use these capabilities?

• **How often** would you use these capabilities?

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Module 4: Catalog Client Scripts & Catalog UI Policies

now.

| |
|--------------------------|
| Scripting Overview |
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Define what it means to be a Catalog Client Script and a Catalog UI Policy
- Learn how to create Catalog Client Scripts and Catalog UI Policies
- Explore how to use Catalog Client Scripts and Catalog UI Policies in Variable Sets
- Discuss the data available to Catalog Client Scripts and Catalog UI Policies

Labs

Lab 4.1 Control Variable Choices Catalog Client Script

Lab 4.2 Control Out of State Shipping Catalog UI Policy

What is a Catalog Client Script?

- Manages the behavior of **Catalog Items** when presented to users

Department or Group Catalog Client Script manages the display of certain fields on the form based on the value selected in this Multiple Choice field

Populate Manager and Name Catalog Client Script auto-populates the Name and Manager fields based on the value selected in the Group field

Catalog client scripts provide form validation and dynamic effects. This control of the form makes the catalog reactive to a user's inputs for an easy, curated, and rich user experience.

This is accomplished with Client Scripts that execute on the client-side (in your web browser).

Be aware that different browsers may present items in different ways. Always test your work in the different browsers where possible (including mobile devices).

FAQ's

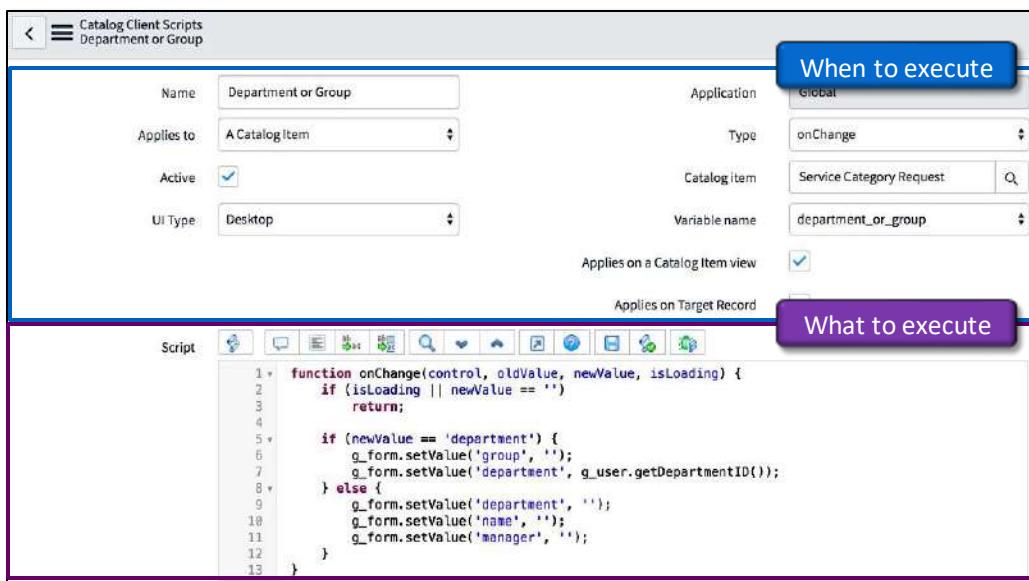
What exists baseline?

- 18 Catalog Client Scripts exist baseline

Where can they be found?

- Navigate to the **Service Catalog > Catalog Administration > Catalog Client Scripts** module to create or modify Catalog Client Scripts

Create a Catalog Client Script



Name: Unique name for the Catalog Client Script.

Applies to: Select the item type this Catalog Client Script applies to (*A Catalog Item* or *A Variable Set*).

Active: Select the check box to enable the Catalog Client Script. Clear the check box to disable the script.

UI Type: Identify where the Catalog Client Script executes (*Desktop*, *Mobile/Service Portal*, or *All*).

Application: Identify the scope of the Catalog Client Script.

Type: Identify when the Catalog Client Script executes (*onChange*, *onLoad*, or *onSubmit*).

Catalog item or Variable set: Select a Catalog Item or Variable Set from the list.

Variable name: Identify which field to watch for changes when *onChange* is selected as the *Type*.

Applies on a Catalog Item view: Select the check box to apply the Catalog Client Script to Catalog Items displayed within the order screen on the *Service Catalog*. Available in the Requester view.

Applies on Requested Items: Select the check box to apply the Catalog Client Script on a *Requested Item* form, after the item is requested. Available in the Fulfiler view.

Applies on Catalog Tasks: Select the check box to apply the Catalog Client Script when a *Catalog Task* form for the item is being displayed. Available in the Fulfiler view.

Applies on the Target Record: Select the check box to support the Catalog UI Policy on a record created for task-extended tables via Record Producers.

Script: Script *what* needs to happen when the conditions in the trigger are met.

What is a Catalog UI Policy?

now.

- Manages the **behavior of Catalog Items** when presented to your users
- Condition must be true for the policy to execute
- Offers the use of the **Condition builder** to easily configure a condition vs. scripting it

Variables in a Catalog UI Policy condition must be visible (even if it is hidden by another UI policy or read-only) on the form for the condition to be tested. Catalog UI Policies can also be applied when the variables are present in a Requested Item or Catalog Task form.

Catalog UI Policies without a condition execute every time a form loads and/or is changed.

Limited UI Policy functionality applies to the following variables:

- The **Mandatory** and **Read only** policies do not apply to the following variable types: Break, Container Split, Container End, UI Macro, UI Macro with Label, Label, UI Page.
- The **Visible** policy does not apply to the following variable types: Break, Container Split, Container End.



What Exists Baseline?

- 27 Catalog UI Policies exist baseline

Where can they be found?

- Navigate to the **Service Catalog > Catalog Administration > Catalog UI Policies** module to create or modify Catalog UI Policies.

IMPORTANT

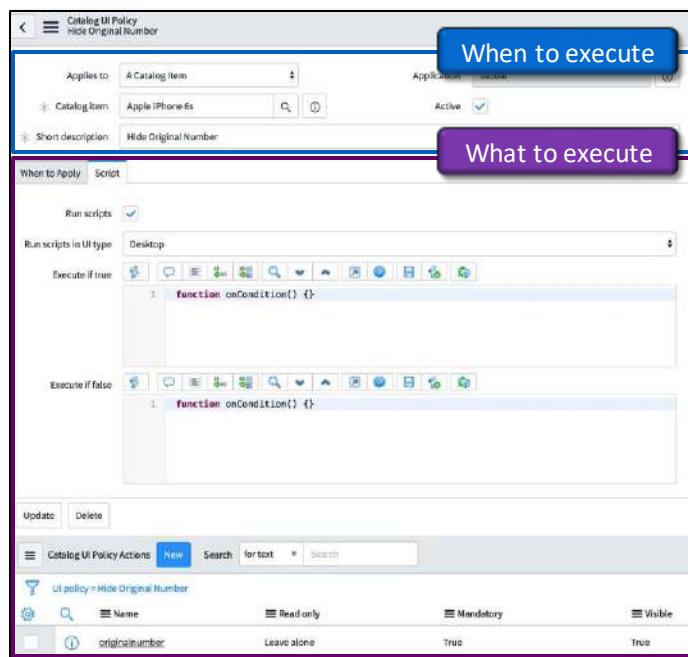
Not all Catalog UI Policies require scripting!



Create a Catalog UI Policy

now.

- Use the Condition builder on the **When to Apply** tab to configure simple conditions
- Configure simple **Catalog UI Policy Actions**
 - Mandatory
 - Read only
 - Visible
- Use the **Execute if true** and **Execute if false** fields to script complex conditions and actions



Applies to: Select the type of item this UI Policy applies to (A Catalog Item or A Variable Set).

Catalog item/Variable set: Select the catalog item or a variable set this UI Policy applies to.

Short description: Enter a brief description of the Catalog UI Policy. Use this field as a pseudo name.

Application: Identifies the scope of the Catalog UI Policy.

Active: Select the check box to enable the Catalog UI Policy. Clear the check box to disable it.

Order: Sequence in which policies are applied, from lowest to highest. Use when multiple policies for the same table contain conflicting logic. Configure the form to display the existing field.

When to Apply

Catalog Conditions: Use the Condition builder to configure simple conditions for the policy using Catalog Item variables. The policy is applied if the conditions evaluate to true.

Applies on a Catalog Item view: Select the check box to apply the policy to Catalog Items on the order screen.

Applies on Catalog Tasks: Select the check box to apply the policy on a Catalog Task form.

Applies on Requested Items: Select the check box to apply the policy on a Requested Item form.

Applies on the Target Record: Select the check box to support the policy on a record created for task-extended tables via Record Producers.

On load: Select the check box to apply the policy when the form loads, clear the check box to apply the policy only when the form changes.

Reverse if false: Select check box to reverse the policy if the *Catalog Conditions* evaluate to false.

Script

Run scripts: Select the check box to script complex conditions and actions.

Run scripts in UI Type: Identify where policy executes (*Desktop*, *Mobile/Service Portal*, or *All*).

Execute if true: Script to execute when the *Catalog Conditions* evaluate to true. If *Catalog Conditions* is left blank, script in this field always executes. Use to script complex conditions.

Execute if false: Script to execute when the *Catalog Conditions* evaluate to false.

Variable Sets

- Catalog Client Scripts and Catalog UI Policies can also be applied to Variable Sets
- Execute every time the Variable Set is used

The screenshot shows the 'Variable Set' configuration page for 'Standard Employee Questions'. The 'Catalog UI Policies' tab is selected, indicated by a red box around it. The page displays two variables:

| Name | Type | Description |
|---------------|------------|--------------------------|
| requested_for | Reference | Who is this request for? |
| needed_by | Select Box | When do you need this? |

Recall from ServiceNow Fundamentals training, a Variable Set allows you to define a set of variables to be stored as a Variable Set for future use.

A Catalog UI Policy for Catalog Items always takes precedence over Catalog UI Policies for Variable Sets.

Search for the **Service catalog variable sets** article on docs.servicenow.com for more information on Variable Sets.

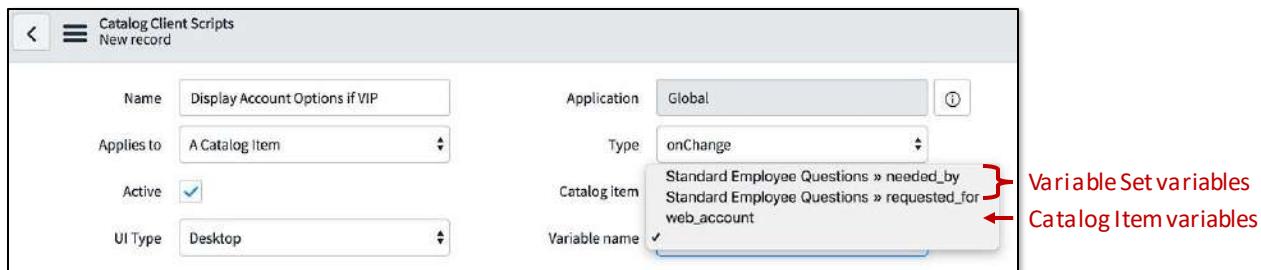


IMPORTANT

Applying Catalog UI Policy Actions on a Variable Set is NOT supported on Service Portal.

Variable Sets

- Both Catalog Item variables and Variable Set variables are available to Catalog Client Scripts and Catalog UI Policies
- Allows you to manage the behavior of any variable at the Catalog Item level



If a Variable Set **behaves the same** each time it is used, manage its behavior with a Catalog Client Script or Catalog UI Policy at the **Variable Set** level.

If a Variable Set **behaves differently** each time it is used, manage its behavior with a Catalog Client Script or Catalog UI Policy at the **Catalog Item** level.

What Data Can You See in Catalog Client Scripts and Catalog UI Policies?

- Local variables declared in a script
- Predefined client-side Global variables such as
 - **g_form** (GlideForm)
 - **g_user** (GlideUser)
 - **g_scratchpad** (in conjunction with a Display Business Rule, Module 5)



ServiceNow client-side global variables:

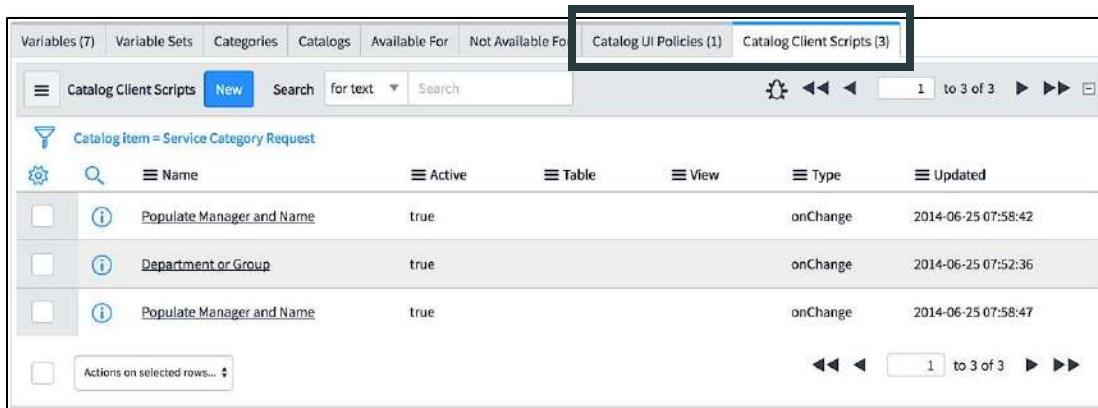
g_form – object whose properties are methods used to manage form and its fields in the record.

g_user – object whose properties contain session information about the currently logged in user and their role(s).

Quickly Locate Catalog Item Elements

now.

- Use the **Catalog Client Script** and **Catalog UI Policies** Related Lists
- Identifies the related elements of the **current Catalog Item**
- Bottom of the Catalog Item form



The screenshot shows two related lists side-by-side. The left list is titled 'Catalog Client Scripts' and contains three entries. The right list is titled 'Catalog UI Policies' and also contains three entries. Both lists have columns for Name, Active, Table, View, Type, and Updated. The 'Catalog Client Scripts' list includes a 'Actions on selected rows...' button at the bottom. A black box highlights the 'Catalog Client Scripts' tab and the 'Catalog UI Policies' tab.

| Name | Active | Table | Type | Updated |
|---------------------------|--------|-------|----------|---------------------|
| Populate Manager and Name | true | | onChange | 2014-06-25 07:58:42 |
| Department or Group | true | | onChange | 2014-06-25 07:52:36 |
| Populate Manager and Name | true | | onChange | 2014-06-25 07:58:47 |

| Name | Active | Table | Type | Updated |
|---|--------|-------|----------|---------------------|
| Catalog item = Service Category Request | true | | onChange | 2014-06-25 07:58:42 |
| Populate Manager and Name | true | | onChange | 2014-06-25 07:58:47 |
| Department or Group | true | | onChange | 2014-06-25 07:52:36 |

You can also use the Application Navigator to locate the Catalog Client Script and Catalog UI Policies modules. You can save the modules as favorites too!

Module Labs

now.

- **Lab 4.1**

- **Time:** 15-20m
- Control Variable Choices Catalog Client Script
 - Practice scripting using Catalog Client Scripts and Catalog UI Policies

- **Lab 4.2**

- **Time:** 20-25m
- Control Out of State Shipping Catalog UI Policy
 - Practice using a Catalog UI Policy to easily configure the display of specific location fields



Control Variable Choices Catalog Client Script

Lab
04.01
⌚15-20m

Lab Summary

You will achieve the following:

- Practice scripting using Catalog Client Scripts and Catalog UI Policies.
- You will first create a new Catalog Item for users to request new toner cartridges.
Users are able to order individual cartridges, as well as a multipack option with different colors.
- Rather than create separate Catalog Items for the different options, you will create a single Catalog Item to contain the various options, then use a script to filter the options according to the role of the logged-in user.
 - All users may only order black printer cartridges.
 - Users with the *ITIL Admin* role may order all printer cartridges (black and color).

A. Preparation

1. Navigate to **Self-Service > Knowledge**.
2. Use the Search field to look for **Toner**.
3. Select the **Toner.png** article. The file automatically downloads to your computer.

B. Create a New Catalog Item

1. Navigate to **Service Catalog > Catalog Definitions > Maintain Items**.
2. Create a new Catalog Item.

Name: **Brother Network-Ready Printer Toner**
Catalogs: **Service Catalog**
Category: **Printers**
Workflow: **Service Catalog Item Request**
Short Description: **Brother TN 2000 Series Toner**
Description: **Yields approximately 1200 pages**
Picture: **Toner.png** (downloaded earlier in the lab)

3. **Save** the record, remain on the form.

4. Select the **Additional Actions** () > View > Advanced.

C. Create a Variable to Contain the Available Cartridge Types

1. Select **New** on the *Variables* Related List.

Type: **Select Box**

Order: **100**

On the **Question** Tab

Question: Which pack type would you like to order?

Name: **pack_type**

On the **Type Specification** Tab

Include none: **Selected (checked)**

2. **Save** the record, remain on the form.



TIP FROM
THE FIELD:

The **Question** is actually a label for the variable. In a Catalog Client Script, the **Name** is used to reference the variable. Since the **Question** will not be in the script, the **Name** should be a good description of the variable and unique.

3. Select **New** on the *Question Choices* Related List.

Price: **60.00**

Order: **100**

Text: **Black Cartridge – Single**

Value: **black1**

4. Select **Submit**.

5. Repeat steps 3 and 4 to add these additional Question Choices.

| Price | Order | Text | Value |
|--------|-------|---|--------|
| 260.00 | 200 | Black Cartridges – 5 Pack | black5 |
| 150.00 | 300 | CYM Cartridges – 3 Pack (1 of each color) | cym3 |
| 250.00 | 400 | CYM Cartridges – 6 Pack (2 of each color) | cym6 |

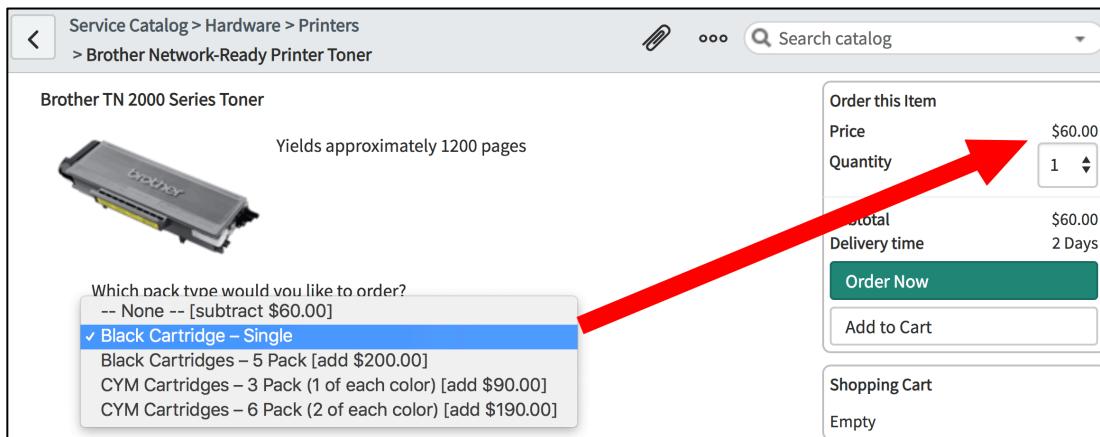
6. Compare your **Which pack type would you like to order?** Variable record with the image below, make any adjustments if necessary.

The screenshot shows the configuration of a variable named 'Which pack type would you like to order?'. The variable is of type 'Select Box' and is set to 'Global'. It is mandatory and active. The order is set to 100. The question text is 'Which pack type would you like to order?' and the name is 'pack_type'. There is no tooltip defined. The variable has four question choices:

- Black Cartridge – Single (Value: black1, Price: \$60.00, Order: 100)
- Black Cartridges – 5 Pack (Value: black5, Price: \$260.00, Order: 200)
- CYM Cartridges – 3 Pack (1 of each color) (Value: cym3, Price: \$150.00, Order: 300)
- CYM Cartridges – 6 Pack (2 of each color) (Value: cym6, Price: \$250.00, Order: 400)

Note: Consider configuring the *Question Choices List Layout* to include the **Price** and **Order** columns. Select any Column header Context menu, then select **Configure > List Layout**.

7. Select **Update**
8. Select **Try It** on the *Brother Network-Ready Printer Toner* Catalog Item's Header bar.
9. Verify the price information in the shopping cart changes as expected.



D. Create a Catalog Client Script to Control the Display of Choices Based on Role

Now that the Catalog Item is in place, you can create the script to remove the choices as described in the lab goal.

1. On the **Brother Network-Ready Printer Toner** Catalog Item form, configure the related lists to include *Catalog Client Scripts*
2. Scroll down to the *Catalog Client Scripts* Related List and select **New**.
3. Configure the record.

Name: **Toner Choices by Role**

Type: **onLoad**

Note: Notice the **Applies on a Catalog Item view** field is pre-populated. This is because you navigated to this form from a Catalog Item record.

4. **Save** the record, remain on the form.
5. Examine the pseudo-code for the script you will write.
 - If the logged-in user does not have the `itil_admin` role
 - Hide the CYM Cartridges – 3 Pack (1 of each color) [cym3] option
 - Hide the CYM Cartridges – 6 Pack (2 of each color) [cym6] option

6. Write the script.

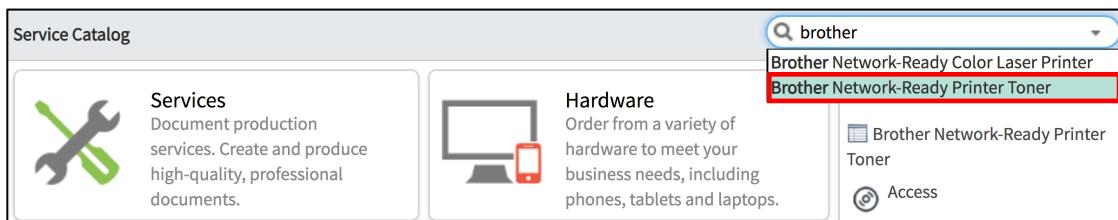
```
function onLoad() {  
    if (!g_user.hasRoleExactly('itil_admin')) {  
        g_form.removeOption('pack_type','cym3');  
        g_form.removeOption('pack_type','cym6');  
    }  
}
```

7. Select **Update**.

E. Test Your Work

The System Administrator does not have the 'itil_admin' role, therefore you do not have to impersonate anyone to confirm users without this role can only order black printer cartridges.

1. On the Brother *Network-Ready Printer Toner Catalog Item*'s Header bar, select **Try It**.
2. Confirm only black printer cartridges are available for selection. If not, debug and re-test.
3. Impersonating **Bow Ruggeri** (this user has the 'itil_admin' role).
4. Navigate to **Self-Service > Service Catalog**.
5. Use the **Search catalog** field to locate the *Brother Network-Ready Printer Toner Catalog Item*. Select it when it appears in the drop-down list.



6. Select the **Brother Network-Ready Printer Toner** name to open the Catalog Item.

 **Brother Network-Ready Printer Toner**

Brother TN 2000 Series Toner

▼ preview



Yields approximately 1200 pages

Service Catalog > Hardware > Printers

7. Confirm Bow can order both black and CYM printer cartridges. If not, debug and re-test.
8. End the impersonation. You should be logged in as the System Administrator when this step is complete.

Lab Completion

Excellent work! You have successfully controlled available items for order using a Catalog Client Script.

Control Out of State Shipping Catalog UI Policy

Lab
04.02
⌚20-25m

Lab Summary

You will achieve the following:

- Use a Catalog UI Policy to easily configure the display of specific location fields.
- The Catalog UI policy should allow:
 - Regular users order a 'Brother Network-Ready Printer Cartridge', they are only provided a list of California locations as delivery options.
 - Users with 'itil_admin' role can choose location outside of California, however they will receive a message reminding them additional charges will be applied to the receiving location.

Lab Dependency: You must have *Lab 4.1 Control Choices with a Catalog Client Script* complete, as this exercise continues to build on the Catalog Item created in the previous lab.

A. Create a Variable to Confirm a California Delivery

1. Navigate to Service Catalog > Catalog Definitions > Maintain Items.
2. Locate and open the Brother Network-Ready Printer Toner Catalog Item.
3. Select **New** on the *Variables* Related List.

Question

Type: **Select Box**

Order: **200**

Question: **California delivery?**

Name: **ca_location**

Default Value

Default value: **yes**

4. **Save** the record, remain on the form
5. Select **New** on the *Question Choices* Related List.

Order: **100**

Text: **Yes**

Value: **yes**

6. Select **Submit**.
7. Repeat steps 5-6 to add a **NO** Question Choice as well.

Order: **200**

Text: **No**

Value: **no**

B. Create Two Variables to Provide Location Selections

1. On the **Brother Network-Ready Printer Toner Catalog** Item form, select **New** on the *Variables* Related List.
2. Create a California locations only reference field.

Type: **Reference**

Order: **300**

Question

Question: **Select a California location for delivery**

Name: **location_ca**

Type Specifications tab

Reference: **Location [cmn_location]**

Reference qualifier condition: **State/Province | contains | CA**

3. Select **Submit**.
4. Select **New** on the *Variables* Related List again.
5. Create a non-California locations reference field.

Type: **Reference**

Order: **400**



TIP FROM
THE FIELD:

Notice the **Text** is mixed case and the **Value** is lowercase. **Text** (like a field label) is just for display. The **Value** should adhere to the rules of the language you are writing for. It is a good habit to be consistent with **Values**, so use lowercase when possible.

Question

Question: **Select a location outside of California for delivery**

Name: **location_other**

Type Specifications tab

Reference: **Location [cmn_location]**

Reference qualifier condition: **State/Province | does not contain | CA**

6. Select **Submit**.
7. Compare your **Brother Network-Ready Printer Toner** Variable records with the image below; make any adjustments if necessary.

The screenshot shows the 'Variables' tab of a catalog client script for the 'Brother Network-Ready Printer Toner' catalog item. The interface includes a toolbar with 'Includes', 'Variables (4)', 'Available For', 'Not Available For', and 'Catalog Client Scripts (1)'. Below the toolbar is a search bar and a navigation bar with icons for back, forward, and search. The main area displays a table of variables:

| | Type | Question | Order ▲ |
|--------------------------|----------------------------|---|---------|
| <input type="checkbox"/> | Select Box | Which pack type would you like to order? | 100 |
| <input type="checkbox"/> | Select Box | California delivery? | 200 |
| <input type="checkbox"/> | Reference | Select a California location for delivery | 300 |
| <input type="checkbox"/> | Reference | Select a location outside of California ... | 400 |

C. Restrict the Alternate Location Selections to Users With the 'itil_admin' Role

1. On the **Brother Network-Ready Printer Toner** Catalog Item form, open the **Toner Choices by Role** Catalog Client Script you created in Lab 4.1.
2. Add statements to the existing script to also restrict the display of the **California delivery? [ca_location]** and **Select a location outside of California for delivery [location_other]** fields to users with the 'itil_admin' role.

```
function onLoad() {
    if (!g_user.hasRoleExactly('itil_admin')) {
        g_form.removeOption('pack_type','cym3');
        g_form.removeOption('pack_type','cym6');
        g_form.setDisplay('ca_location',false);
        g_form.setDisplay('location_other',false);
    }
}
```

3. Select **Update**.

D. Use a Catalog UI Policy to Display the Correct Location Fields

Since the Catalog Client Script is already controlling who can view the **California delivery [ca_location]** and **Select a location outside of California for delivery [location_other]** fields, only one Catalog UI Policy is needed to display the correct Location field to the users who can see the fields, as well as script the display of a user-friendly Field messages.

1. From the **Brother Network-Ready Printer Toner** Catalog Item form, configure the related lists to include *Catalog UI Policies*.

2. Select **New** on the *Catalog UI Policies* Related List.

Short description: **Display Location Options**

Catalog Conditions: **ca_location | is | No**

3. **Save** the record, remain on the form.

4. Select **New** on the *Catalog UI Policy Actions* Related List.

Variable name: **location_ca**

Visible: **False**

5. Select **Submit**.

6. Add one more Catalog UI Policy Action.

Variable name: **location_other**

Visible: **True**

7. Compare your **Catalog UI Policy Actions** with the image below; make any adjustments if necessary.

| Catalog UI Policy Actions | | | | | |
|--------------------------------------|--|----------------|-------------|-------------|---------|
| | | Name | Read only | Mandatory | Visible |
| UI policy = Display Location Options | | | | | |
| <input type="checkbox"/> | | location_other | Leave alone | Leave alone | True |
| <input type="checkbox"/> | | location_ca | Leave alone | Leave alone | False |

E. Inform Users Additional Shipping Fees Outside of California Apply

1. Select the **Script** tab on the *Display Location Options Catalog* UI Policy form.
2. **Select (check)** the *Run scripts* field.
3. Examine the pseudo-code for the scripts you will write:
 - If the item is not being delivered to a California location
 - Add a field message below the 'Select a Location' field advising the user additional shipping fees apply to out of State deliveries
 - If the item is being delivered to a California location
 - Ensure no field messages are on the form
4. Write the **Execute if true** script.

```
function onCondition() {
    //Using 'error' as the Type for high impact on the form
    g_form.showFieldMsg('location_other','PLEASE NOTE: Additional
    shipping fees apply to out of State deliveries. The location you
    select in this field will be charged any additional
    costs.','error');
}
```

5. Write the **Execute if false** script.

```
function onCondition() {
    //Removes exiting Msgs if user updates 'California location?' to yes
    g_form.hideAllFieldMsgs();
}
```

6. Select **Update**.

F. Test Your Work

The System Administrator does not have the 'itil_admin' role, therefore you do not have to impersonate anyone to confirm that non-California delivery fields do not display on the Catalog Item form.

1. Select **Try It** on the *Brother Network-Ready Printer Toner Catalog* Item's Header bar.
2. Complete the **Which pack type would you like to order?** and **Select a California location for delivery** fields. Were the fields displayed on the form? If not, debug and re-test.

3. Impersonating **Bow Ruggeri** (this user has the 'itil_admin' role).
4. Locate and open the **Brother Network-Ready Printer Toner Catalog Item**.
5. If these three fields do NOT display on the form, debug and re-test.
 - Which pack type would you like to order?
 - California delivery?
 - Select a California location for delivery
6. Update the value in the *California delivery?* field to **No**. If the following three actions do NOT occur, debug and re-test.
 - The *Select a California location for delivery* is NOT visible.
 - The *Select a location outside of California for delivery* is visible.
 - The fieldMsg advising users of the additional shipping fees is visible below the *Select a location outside of California for delivery* field in red.
7. Update the value in the *California delivery?* field to **Yes**. If the following three actions do not occur, debug and re-test.
 - The *Select a California location for delivery* is visible.
 - The *Select a location outside of California for delivery* is NOT visible.
 - The fieldMsg no longer appears on the form.
8. End the impersonation. You should be logged in as the *System Administrator when this step is complete*.

Lab Completion

Great job! You successfully implemented one single Catalog UI Policy to perform numerous actions. It is always considered good practice to use a Catalog UI Policy vs. a Catalog Client Script whenever possible.

Catalog Client Scripts and Catalog UI Policies

now.

Good Practices

- Ensure Catalog variables have names so they can be referenced in a script
- Use a Catalog UI Policy instead of a Catalog Client Script if you can for faster load times and easier maintenance
- Use the Client-side debugging techniques you learned in the previous modules to debug your Catalog Client Scripts and Catalog UI Policies
- **Comment your scripts!!**

Module Recap: UI Policies

now.

Core Concepts

Catalog Client Scripts and Catalog UI Policies manage the behavior Catalog Items

Execute in the browser

Variables in a condition must be visible (even if it is hidden or read-only) on the form layout for the condition to be tested

Catalog Client Scripts and Catalog UI Policies can also be applied to Variable Sets

Behavior of variables can be managed at the Variable Set level or the Catalog Item level

Real World Cases

• **Why** would you use these capabilities?

• **When** would you use these capabilities?

• **How often** would you use these capabilities?

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Module 5: Business Rules

now.

| |
|-----------------------|
| Scripting Overview |
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Define what a Business Rule is
- Know when to write Business Rules
- Write, test, and debug Business Rules
- Use dot-walking to access data on related tables

Labs

- 5.1 Debugging Business Rules
- 5.2 Current and Previous
- 5.3 Display Business Rules and Dot-walking

In this module you will write, test, and debug Business Rules.

Business Rules Overview

When Business Rules Run

What Business Rules Execute

Server-side Global Variables

Server-side Debugging

Overview: What is a Business Rule?

now.

- JavaScript that runs when
 - A **record** is **displayed, inserted, updated, or deleted**
 - A **table** is **queried**
- Can be used to change values in fields when certain conditions are met
- Execute server-side
 - Faster
 - Does not monitor forms or lists



Business Rules respond to all record accesses regardless of access method: forms, lists, or Web Services.

FAQ's

What Exists Baseline?

- 2,000+ Business Rules exist baseline

When do they execute?

- Run whenever records are accessed

Where can they be found?

- Navigate to the **System Definition > Business Rules** module to create or modify Business Rules

Business Rules Overview

When Business Rules Run

What Business Rules Execute

Server-side Global Variables

Server-side Debugging

When Business Rules Run: Business Rule Execution

now.

- Watches the database for record access and responds to select access types
- Trigger specifies **when to execute**
- Not all Business Rules require scripting

The screenshot shows the 'Business Rule' configuration page for a rule named 'problem_reopen'. The 'Table' is set to 'Problem [problem]'. The 'When to run' tab is selected, showing 'before' as the trigger type and an 'Order' of 101. The 'Actions' tab is visible on the right, with checkboxes for Insert (unchecked), Update (checked), Delete (unchecked), and Query (unchecked). The 'Advanced' tab is also visible. Below the tabs, there are 'Filter Conditions' and 'Role conditions' sections. The 'Filter Conditions' section contains two rules: 'Problem state' is less than 'Closed/Resolved' AND 'Active' is false. The 'Role conditions' section is empty. At the bottom are 'Update' and 'Delete' buttons.

Navigate to **System Definition > Business Rules** and select the **New** button to create a new Business Rule.

Configure the Business Rule trigger:

Name – name of the Business Rule.

Application – identifies the scope of the Business Rule. Important to correctly identify the scope to ensure the Business Rule's logic does not inadvertently impact other areas of ServiceNow.

Table – specifies the database table containing the records of interest. You can create Business Rules on tables in the same scope as well as on tables permitting access from another scope.

Active – if selected the Business Rule is executing in the runtime environment.

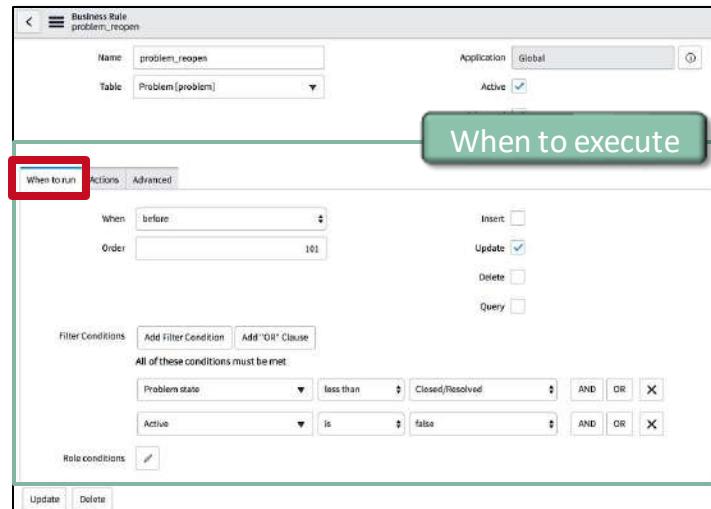
Advanced – select this option to turn on the Advanced tab for scripting.

Although not included by default, it is recommended you add the **Description** field to the Business Rule form to document the Business Rule.

When Business Rules Run: Business Rule Trigger – When to Run

now.

- Set the criteria for execution of the Business Rule logic
- Responds to selected access types
- Use **Filter Conditions** and **Role conditions** to configure simple conditions



Configure when the Business Rule will execute:

Insert – select this check box to execute the Business Rule when a record is inserted into the database.

Update – select this check box to execute the Business Rule when an existing record is updated.

Filter Conditions – must return true for the Business Rule logic to execute.

Role conditions – select the roles that users who are modifying records in the table must have for this Business Rule to execute.

When the **Advanced** check box is selected, additional configuration options are available on the **When to run** tab:

When – timing of Business Rule script execution relative to the record access.

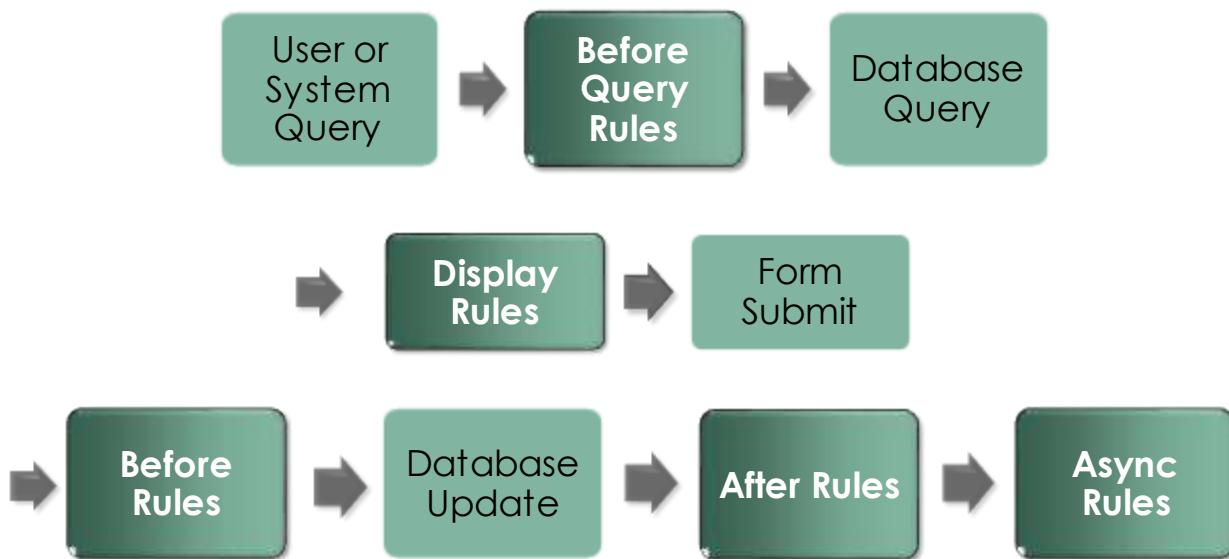
Order – order of execution; executed from low to high.

Delete – select this check box to execute the Business Rule when a record is deleted from the database.

Query – select this check box to execute the Business Rule when a table is queried. **Business Rules defined for a database view can only run on Query. A Business Rule for a database view cannot run on insert, update, or delete.**

When Business Rules Run: When Business Rules Execute

now.

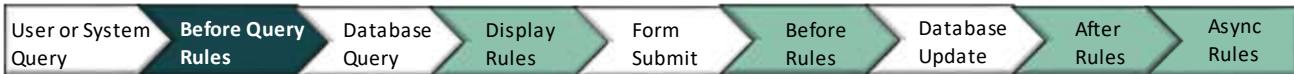


Business Rules run at different **times** relative to database access.

When Query is part of the trigger condition, the Business Rule runs before the database query.

When Business Rules Run: Before Query

now.



- Execute *before* a query is sent to the database
- *Example: Prevent users from one location seeing CIs from another*

Before Query
Business Rule
queries the
database



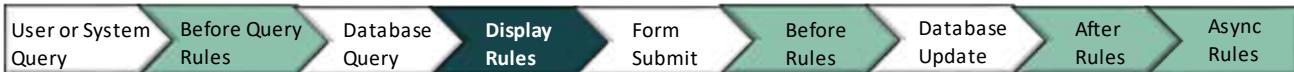
| number | caller_id |
|-----------|---|
| INC000002 | 5137153cc611227 c-000b-0bd1bd8cd2 005 |
| INC000003 | 681cccaf9c0a8016 400b98a06818d57 c7 |

Before Query Business Rules execute synchronously. The current Business Rule must finish execution before the next Business Rule runs.

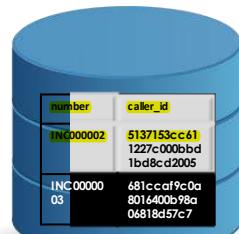
When a user is not authorized to see all records in a list, the "Number of rows removed by Security constraints" message appears. *Before Query* Business Rules such as the baseline 'incident query' Business Rule act like Access Control Lists (ACLs) and prevents users from seeing certain records. When access is controlled through a *Before Query* Business Rule, the "Number of rows removed by security constraints" message is not displayed and the user does not know access is restricted.

When Business Rules Run: Display

now.



- Execute *after* the data is read from the database and *before* the form is presented to the user
- Primary purpose is to populate the `g_scratchpad` global object
- Example: Provide Client Scripts with access to data from other records



Display
Business Rule executes
and populates the
`g_scratchpad` object

Display Business Rules execute when a user requests a record form. Data is read from the database, the Display rules are executed, and the form is presented to the user.

The `g_scratchpad` global object has no property/value pairs unless populated by a *Display Business Rule*. For example, the `sys_created_by` and `sys_created_on` fields are not part of the Incident form. To access their values from a Client Script would require a call to the server. Server calls are expensive from a processing time perspective and should be avoided. Instead, the *Display Business Rule* populates the `g_scratchpad` object when the form loads and the Client Script reads the values from the `g_scratchpad` object.

Display Business Rule

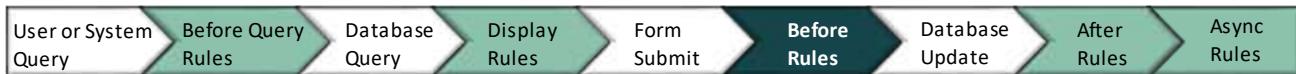
```
g_scratchpad.createdBy = current.sys_created_by;  
g_scratchpad.caller = current.caller_id;  
g_scratchpad.callerDV = current.caller_id.getDisplayValue();
```

Client Script

```
if(g_scratchpad.createdBy == 'admin') {  
    g_form.setValue('opened_by', g_scratchpad.caller,  
    g_scratchpad.callerDV);  
}
```

When Business Rules Run: Before

now.



- Execute *after form submission and before record is updated in the database*
- Example: Populate 'closed_by' with the name of the currently logged in user*

A screenshot of the ServiceNow Incident detail page. The page shows various fields for an incident, including:

- Number: INC0000039
- Caller: Bud Richman
- Category: Network
- Subcategory: None
- Business service: MailServerUS
- Configuration item: MailServerUS
- Contact type: Phone
- State: New
- Impact: 3 - Low
- Urgency: 3 - Low
- Priority: 5 - Planning
- Assignment group: Network
- Assigned to: (empty field)

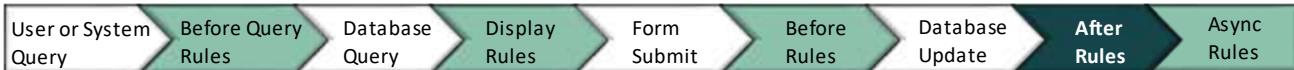
Before
Business Rule
executes



Before Business Rules execute synchronously. The current Business Rule must finish execution before the next Business Rule runs.

When Business Rules Run: After

now.



- Execute *after* form submission and *after* the record update in the database
- Example: Cascade changes made to the approval field of a Service Catalog request to the requested items attached to that request

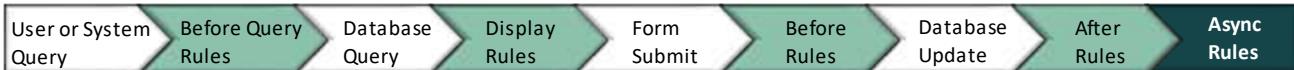


After
Business
Rule
Executes

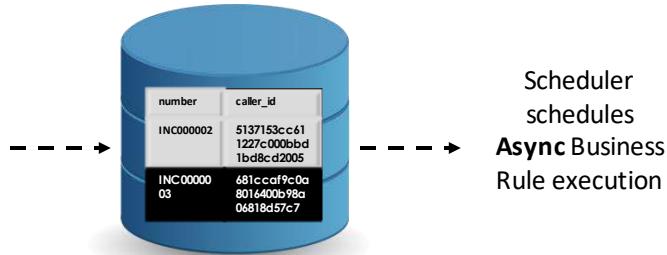
After Business Rules execute synchronously. The current Business Rule must finish execution before the next Business Rule runs.

When Business Rules Run: Async

now.



- Execute *after* records are inserted/modified/queried
- Run asynchronously as Scheduled Jobs
- Examples:
 - Notify subscribers when Cls are affected by an Incident
 - SLA calculations



Async Business Rules are queued by the scheduler to run as soon as they can be fit in, this allows the current transaction to finish without waiting for the rule. They do not have access to the previous version of a record.

The **Priority** field is visible on the Business Rule form when **Async** is selected in the *When* field. The value in this field is used when creating the associated scheduled job. By convention, the Priority numbers in an Async Business Rules contain three digits. For example: 100, 300, or 675.

To see Async Business Rules queued up for execution, open **System Scheduler > Scheduled Jobs > Scheduled Jobs**. Async Business Rules appear with their names prepended with ASYNC. They go in and out very quickly so they can be hard to catch.



TIP FROM FIELD

Use Async in place of After when immediate response is not required. The user will not have to wait on the Business Rule to finish before they can continue to work.

Business Rules Overview

When Business Rules Run

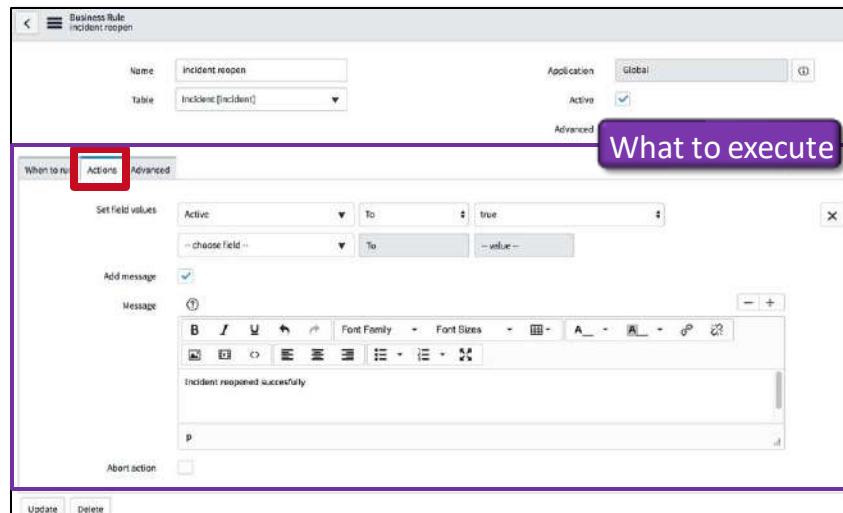
What Business Rules Execute

Server-side Global Variables

Server-side Debugging

What Business Rules Execute: Business Rule Trigger – Actions now

- Configure **Actions** to specify **what to execute**
- Scripting is not required to
 - Set a field value
 - Add a message



Configure what the Business Rule will do:

Set field values – set values for fields in the selected table using the choice lists.

Add message – select this checkbox to identify if a message will be displayed to the user on the next rendered screen when the Business Rule executes.

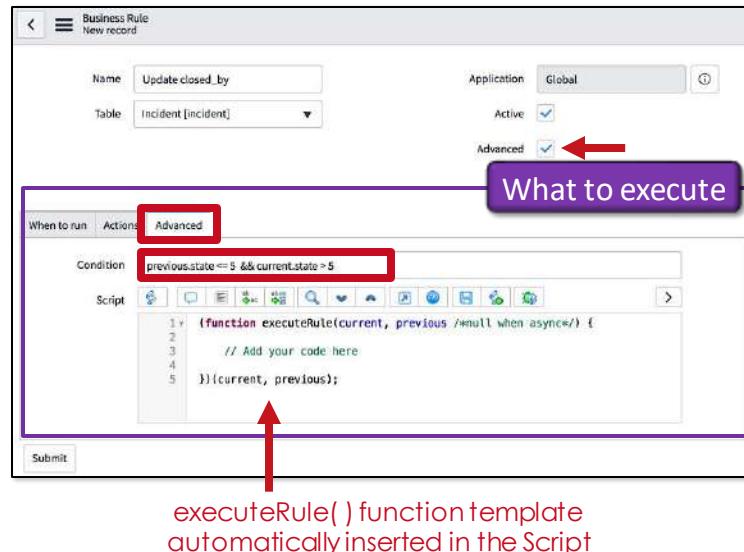
Message – compose the message to be sent.

Abort action – select this check box to abort the current database transaction. **If you select this option, you cannot perform additional actions on the record.** You can still display a message to users by selecting the *Add message* checkbox and composing the message.

What Business Rules Execute: Business Rule Trigger – Advanced

now.

- More complex conditions and actions can be scripted
- Select the **Advanced** tab to access the scripting fields
- Script a **Condition** to avoid loading unnecessary script logic



Use the scripting fields on the **Advanced** tab when the Business Rule's conditions and actions are more complex. Example: evaluating the *previous* value of a field.

Condition – create a JavaScript statement to specify when the Business Rule should execute. The Condition field returns *true* when left blank.

Script – JavaScript that executes server-side when the **When to run** criteria is met and the **Condition** field returns *true*. Insert the appropriate script logic to be processed when the Business Rule executes.



IMPORTANT

Condition(s) on the *When to run* tab and condition(s) in the Condition field on the *Advanced* tab must both evaluate to true for the logic in the Script field to execute.

Business Rules: Topics

now.

Business Rules Overview

When Business Rules Run

What Business Rules Execute

Server-side Global Variables

Server-side Debugging

Server-side Global Variables: What Data Can You See in a Business Rule?

now.

- Local variables declared in a script
- Predefined server-side Global variables
 - **current**
 - **previous**
 - **g_scratchpad**
 - Display Business Rules only
 - Works in conjunction with a client-side script



```
if(current.state >= 5 && previous.state < 5) {  
    var reOpened = true;  
    g_scratchpad.createdBy = current.sys_created_by;  
    g_scratchpad.caller = current.caller_id;  
}
```

current is an object that stores the current record's fields and values; new/modified values. A script can change a field's value many times current will store the most recent value. For example:

```
if(previous.priority == current.priority {  
    return;  
}
```

previous is an object that stores the record's fields and values before any changes were made; the original values when the form loaded. Reference this object using previous.<field_name>. For example:

```
if(previous.priority == 1){  
    //logic here  
}
```



IMPORTANT

The **previous** object is not available in a Async Business Rules.

Global Variables: Dot-Walking

now.

- Allows direct access to fields for related records (reference objects) by traversing tables

<object>.<reference object name>.<field of interest> Syntax

current.caller_id.department.name Example

Incident [incident]

| number | caller_id |
|------------|--------------------------------------|
| INC0000040 | 46c6f9efa9fe198101ddf5eed9 adf6e7 |

Department [cmn_department]

| sys_id | name |
|--------------------------------------|-------|
| 221db0edc611228401760aec06c 9d929 | Sales |

User [sys_user]

| user_name | sys_id | department |
|-------------|--------------------------------------|--------------------------------------|
| bud.richman | 46c6f9efa9fe198101ddf5eed9 adf6e7 | 221db0edc611228401760aec06c 9d929 |

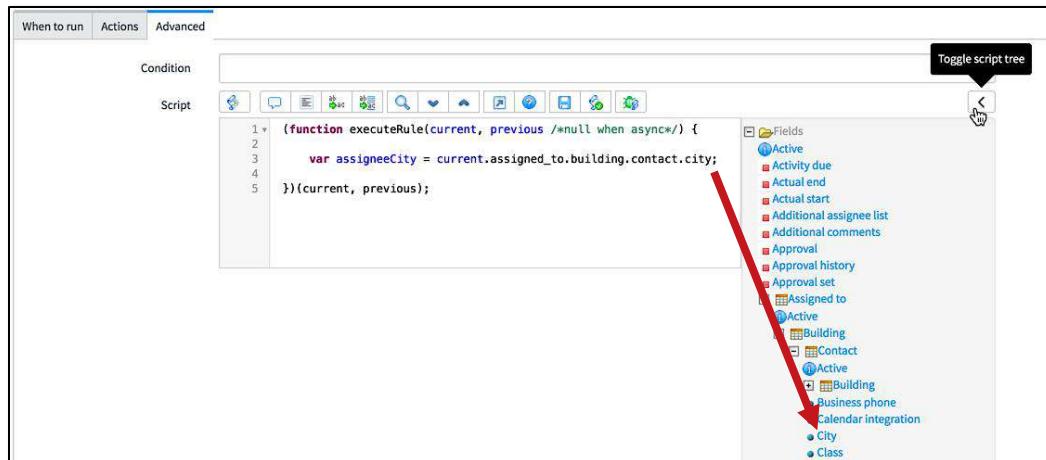
Client-side you must use the `getReference()` method to access data from related record. Recall this method requires a call to the server from the client and it only allows one-level of dot-walking on the retrieved object. Business Rules run on the server and have direct access to the database. To hop from table to table you use a strategy called 'dot-walking'.

Dot-walking is for a one-to-one relationships and not for one-to-many (e.g. – related lists).

In this example, the current object is an Incident [incident] table record. The statement `current.caller_id.department.name` jumps from the Incident [incident] table record to the related record in the User [sys_user] table where the sys_id matches the caller_id from the Incident table. The department id is then used to jump to the matching record on the Department [cmn_department] table. The caller's department name is then retrieved.

Global Variables: Automatically Insert the Dot-Walking Path now

- The **Script Tree** makes dot-walking easy
- Syntax is automatically inserted when the field of interest is selected



Expand the **Fields** category to see the list of variables in the current object.

Informational icons indicate the source and type of each field.

- A table icon indicates the field references another table. Expand the Reference Object to select the field of interest from that table.
- A blue circle indicates the field resides on the current table.
- A red square indicates the field resides on a parent table.
- An Index field icon indicates the field is part of one or more indexes.

Business Rules: Topics

now.

Business Rules Overview

When Business Rules Run

What Business Rules Execute

Server-side Global Variables

Server-side Debugging

Debugging: Debugging Business Rules

now.

- ServiceNow server-side debugging tools
 - GlideSystem logging methods
 - Debug Business Rules module
 - Script Debugger
- JavaScript debugging tool
 - try/catch



Debugging: GlideSystem Logging Methods

now.

- Send debugging information to the system log

| | |
|------------|---|
| gs.log() | This will always log a message when used in the global scope. |
| gs.error() | Used for critical errors that are encountered; the issue will be logged. |
| gs.warn() | Used for logging occurs when there might be an issue. |
| gs.info() | Used for informational events, such as messages that describe the progress of the application |
| gs.debug() | Logging occurs when in debug mode or when session debug for that application is enabled. |

- Send debugging information to the top of the form

- gs.addInfoMessage()
 - gs.addErrorMessage()
 - gs.methodName(<message>);

```
gs.log ("The record was opened: " + current.opened_at);
```

← Syntax

← Example

Use the **System Logs > System Log** modules to view messages logged by the gs.log(), gs.error(), gs.info(), gs.warn(), and gs.debug() methods. gs.log() will always write a message, so overuse of this can adversely affect performance. Whenever possible, comment out or remove gs.log().

The gs.addInfoMessage() and gs.addErrorMessage() methods are helpful debugging strategies during class, because the feedback is immediate. This strategy can be less effective in an environment with multiple administrators as they will see your messages at the top of the form.



TIP FROM THE FIELD

The default source for the gs.log() methods is ***Script. To use a different source, pass an optional source parameter to the method. Then save a query locating all the records in your source as a favorite. Example:

```
gs.log ("The value of State: " + current.state, "Fred's  
Logs");
```



IMPORTANT

Keep in mind, If you are developing in a scoped application, you will not be able to use gs.log(). Error, warn, info, and debug logging methods are intended for Scoped Applications, but can be used in the Global Scope. These logging methods provide verbosity levels to a log message.

Debugging: Debug Business Rules Module

now.

- Select **System Diagnostics > Session Debug > Debug Business Rule**
- Writes logging information to the bottom of forms and lists
 - *Can test and debug on the same form*
- Session specific – no other users impacted
- Debug condition evaluation

```
⌚ 15:17:29.911: Global 🌐 ==> 'Logging Methods' on incident:INC0000044
⌚ 15:17:29.914: ; This message is from gs.log()
⌚ 15:17:29.917: ; This message is from gs.error(): no thrown error
⌚ 15:17:29.919: ; This message is from gs.info()
⌚ 15:17:29.919: ; This message is from gs.warn()
⌚ 15:17:29.921: ; [DEBUG] This message is from gs.debug()
⌚ 15:17:29.921: Global 🌐 <== 'Logging Methods' on incident:INC0000044
⌚ 15:17:29.921: Global === Skipping 'mark_closed' on incident:INC0000044; condition not satisfied: Condition: current.incident_state == IncidentState.CLOSED || current.incident_state == IncidentState.CANCELED
```

To locate a specific Business Rule in the Debug Output look for the string ==> '<Business Rule Name>' (*Ctrl-F/Cmd-F*). All debugging information following that string until the string <== '<Business Rule Name>' pertains to the Business Rule.

In the example shown the Business Rule name is 'Logging Methods'.

Select **System Security > Debugging > Stop Debugging** to stop writing debugging output to forms.



IMPORTANT

Remember to disable the debugger when you are done testing, because debugger consumes resource when it is enabled.

Debugging: Debug Business Rules (Details)

now.

- Select **System Diagnostics > Session Debug > Debug Business Rule (Details)**
- Field values set in Business Rules appear in the detailed debugging information

| |
|---|
| ⌚ 15:23:10.141: Global ==> 'mark_closed' on incident:INC0000047 |
| ⌚ 15:23:10.185: Global active: 1 => 0 |
| ⌚ 15:23:10.185: Global business_duration: => 1970-01-30 04:23:10 |
| ⌚ 15:23:10.185: Global business_stc: => 2521390 |
| ⌚ 15:23:10.185: Global calendar_duration: => 1970-05-03 00:29:52 |
| ⌚ 15:23:10.185: Global calendar_stc: => 10542592 |
| ⌚ 15:23:10.185: Global closed_at: => 2017-02-01 21:23:10 |
| ⌚ 15:23:10.185: Global closed_by: => 6816f79cc0a8016401c5a33be04be441 |
| ⌚ 15:23:10.186: Global ==> 'mark_closed' on incident:INC0000047 |

When detailed debugging is enabled, record field values set in Business Rules appear in the debugging output. The syntax is:

```
<field name>: <old value> => <new value>
```

In the example shown, the active field's value changed from 1 to 0.

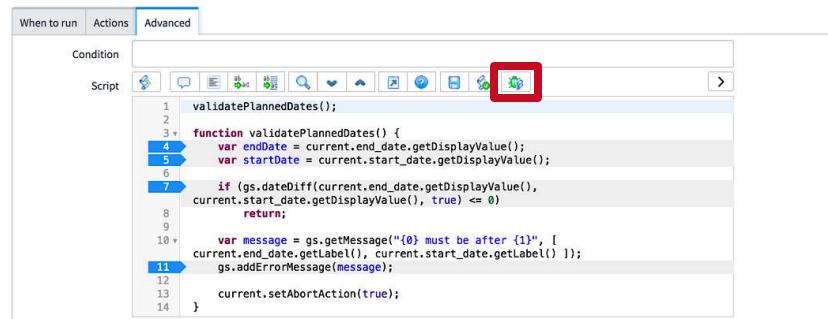
This strategy allows you to see changed values for all fields changed by Business Rules.

Select **System Security > Debugging > Stop Debugging** to stop writing debugging output to forms, as the Debugger can cause performance issues in a Production instance.

Debugging: Script Debugger

now.

- For debugging **server-side** JavaScript
- Launches by selecting
 - **Script Debugger** icon on the Syntax Editor toolbar
 - **System Diagnostics > Script Debugger** on the Application Navigator
- Opens in a new browser window



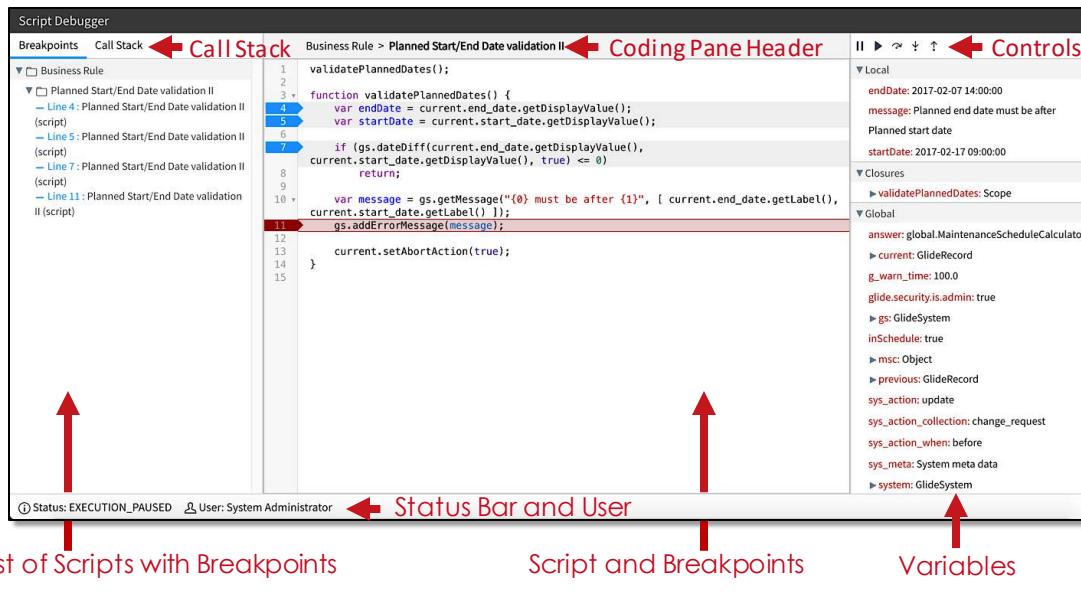
The Script Debugger offers:

- The ability to step through your script line-by-line.
- Set and remove breakpoints.
- Pause a script at a breakpoint.
- Step into and out of function and method calls.
- View the value of local and global variables in real time.
- View the value of private variables from function closures.
- View the call stack.
- View the transaction the system is processing.

Requires the *script_debugger* or the *admin* or role.

The Script Debugger can also be launch in the Studio by selecting **File > Script Debugger**.

Debugging: Anatomy of the Script Debugger



The Script Debugger is session specific; no other users are impacted. Each user can only have one Script Debugger console open and executing at a time.

Administrators can use the Script Debugger while impersonating another user, but only if the impersonated user has the *admin* or *script_debugger* role and has read access to the target script.

While impersonating another user you can:

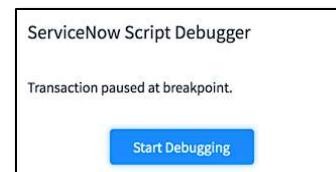
- See and change breakpoints that belong to the impersonated user.
- View and pause on scripts that the impersonated user has read access to.
- The Script Debugger step-through controls also use the read access of the impersonated user. *For example, if the impersonated user does not have read access to a function in the call stack, any Step into action instead becomes a Step over action.*
- The impersonated debugging session lasts until:
 - You stop impersonating the user.
 - You log out or the user session ends.
 - You pause the Script Debugger.
 - You close the Script Debugger.

Debugging: To Debug Scripts

now.

- Script Debugger must be open
- Script must have a breakpoint set
- Script must be executing in the same interactive session as the user with the breakpoints
- Script debugging is session and user specific
 - Only you are affected by your breakpoints
 - Cannot step through scripts running asynchronously
 - Asynchronous Business Rules
 - Scheduled Jobs
 - Workflow scripts
 - Inbound email actions

Even if Fred has a breakpoint at a point in code, Christen will never experience it



Notification advises
debugging
triggered

The Script Debugger only debugs server-side synchronous scripts. It is not possible to step through scripts running asynchronously as they are not running in the the same session as the logged-in user.

A debugger transaction remains open as long as the user session is valid. If the user logs out, or the session times out (which will occur if paused for too long), the system stops the debugger transaction. If this occurs, it can be re-enabled by selecting the **on** button, pressing **F2**, or by closing and re-opening the debugger.



TIP FROM THE FIELD

Breakpoints do not need to be removed from a script. If you would like to debug at another time with the same breakpoints, simply close the Script Debugger window to turn off debugging. The script will execute without pausing at breakpoints until you open the Script Debugger again. This strategy makes it easy for you to control when you want to debug your scripts.

Debugging: Script Debugger - Breakpoints

now.

- Pause execution of a script
- Select the gutter to the left of a line to set a breakpoint
 - Can be set in a Script field or in the Script Debugger

Script Debugger

Breakpoints Call Stack

Business Rule > Multiply Two Numbers

```
1 * (function executeRule(current, previous /*null when async*/){  
2  
3     var integer1 = 32;  
4     var integer2 = current.reopen_count;  
5  
6     gs.addInfoMsg("This is an Informational Message"); ↑  
7  
8     //Call the Script Include  
9     MultiplyTwoNums(integer1,integer2); ↑  
10    })(current, previous);  
11  
12  
13
```

Current line highlighted when paused

- Every breakpoint the logged-in user has defined (*across multiple scopes*) appears in the list

Breakpoints are an intentional stopping or pausing place in a script, for debugging purposes.

The Script Debugger lists all of the logged-in user's Breakpoints across the platform (regardless of scope). The artifacts are listed by script type, script name, and line number. The Script Debugger updates this list as you add and remove breakpoints in real time.

In the coding pane, current breakpoint is highlighted in red and inactive breakpoints are highlighted in blue.

Select a breakpoint to remove it.

Debugging: Script Debugger - Call Stack and Transaction Detail

now.

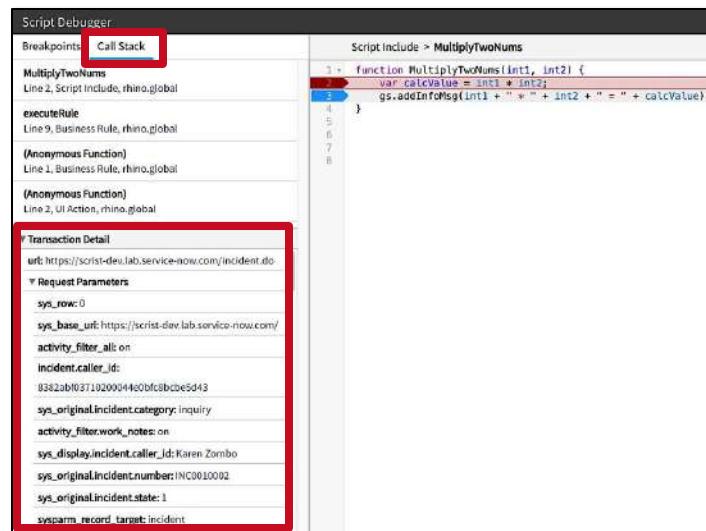
- Call Stack

List of script calls that either

- Call the current script
- The current script calls

- Transaction Detail

- Details for the current paused user session
- Visible only when the debugger pauses on a breakpoint



Transaction details can be used to:

- Inspect the URL of the currently paused transaction.
- Inspect request parameters for the currently paused transaction.
- Inspect network information about the current transaction.
- Inspect the user and session ID that initiated the debug transaction.
- Parameters of the HTTP request (if they exist).

See ServiceNow's Product Documentation for the full list of **Available Transaction Details** and their descriptions.

Debugging: Script Debugger - Coding Pane

- Displays the script currently executing
- Select another script to view using the list of scripts with Breakpoints

The screenshot shows the Script Debugger interface with the following details:

- Header:** Business Rule > Planned Start/End Date validation II
- Breakpoints:** A list of breakpoints for the current script, including:
 - Line 4: Planned Start/End Date validation II (script)
 - Line 5: Planned Start/End Date validation II (script)
 - Line 7: Planned Start/End Date validation II (script)
 - Line 11: Planned Start/End Date validation II (script)
- Call Stack:** Not visible in the screenshot.
- Script Content:**

```

1 validatePlannedDates();
2
3 v function validatePlannedDates() {
4     var endDate = current.end_date.getDisplayValue();
5     var startDate = current.start_date.getDisplayValue();
6
7     if (gs.dateDiff(current.end_date.getLabel(),
8         current.start_date.getLabel(), true) <= 0)
9         return;
10    var message = gs.getMessage("{0} must be after {1}", [current.end_date.getLabel(),
11    current.start_date.getLabel()]);
12    gs.addErrorMessage(message);
13
14    current.setAbortAction(true);
15

```
- Local Variables:**
 - endDate: 2017-02-07 14:00:00
 - message: Planned end date must be after
 - Planned start date
 - startDate: 2017-02-17 09:00:00
- Closures:**
 - validatePlannedDates: Scope
- Global Variables:**
 - answer: global.MaintenanceScheduleCalculator
 - current: GlideRecord
 - g_warn_time: 100.0
 - glide.security.is.admin: true
 - system: GlideSystem

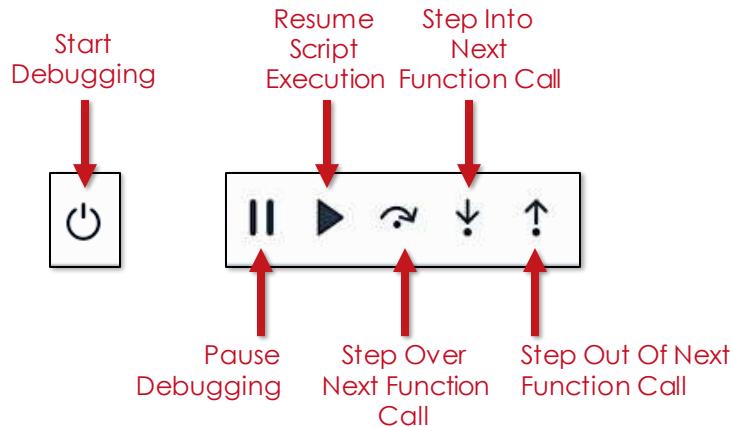
The **Coding Pane Header** displays the script type and name of the script currently in the coding pane.

Scripts are read-only when viewed in the Coding Pane.

Debugging: Script Debugger - Controls

now.

- Available when the script pauses at a breakpoint
- Move between script lines and scripts in the Call Stack



Start Debugging (F2) – enables the Script Debugger for the current user. If the Script Debugger is already enabled, the Pause Debugging icon will display instead.

Pause Debugging (F2) – stops current debugging session, and disables the Script Debugger for the current user. The Script Debugger does not pause on breakpoints for the current user until it is restarted. If the Script Debugger is already paused, the Start Debugging icon will display instead.

Resume Script Execution (F9) – advances from the current breakpoint to the next breakpoint. If no breakpoints exist, the script runs to completion.

Step Over Next Function Call (F8) – advances to the next evaluated line of script based on current conditions. The Script Debugger skips any lines of code that do not run because their conditions are not met. *For example, when the condition of an if() statement is not true, the Script Debugger skips the code block for the condition.*

Step Into Next Function Call (F7) – when the Script Debugger pauses ON a method call, this control allows the user to advance to the first line of executed code in the called method. Stepping into a method also updates the current position within the Call Stack. If the user does not have read access to the method call, this control instead behaves like *Step Over*.

Step Out of Current Function (Shift + F8) – when the Script Debugger pauses WITHIN a method call, this control allows the user to exit the called method and return to the calling script from the Call Stack. If the user is not within a method call, this control instead behaves like *Step Over*.

Debugging: Script Debugger - Variables

now.

- When stopped at a breakpoint, examine variables and their values in real time
 - Local
 - Closures
 - Global



Variable information is only visible when the Script Debugger pauses on a Breakpoint.

Local – displays a list of local scope variable names and their values. Includes both the variables you declared inside the script, as well as declared baseline.

Closures – displays a list of global scope variable names and their values set by function closure.

Global – displays a list of global scope variable names and their values.

Debugging: Script Debugger - Status Bar

now.

- Provides the name of the user running the current Script Debugger session
- Displays the current debugging action
 - EXECUTION_PAUSED
 - WAITING_FOR_BREAKPOINT
 - WAITING_FOR_FIRST_BREAKPOINT
 - OFF

 Status: EXECUTION_PAUSED  User: System Administrator

EXECUTION_PAUSED – displays when the Script Debugger pauses on a breakpoint, or when the user steps over, steps into, or steps out to the next line of code.

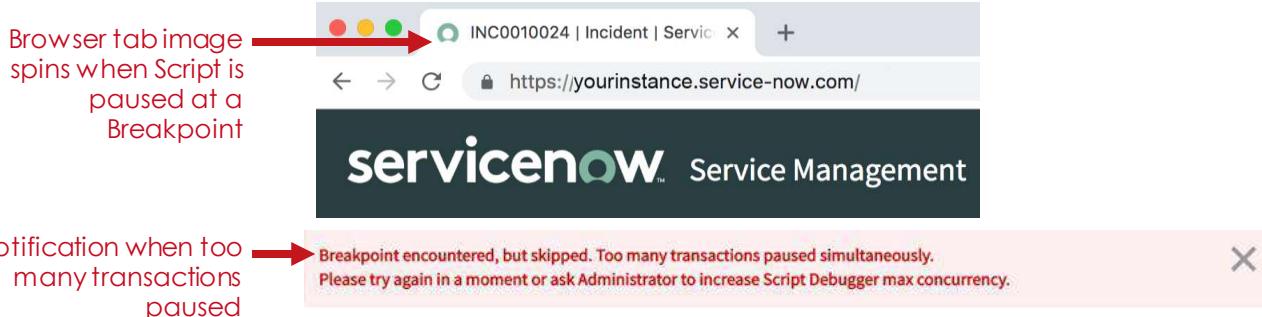
WAITING_FOR_BREAKPOINT – displays when the Script Debugger is searching for the next breakpoint. Users rarely see this status as it changes quickly back to EXECUTION_PAUSED as soon as the next breakpoint pauses the script.

WAITING_FOR_FIRST_BREAKPOINT – displays when the user first opens the Script Debugger and it is ready to pause the script and display debugging information.

OFF – displays when the Script Debugger is inactive and does not pause scripts or display debugging information. The Script Debugger turns off when a user pauses or closes the Script Debugger, the user session times-out or ends for any reason, or the Administrator resets all Script Debugger instances by navigating to the *debugger_reset.do* page.

Debugging: Script Debugger - Multiple Developer Support now.

- Debugging causes transactions to pause for significant periods
- The Script Debugger has its own semaphore
- Controls/coordinates debugging when multiple developers compete for the same platform resources



By default, the platform supports debugging $\lceil \# \text{ of semaphores on the instance} / 4 \rceil$ concurrent transactions. For example, if you have 12 semaphores by default 3 transactions can be debugged at a given point in time.

Administrators can specify the number of concurrent transactions the system can debug by setting the **glide.debugger.config.max_node_concurrency** system property. The system can debug up to $\lceil \# \text{ of semaphores on the instance} - 2 \rceil$ concurrent transactions.

To see the platform's Script Debugging semaphore set, type `stats.do` in the Application Navigator followed by `<enter>` on the keyboard. Example:

Semaphore Sets

Debug

Available semaphores: 4

Queue depth: 0

Max queue depth: 0

Maximum transaction concurrency: 4

Maximum concurrency achieved: 0

Module Labs

now.

- **Lab 5.1**

- **Time:** 25-30m
- Debugging Business Rules
 - Practice debugging Business Rules using several different strategies

- **Lab 5.2**

- **Time:** 15-20m
- Current and Previous
 - Practice using **current** and **previous** global objects
 - Use Business Rule with form fields.
 - More debugging

- **Lab 5.3**

- **Time:** 15-20m
- Display Business Rules and Dot-walking
 - Use `g_scratchpad` global object, Display Business Rules, and dot-walking



Debugging Business Rules

Lab
05.01

⌚25-30m

Lab Summary

You will achieve the following:

- Debug Business Rules using several different strategies.

Note: Should the Script Debugger lose connection to the script you are debugging at any time, simply close and re-open the window.

A. Preparation

1. Navigate to **System Definition > Business Rules**.
2. Locate and open the **Lab 5.1 Business Rule Debugging** Business Rule.
3. Select the Active checkbox to make the Business Rule active.
4. On the Advanced Tab, overwrite the string `<your_initials>` with **your personal initials** in the Script. Example:

```
catch(err){  
    gs.log("JS!!!: a JavaScript runtime error occurred - " + err);  
}
```

5. **Save** the record, remain on the form.
6. Review the Business Rule and read the script so you understand:
 - **when** it triggers
 - **what** it executes
7. Consider saving this Business Rule as a Favorite as you will be opening the record often throughout the lab. Select **Create Favorite** on the form's Context menu.

B. Practice Using the Script Debugger – Breakpoints and Variables

1. In the *Script* field, set breakpoints by clicking the gutter to the left of the lines beginning with:

- **var myNum**
- **var priorityValue**
- **var createdValue**

```
1 current.short_description = "This text set by the Lab 5.1 Business Rule Debugging BR";
2 var myNum = current.state;
3
4 //Advise logged in user when Incident was created
5 var priorityValue = current.priority;
6 var createdValue = current.sys_created_on;
7
8 SlaTargetNotification(priorityValue,createdValue);
```

2. Select the **Open Script Debugger** () icon on the Syntax Editor toolbar or the **System Diagnostics > Script Debugger** module on the Application Navigator. *The Script Debugger opens in another browser window.*
3. Notice the list of all breakpoints set in the platform by you appears on the left-side of the code pane.
4. Set another breakpoint by clicking the gutter to the left of the line beginning with **SlaTargetNotification**.

Business Rule > **Lab 5.1 Business Rule Debugging**

```
1 current.short_description = "This text set by the Lab 5.1 Business
2 Rule Debugging BR";
3 var myNum = current.state;
4 //Advise logged in user when Incident was created
5 var priorityValue = current.priority;
6 var createdValue = current.sys_created_on;
7
8 SlaTargetNotification(priorityValue,createdValue);
```



5. What can you conclude about where breakpoints can be set? Record your answer here:

6. **Keep the Script Debugger window open.** Force the *Lab 5.1 Business Rule Debugging* Business Rule to execute.
 - a) Open any active Incident, change the value of **State** to anything except Closed.
 - b) **Save** the record, remain on the form.
7. Select **Start Debugging**.

Note: Had the Script Debugger window not been open, the script would have executed normally without stopping at any breakpoints.



8. Script execution is paused at the first breakpoint. How can you tell? Explain your reasoning:

9. Examine the value of the **myNum** variable in the *Local* variables section on the right-side of the window. Does it contain the value you expected? _____



10. Select the **Next Breakpoint** button (▶) to resume script execution and pause at the next breakpoint.
11. Re-examine the value of the **myNum** variable in the *Local* variables section. What value does it contain now? _____

12. Explain why the **myNum** variable value is sometimes *undefined* and sometimes has a value:
-
-

13. Repeat steps 9-11 to watch the values of the **priorityValue** and **createdValue** variables update as you step through the code.

Note: Specifically notice the value of **createdValue**, this type of output is extremely valuable when you are scripting dates. E.g. you may write a script similar to:

```
if(current.sys_created_on <= 'some value')...
```

and it does not work. This output will quickly confirm the syntax/value in the [sys_created_on] field and you can then adjust the value of 'somevalue' accordingly.

14. When you reach the breakpoint on the line beginning with *SlaTargetNotification*, select the **Next Breakpoint** button one last time to complete the execution of the script.
15. Think of an example in the past where you declared variables in a script and they did not return the values you were expecting. Would it have been helpful to have variable value output available like this?
-

C. Practice Using the Script Debugger – Current and Previous Values

1. **Keep the Script Debugger window open.** Open any Incident with a *Short description* value other than "This text set by the Lab 5.1 Business Rule Debugging BR".
2. Force the *Lab 5.1 Business Rule Debugging* Business Rule to execute:
 - a) Change the **State** value to anything except Closed.
 - b) **Save** the record, remain on the form.
3. Select **Start Debugging**.

4. Document the **previous** object's *short_description* field value.
 - a) Expand the **previous: GlideRecord** object in the *Local* variables list.



- b) Record the previous value of the *short_description* field here:
-

5. Document the **current** object's *short_description* field value.
 - a) Expand the **current: GlideRecord** object in the *Local* variables list.



- b) Record the current value of the **short_description** field here:
-

6. Why are they different? Explain your reasoning:
-
-

7. Select the **Next Breakpoint** button as many times as needed to complete the execution of the script.
8. Close the Script Debugger window.

D. Practice Using the Script Debugger – Call Stack

In this section, you will practice using the Script Debugger when one server-side script calls another server-side script. This feature can help you pinpoint exactly which script needs fixing when multiple scripts are executing.

1. Navigate to **System Definition > Script Includes**.

Note: You will learn about *Script Includes* in an upcoming module, for now you are only adding a breakpoint to the script.

2. Locate and open the **SlaTargetNotification** Script Include.
3. Set a breakpoint in this script by clicking in the gutter to the left of the line beginning with **gs.log**.

```
1 function SlaTargetNotification(priorityLevel, sysCreatedOn) {  
2     var loggedInUser = gs.getUserDisplayName();  
3  
4     gs.log(loggedInUser + ", this Priority-" + priorityLevel + " Incident has been open since " +  
5         sysCreatedOn);  
6 }
```

4. Select the **Open Script Debugger** button on the Syntax Editor toolbar. *The Script Debugger opens in another window.*
5. Notice the list of breakpoints now includes the one you just set in step-3.
6. Select any one of the *Lab 5.1 Business Rule Debugging* lines to load that script in the code pane.
7. Update the breakpoints so only the line of code beginning with **var currentValue** and **try** are set.

| Business Rule > Lab 5.1 Business Rule Debugging | |
|--|--|
| 1 | current.short_description = "This text set by the Lab 5.1 Business Rule Debugging BR"; |
| 2 | var myNum = current.state; |
| 3 | |
| 4 | //Advise logged in user when Incident was created |
| 5 | var priorityValue = current.priority; |
| 6 | var currentValue = current.sys_created_on; |
| 7 | |
| 8 | SlaTargetNotification(priorityValue,createdValue); |
| 9 | |
| 10 | // The function in this try/catch is not defined |
| 11 | try{ |
| 12 | thisFunctionDoesNotExist(); |
| 13 | } |
| 14 | |

8. **Keep the Script Debugger window open.** Force the *Lab 5.1 Business Rule Debugging* Business Rule to execute:

- a) Open any active Incident, change the **State** value to anything except Closed.
- b) **Save** (not Submit) the record, remain on the form.

9. Select **Start Debugging**.

10. You are stopped at the first breakpoint in the *Lab 5.1 Business Rule Debugging* script. Review the information displayed in the **Call Stack** and **Transaction Detail** on the left-side of the screen.

11. How can you be sure which script you are currently debugging? What does the **Code Pane Header** read? Record your answer here:

12. Notice the very next line in the script after the breakpoint you are currently paused at calls the *SlaTargetNotification* Script Include. Select the **Next Breakpoint** button one time.

13. Notice the information in the **Call Stack** now includes the Script Includes breakpoint details.

14. How can you be sure which script you are currently debugging? What does the **Code Pane Header** read? Record your answer here:

15. Select the **Next Breakpoint** button one time.

16. Notice the information in the **Call Stack** on the left-side of the screen changed.

17. What does the **Code Pane Header** read? Record your answer here:

18. Select the **Next Breakpoint** button one last time to complete the execution of the script.

19. Close the Script Debugger window.

E. Practice Debugging Using GlideSystem Logging Methods

1. Open the **Lab 5.1 Business Rule Debugging** Business Rule.
2. Select the breakpoints to remove them. (*There should be no breakpoints set in the script after this step is complete.*)
3. Notice two undefined functions are called in this script:
 - **thisFunctionDoesNotExist()** is in a try/catch.
 - **thisFunctionAlsoDoesNotExist()** is NOT in a try/catch.

```
11 // The function in this try/catch is not defined
12 try{
13     thisFunctionDoesNotExist(); ←
14 }
15 catch(err){
16     gs.log("<your_initials>!!!! a JavaScript runtime error occurred - " + err);
17 }
18
19 // This function is not defined and is not part of a try/catch
20 thisFunctionAlsoDoesNotExist(); ←
```

Predict what will occur when the Business Rule executes. Record your answer here:

4. Force the *Lab 5.1 Business Rule Debugging* Business Rule to execute:
 - a) Open any active Incident, change the **State** value to anything except Closed.
 - b) Select **Update**.
5. Open **System Logs > System Log > Script Log Statements**. Which undefined function produced an error and why?

6. Was a log message produced for the undefined **thisFunctionAlsoDoesNotExist()** function?

7. Search the log for records where **Message contains thisFunction**.



-
- Did you find the log message advising `thisFunctionAlsoDoesNotExist()` is not defined?

Note: If you are debugging using the Script Debugger, note that undefined functions not wrapped in a `try/catch` stop script execution and the remaining breakpoints are ignored. If this happens to you, check this list instead to get more information about why a function call may have failed.

F. Practice Debugging Using the Debug Business Rule Feature

- Select **System Diagnostics > Session Debug > Debug Business Rule**.
- Recall the condition for the execution of *Lab 5.1 Business Rule Debugging* Business Rule is `current.state !=7`. Update a record that will not trigger the Business Rule.
 - Open a closed Incident, make any change to the record.
 - Save** the record, remain on the form.
- Scroll to the bottom of the Incident form and search for the execution of the Lab 5.1 Business Rule Debugging Business Rule by searching for the string ==> **Lab 5.1 Business Rule Debugging**. Did your test meet the Business Rule's Condition criteria? How can you tell?:

- Test the case where the Condition field returns false. Examine the debugging output at the bottom of the Incident form for this lab.
- Disable Business Rule Debugging: **System Security > Debugging > Stop Debugging**.
- Make the *Lab 5.1 Business Rules Debugging* Business Rule **inactive**.

Lab Completion

Well done! You have successfully practiced testing scripts using five different server-side debugging techniques.

Current & Previous

Lab
05.02

⌚15-20m

Lab Summary

You will achieve the following:

- Use the **current** and **previous** global objects in a Business Rule.
- Create two new form fields for use in your Business Rule.
- Practice debugging.
- A Cloud Dimensions' Phase II requirement will be marked complete.



Business Problem

The current process to gather Root Cause Analysis (RCA) at Cloud Dimensions asks users to provide their findings and documentation in the 'Additional comments' field prefixed by the text "RCA: ". This strategy has proven to be unsuccessful as the prefix is often forgotten, and it is difficult to extract the data from records with multiple RCA updates. RCA reporting is currently performed manually, it takes hours to produce, and the results are often incomplete.

Project Requirement

Create a new 'RCA' field on the Incident form to capture root cause analysis documentation.

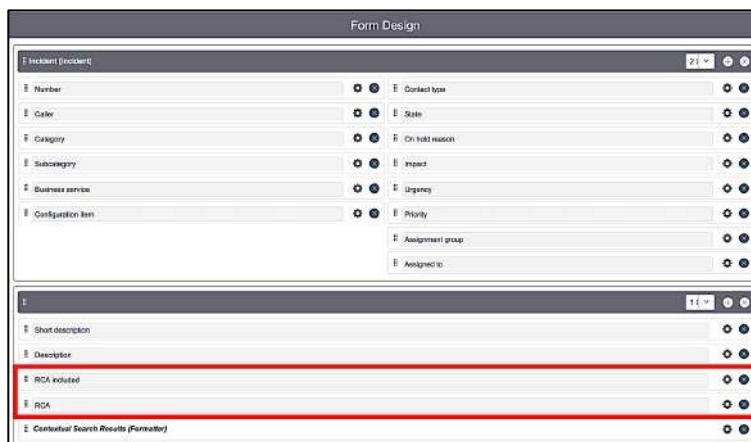
For reporting purposes, create a second field on the Incident form called 'RCA included' to easily identify records where root cause analysis documentation is included. In the event the updating analyst forgets to update this field, the system should evaluate the 'RCA' field and if populated, the 'RCA included' field should also be selected (checked). In the case where an existing value in the 'RCA' field is removed, the check should be removed from the 'RCA included' field as well.

A. Preparation

1. Create two new fields on the Incident form.
 - a) Open an Incident.
 - b) Right-click the form's Context menu and select **Configure > Form Design**.
 - c) Select the **Field Types** tab on the left side of the screen.
 - d) Drag a **True/False** field to the form below the *Description* field.
 - e) Select the **Edit this field** (gear) icon on the right side of the field.
 - f) Configure the record:

Label: **RCA included**
Name: **u_rca_included**
 - g) Close the Properties window.
 - h) Drag a **String** field to the form below the *RCA included* field.

Label: **RCA**
Name: **u_rca**
Max length: **1000**
 - i) Close the Properties window.
 - j) Select **Save**.



- k) Close the Form Design browser tab/window.
- l) Reload the Incident form to see the changes to the form.

B. Create the Business Rule

1. Create a new Business Rule.

Name: **Lab 5.2 RCA Included**
Table: **Incident [incident]**
Active: **Selected (checked)**
Advanced: **Selected (checked)**
When: **before**
Order: **125**
Insert: **Selected (checked)**
Update: **Selected (checked)**

2. On the Advanced tab, within the `executeRule()` function in the Script field, type **try** followed by the `<tab>` key to insert the **try** Syntax Editor Macro (*created in Lab 2.1*).
3. Select the **Format Code** icon on the Syntax Editor toolbar to properly align the script.
4. Update the statement in the **catch** block to use the server-side `gs.log()` method instead of the client-side `g_form.addErrorMessage()` method.

```
(function executeRule(current, previous /*null when async*/) {  
  
    try {  
  
    }  
  
    catch(err) {  
        gs.log("A runtime error occurred: " + err);  
    }  
  
})(current, previous);
```

5. Examine the pseudo-code for the script you will write:
 - If the RCA field has no value and the RCA included checkbox is selected (checked)
 - De-select the RCA included checkbox (uncheck).
 - Else if the RCA field has a value and the RCA included checkbox is not selected (not checked)
 - Select the RCA included checkbox (check).

6. Write the script in the try:

```
(function executeRule(current, previous /*null when async*/) {  
    try {  
        if(current.u_rca.nil() && current.u_rca_included) {  
            current.u_rca_included = false;  
        }  
        else if(!current.u_rca.nil() && !current.u_rca_included) {  
            current.u_rca_included = true;  
        }  
    }  
  
    catch(err) {  
        gs.log("A runtime error occurred: " + err);  
    }  
}  
})(current, previous);
```

7. Select **Submit**.

C. Test Your Work

1. Open an Incident record.
2. Select (check) the **RCA included** checkbox.
3. **Save** the record, remain on the form.
4. Is the *RCA included* checkbox unchecked? If not, debug and re-test.
5. Enter a value of your choice in the **RCA** field.
6. **Save** the record, remain on the form.
7. Is the *RCA included* checkbox checked? If not, debug and re-test.
8. Does the *RCA included* field need to remain visible on the form? _____
9. Make the *5.2 RCA Included Business Rule* **inactive**.

Lab Completion

You have successfully completed the lab and practiced using the current and previous global objects.

Display Business Rules and Dot Walking

Lab
05.03

⌚15-20m

Lab Summary

You will achieve the following:

- Write a script using `g_scratchpad` global object, Display Business Rules, and dot-walking.

A. Create a Business Rule

1. Create a new Business Rule.

Name: **Lab 5.3 Display Business Rule**

Table: **Incident [incident]**

Active: **Selected (checked)**

Advanced: **Selected (checked)**

When: **display**

Order: **150**

2. Examine the pseudo-code for the script you will write:

- Add the current record's Resolved By reference object's First Name to the `g_scratchpad` object.
- Add the current record's Resolved By reference object's Last Name to the `g_scratchpad` object.
- If there is no value in `reopen_count`
 - Set the `g_scratchpad.reopenCount`'s value to zero.
- Else
 - Add the current record's Reopen Count to the `g_scratchpad` object.

3. Write the script:

```
(function executeRule(current, previous /*null when async*/) {  
  
    g_scratchpad.resolvedByFirstName = current.resolved_by.first_name;  
    g_scratchpad.resolvedByLastName = current.resolved_by.last_name;  
  
    if(current.reopen_count.nil()) {  
        g_scratchpad.reopenCount = "0";  
    }  
  
    else{  
        g_scratchpad.reopenCount = current.reopen_count;  
    }  
  
})(current, previous);
```

4. Select **Submit**.

B. Create a Client Script

1. Create a new Client Script.

Name: **Lab 5.3 ResolvedBy Client Script**
Table: **Incident [incident]**
UI Type: **Desktop**
Type: **onChange**
Field name: **State**
Active: **Selected (checked)**
Inherited: **Not selected (not checked)**
Global: **Selected (checked)**

2. Examine the pseudo-code for the script you will write:

- When the State field changes,
 - If the old State is Resolved, Closed, or Canceled and the new State is not Resolved, Closed, or Canceled
 - Display a confirmation box stating the Incident was previously resolved by <user who closed the record> and how many times the Incident has been reopened. Confirm the user really wants to reopen.
 - If the user cancels reopening the Incident
 - Set the State back to the old State value.

3. Write the script:

```
function onChange(control, oldValue, newValue, isLoading, isTemplate) {  
    if (isLoading || newValue === '') {  
        return;  
    }  
  
    if(oldValue > 5 && newValue <= 5) {  
  
        var answer = confirm("This incident was Resolved by " +  
            g_scratchpad.resolvedByFirstName + " " + g_scratchpad.resolvedByLastName  
            + " and has been reopened " + g_scratchpad.reopenCount + " times.\n\nAre  
            you sure you want to reopen it?");  
  
        if(answer == false) {  
            g_form.setValue('state', oldValue);  
        }  
    }  
}
```

4. Select **Submit**.

C. Test Your Work

1. Open an Incident in a **Resolved** state.
2. If the **Resolved by** field has no value, enter the user of your choice and **Save** the Incident.
3. Change the State to **In Progress**.
4. Did the confirmation dialog box appear? Did the g_scratchpad properties resolve correctly? If not, debug and re-test.
5. Select the **Cancel** button in the confirmation dialog box.
6. What happened to the State field? Is this the expected behavior? If not, debug and re-test.
7. Change the State to **In Progress**.
8. This time select the **OK** button in the confirmation dialog box.
9. **Save** the record, remain on the form.
10. Set the State to **Resolved**.
11. **Save** the record, remain on the form.

12. Set the State to **In Progress**.
13. Did the reopen count increase by 1 correctly? If not, debug and re-test.
14. Make the Lab 5.3 Display Business Rule **inactive**.
15. Make the Lab 5.3 ResolvedBy Client Script **inactive**.

Lab Completion

You have successfully completed the lab and practiced using the g_scratchpad global object, Display Business Rules, and dot-walking. You are ready to take it up a notch!

Business Rules: Cloud Dimensions Requirements

now.

User Interface Requirements

- | | |
|--|---|
| 1 Confirm Major Incident process is followed before a P1 is submitted | ✓ |
| 2 Enforce mandatory Incident requirements if State is Resolved or Closed | ✓ |

Database Requirements

- | | |
|--|---|
| 3 Identify Incident records with RCA documentation | ✓ |
| 4 Populate Change's CAB date field with next CAB meeting date | |
| 5 Prevent Problems from re-opening if closed for more than 30 days | |
| 6 Update Problem and Child Incidents with RCA details from parent Incident | |
| 7 Implement SLA targets and identify Incidents in danger of breaching them | |
| 8 Automatically assign Incidents to Assignment group members | |

Security Requirements

- | |
|------------------------|
| 9 Track Impersonations |
|------------------------|

Lab 5.2 fulfills requirement 3.

Business Rules

now.

Good Practices

- Use Async Business Rules whenever possible
- Use Display Business Rules to pass data from the server-side to the client-side during a form load using the g_scratchpad method
- Write condition statements in the Condition field and not in the script so unnecessary scripts do not run
- Choose an appropriate debugging strategy for your situation
- Add the Description field to the Business Rule form to document for yourself and others what the rule does
- **Comment your scripts!!**

Module Recap: Business Rules

now.

Core Concepts

Business Rules execute when records are accessed
When
Insert/Update/Query/Delete
Condition

Debug Business Rules with log messages written to files or to forms

Use Global Object Variables: previous, current, g_scratchpad

Dot-walking allows 'table hopping' between related tables

Real World Cases

• **Why** would you use these capabilities?

• **When** would you use these capabilities?

• **How often** would you use these capabilities?

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Module 6: GlideSystem

now.

| |
|--------------------|
| Scripting Overview |
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Explore the GlideSystem API
- Practice with GlideSystem methods
- Write, test, and debug Business Rules using GlideSystem methods

Labs

Lab 6.1 Setting the CAB Date

Lab 6.2 Re-open Problem Date Validation

PLEASE NOTE:

- This module demonstrates how to use various GlideSystem methods using Business Rules,
- Once you learn how to use these Class methods, **they can be included in any server-side script.**

The GlideSystem API

now.

- Collection of methods
- Execute server-side
- **Access system-level information**

– Logged in user



– System



– Date and time



Logged in user is **Beth Anglin**
Today is **7 September 2017**
Write debugging information



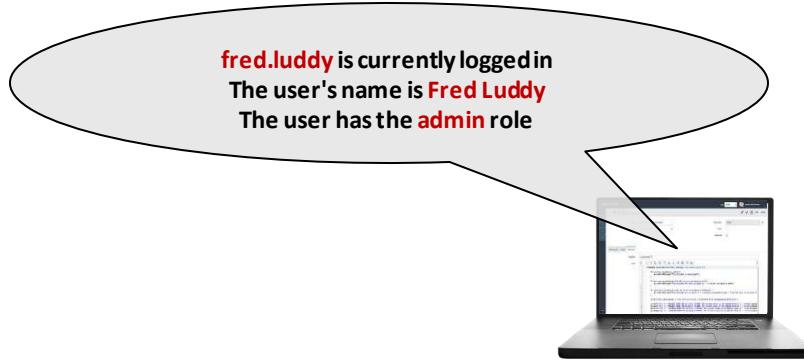
GlideSystem methods all begin with **gs**. For example, the method **gs.addInfoMessage()** displays informational messages to the user at the top of a form or list.

The complete set of GlideSystem methods can be found on the ServiceNow Developer site.

User Methods



- Return information about the currently logged in user
- Examples
 - `getUser()`, `getUserName()`, `getUserID()`, `getUserDisplayName()`
 - `hasRole()`, `hasRoleInGroup()`



getUser() – returns a reference to the *User object* for the current user.

getUserDisplayName() – returns the *name field* of the current user (*Fred Luddy instead of fred.luddy*).

getUserID() – returns the *sys_id* of the current user.

getUserName() – returns the *username* of the current user (*fred.luddy*).

hasRole() – returns *true* if the user has the specified role.

hasRoleInGroup() – returns *true* if the user has the specified role within a specified group.

User Method Examples



```
if(gs.hasRole('itil')){  
    gs.addInfoMessage("getUser() returns: " + gs.getUser());  
    gs.addInfoMessage("getUserName() returns: " + gs.getUserName());  
    gs.addInfoMessage("getUserID() returns: " + gs.getUserID());  
    gs.addInfoMessage("getUserDisplayName() returns: " + gs.getUserDisplayName());  
}
```

```
getUser() returns: com.glide.sys.User@36426f72  
getUserName() returns: beth.anglin  
getUserID() returns: 46d44a23a9fe19810012d100cca80666  
getUserDisplayName() returns: Beth Anglin
```

This example shows the different values returned by the getUserXXXX() methods.

addInfoMessage() – adds a blue informational message to the top of a form or list.

addErrorMessag() – adds a red warning message to the top of a form or list.

System Methods

now.



- Primarily work with form objects, tables, fields, and logging
- Examples
 - `getProperty()`, `getPreference()`
 - `getDisplayColumn()`, `tableExists()`
 - `nil()`, `eventQueue()`
 - `print()`, `log()`, `logError()`, `getMessage()`

The display value for sys_user is **Beth Anglin**
The assigned_to field does not have a value
`glide.ui.chart.height=300`



getProperty() – returns the value of a Glide property.

getPreference() – gets a user preference.

getDisplayColumn() – gets the display column for the table.

tableExists() – determines if a database table exists by returning true.

nil() – returns true if a field's value is null or an empty string ("").

eventQueue() – queues an event for the event manager.

print() – writes a message to the system log.

log() – logs a message to the system log and saves it to the syslog table.

logError() – logs an error to the system log and saves it to the syslog table.

getMessage() – Retrieves a message from UI messages.

System Method Examples

now.



```
//if Location is empty, use the caller's location  
if(gs.nil(current.location)){  
    current.location.setValue(current.caller_id.location);  
}  
  
//Log the value of a system property  
var from = gs.getProperty('glide.email.username');  
gs.log("Email recipients will see '" + from + "' in their mail inbox FROM column");
```



The screenshot shows the ServiceNow Log viewer interface. At the top, there are buttons for Log (selected), New, Go to, Created (with a dropdown arrow), and Search. Below the buttons are filter options for Created (set to Created), Level (set to Message), and Message. The main area displays a single log entry:

| Created | Level | Message |
|---------------------|-------------|---|
| 2017-01-19 15:23:28 | Information | Email recipients will see 'IT Service Desk' in their mail inbox FROM column |

Date and Time Methods

now.



- Work with Date and Time data
- Examples
 - beginningOfLastWeek()/endOfLastWeek(),
beginningOfNextMonth()/endOfNextMonth()
 - dateDiff()
 - now(), nowDateTime()
 - minutesAgo(), quartersAgo(), monthsAgo(), yearsAgo()



Today is **April 10, 2018**.
2 hours ago it was **13:15:32**.
It is not the first day of the week.

beginningOfLastWeek() – date and time for the beginning of last week in GMT (yyyy-mm-dd hh:mm:ss).

beginningOfNextMonth() – date and time for the beginning of next month in GMT.

dateDiff() – calculates the difference between two dates.

endOfLastWeek() – date and time for the end of last week in GMT.

endOfNextMonth() – date and time for the end of next month in GMT.

minutesAgo()/daysAgo()/quartersAgo()/monthsAgo()/yearsAgo() – returns the date and time a specified number of minutes/quarters/months/years ago. Negative numbers will show a date and time in the future.

now() – current date.

nowDateTime() – current date and time in the user-defined format.

Date and Time Method Examples

now.



```
var firstDay = current.u_hire_date.getDisplayValue();
var lastDay = current.u_last_date.getDisplayValue();

//If firstDay and lastDay both have values, check to ensure lastDay isn't before firstDay
if(firstDay && lastDay) {
    if(gs.dateDiff(firstDay, lastDay, true) < 0){
        gs.addErrorMessage("The last day worked must be after the hire date");
        current.u_last_date = '';
        current.setAbortAction(true);
    }
}
```

This example was prepared using GlideSystem methods in a Business Rule on the User [sys_user] table.

When passed a date field as an argument, the **getDisplayValue()** method returns the date and time in the user's time zone rather than in GMT.

GlideDate and GlideDateTime Method Example



- The **GlideDate** and **GlideDateTime** APIs are used to manipulate date and time values.
- You can create a **GlideDateTime** object from a **GlideDate** object by passing in the **GlideDate** object as a parameter to the **GlideDateTime** constructor.

```
var gDate = new GlideDate();
gDate.setValue('2015-01-01');
gs.info(gDate);

var gDT = new GlideDateTime(gDate);
gs.info(gDT);
```

Output:

2015-01-01

2015-01-01 00:00:00

The **GlideDateTime** class provides methods for performing operations on **GlideDateTime** objects, such as instantiating **GlideDateTime** objects or working with *glide_date_time* fields.

In addition to the instantiation methods described above, a **GlideDateTime** object can be instantiated from a *glide_date_time* field using the **getGlideObject()** method.

For example:

```
var gdt = gr.my_datetime_field.getGlideObject();
```

Note: Methods can use the Java Virtual Machine time zone, so use equivalent local time and UTC methods whenever possible to prevent unexpected result.

GlideSystem Method Documentation

now.

The screenshot shows the ServiceNow developer API documentation for the **GlideSystem** module. On the left, a sidebar lists various methods under the **GlideSystem** category, including `addErrorMessage`, `addInfoMessage`, `addMessage`, `beginningOfLastMonth`, `beginningOfLastWeek`, `beginningOfNextMonth`, `beginningOfNextWeek`, `beginningOfThisYear`, `beginningOfThisMonth`, `beginningOfThisQuarter`, `beginningOfThisWeek`, `beginningOfThisYear`, `beginningOfToday`, `beginningOfYesterday`, `calDateDiff`, `dateDiff`, `dateGenerate`, `daysAgo`, and `daysAgoEnd`. The main content area displays the `addErrorMessage` method documentation. It includes a brief description, usage notes (mentioning `getErrorMessage()`), parameter details (a table with columns Name, Type, and Description), and return information (a table with columns Type and Description). A code example section shows a snippet of JavaScript code demonstrating password validation logic.

The complete set of **GlideSystem** methods can be found on the ServiceNow Developer site.

1. Select **API**.
2. Select **Server**.
3. Select **LEGACY**.
4. Locate and expand the **GlideSystem** category on the left.

Module Labs

now.

- **Lab 6.1**

- **Time:** 10-15m
- Setting the CAB Date
 - Write a Business Rule using GlideSystem date and time methods

- **Lab 6.2**

- **Time:** 10-15m
- Re-open Problem Date Validation
 - Write a Business Rule using GlideSystem date and time methods



Setting the CAB Date

Lab
06.01

⌚10-15m

Lab Summary

You will achieve the following:

- Write a Business Rule using GlideSystem date and time methods to set the CAB date for a Change Request to Wednesday of next week.
- Mark a Cloud Dimensions' Phase II requirement complete.



Business Problem

2nd level IT Analysts from various teams have requested the Change Management team use the CAB date field on the Schedule tab to inform them at which CAB meeting their Change Request will be reviewed. The Change Management team has been doing their best to populate the field manually, but many times the information is missed.

As CAB meetings are held every Wednesday, the Change Management team has requested the field be automatically populated with next Wednesday's date. Using this strategy, they will only be required to update the field if there is an exception vs. populating the date for every new Change Request.

Project Requirement

Set the CAB date to Wednesday of next week for new Change Requests.

A. Create a Business Rule

1. Create a new Business Rule.

Name: **Lab 6.1 Set CAB Date**
Table: **Change Request [change_request]**
Active: **Selected (checked)**
Advanced: **Selected (checked)**
When: **before**
Order: **300**
Insert: **Selected (checked)**

2. On the Advanced tab, within the `executeRule()` function in the Script field, type `try` followed by the `<tab>` key to insert the `try` Syntax Editor Macro (*created in Lab 2.1*).
3. Select the **Format Code** icon on the Syntax Editor toolbar to properly align the code.
4. Update the statement in the `catch` block to use the server-side `gs.log()` method instead of the client-side `g_form.addErrorMessage()` method.
5. Examine the pseudo-code for the script you will write:
 - Set the CAB date to next Monday.
 - Convert the CAB date to a numeric value (UNIX time format).
 - Calculate the number of milliseconds in two days.
 - Add two days to the CAB date.
 - Set the CAB date to Wednesday next week.
6. Write the script in the `try` block:

```
(function executeRule(current, previous /*null when async*/) {  
  
    try {  
        current.cab_date.setDisplayValue(gs.beginningOfNextWeek());  
        var nextMonday = current.cab_date.dateNumericValue();  
        var twoDays = 2*24*60*60*1000; //Milliseconds  
        var nextWednesday = nextMonday + twoDays;  
        current.cab_date.setDateNumericValue(nextWednesday);  
    }  
  
    catch(err) {  
        gs.log("A runtime error occurred: " + err);  
    }  
})(current, previous);
```

7. Select **Submit**.

B. Test Your Work

1. Create a new Change Request.
2. Select **Normal: Changes without predefined plans that require approval and CAB authorization**.
3. Enter **Testing Lab 6.1** in the *Short description* field.
4. **Save** the record, remain on the form.
5. Navigate to the **Schedule** tab.
6. Is the **CAB date** correct? If not, debug and re-test.

Hint: A common mistake is to forget to multiply by 1000. Make sure to account for the milliseconds to get the correct date.

7. Make the Business Rule **inactive**.

Lab Completion

Great job! You successfully completed the requirement and practiced using many different GlideSystem methods in a server-side script.

Re-open Problem Date Validation

Lab
06.02
⌚10-15m

Lab Summary

You will achieve the following:

- Write a Business Rule using GlideSystem date and time methods to verify if a Problem hasn't been closed for more than 30 days if a user attempts to re-open the record.
- Complete a Cloud Dimensions' Phase II requirement.



Business Problem

After analyzing the data, the Problem Management team determined it is common practice for IT Analysts to re-open Problem records to document ongoing work performed on the same Configuration Item (CI). In some cases, the re-open was valid. In other cases, the re-open was not valid, as it was a new issue altogether.

After digging a little deeper, the team also determined where there was a valid re-open, the failure of the permanent fix presented itself in less than one week.

To eliminate the practice of incorrectly re-opening Problem records, the Problem Management team has put a policy in place stating Problem records cannot be re-opened after 30 days. The team feels this window is sufficient to catch any valid re-opens. They are requesting ServiceNow be configured to help enforce this policy.

Project Requirement

Prevent Problem records from re-opening if closed for more than 30 days.

A. Update a UI Action

1. Navigate to **Problem > All** on the Application Navigator.
2. Select a **Problem**.
3. On the form's Context menu () , Select **Configure -> UI Actions**.
4. Search the **Name** column for the UI Action **Re-Analyze**.
5. Select **Re-Analyze**.
6. Examine the pseudo-code for the script you will write:

- Read today's date into a variable.
- Read the Closed date into a variable.
- Determine the difference between today and the closed date.
- Calculate 30 days, read the answer into a variable.
- If the number of days between today and the Closed date is larger than 30,
 - Add an error message to the form page.
- else
 - Allow the problem to be re-analyzed

7. Write the script.

- a) Scroll down to the **Script** field.

- b) Write the script:

```
action.setRedirectURL(current);

var today = gs.nowDateTime();
var closed = current.closed_at;
var diff = gs.dateDiff(closed, today, true);
var thirtyDays = 30*24*60*60; //Seconds

if (diff > thirtyDays) {
    gs.addErrorMessage("A problem can not be reopened after it has been
closed for more than 30 days. Please open a new Problem.");
}
else {
    new ProblemStateUtils().onReAnalyze(current);
}
```

8. Select **Update**.



TIP FROM THE FIELD:

There is a quick way to open a specific UI Action. Navigate to the form. **Right Click** the UI Action above the Related Links. Select **Edit UI Action**.

B. Test Your Work

1. Navigate to **Problem > All** on the Application Navigator.
2. Open any *closed* Problem record.
3. Ensure the value in the **Completed** field (*on the Other Information tab*) is greater than 30 days.
4. Select the UI Action **Re-Analyze**.
5. **Save** the record; remain on the form.
6. Do you see an error message at the top of the form? If not, debug and re-test.

C. Revert to the Original Version

1. Navigate to the UI Action **Re-Analyze**.
2. Scroll down to the **Versions** Related Lists.
3. Select the Version record with the state **Previous**.

| | Name | Recorded at | State | Source |
|--------------------------|---|---------------------|----------|--|
| <input type="checkbox"/> | sys_ui_action_2bd3fb087a313000e3dd61e36cb0bce | 2018-12-10 09:13:09 | Current | Update Set: Default |
| <input type="checkbox"/> | sys_ui_action_2bd3fb087a313000e3dd61e36cb0bce | 2018-11-14 12:53:28 | Previous | System Upgrades: glide-11-13-2018_1930 |

4. Select the Related Links **Compare to Current**.
5. Select the **Script** field.

| Selected Version | Current Version |
|--|---|
| <pre>1 action.setRedirectURL(current); 2 new ProblemStateUtils().onReAnalyze(current);</pre> | <pre>1 action.setRedirectURL(current); 3 var today = gs.nowDateTime(); 4 var closed = current.closed_at; 5 var diff = gs.dateDiff(closed, today, true); 6 var thirtyDays = 30*24*60*60; //Seconds 8 if (diff > thirtyDays).{ 9 gs.addErrorMessage("A problem can not be reopened after 30 days"); 10 } 11 else { 12 new ProblemStateUtils().onReAnalyze(current); 13 }</pre> |

6. Compare the **Selected Version** with the **Current Version**.

7. If the selected version matches the original script, select **Cancel** to return to the page Compare to Current.
8. At the bottom of the page, select **Revert to Selected Version**.
9. Select **OK** and a message will appear to state the ui action has been updated to the previous version.
10. Navigate to the UI Action **Re-Analyze** to confirm it has been restored to original version.

Lab Completion

Well done! You successfully completed the requirement and practiced comparing dates in the Global space using GlideSystem methods.

Scoped GlideSystem

now.

- You must use Scoped APIs when developing scripts for scoped applications
- The **Scoped GlideSystem** API provides methods for accessing system-level information in scoped applications
 - Logged in user
 - General system
 - Logging
- Error messages advise developers when methods are used in the wrong scope

Function log is not allowed in scope x_travel. Use gs.debug() or gs.info() instead



Scoped Glide APIs do not provide all the methods included in the global Glide APIs, and you cannot call a global Glide API in a scoped application. The complete set **Scoped GlideSystem** methods can be found on the ServiceNow Developer site.

1. Select **API**.
2. Select **Server**.
3. Select **SCOPED**.
4. Locate and expand the **GlideSystem** category on the left.



IMPORTANT

The *Scoped GlideSystem* API does not provide methods to work with date/time fields. Use **GlideDate** or **GlideDateTime** methods (Scoped and Legacy). Detailed information on both of these APIs can be found on the ServiceNow Developer site.

Scoped GlideSystem Logging Methods

Write log messages in a private application scope

- debug()
- error()
- info()
- warn()
- isDebugging()

| Created | Level | Message |
|---------------------|-------------|---------------------------------|
| 2017-01-19 16:18:57 | Error | This message is from gs.error() |
| 2017-01-19 16:26:21 | Information | This message is from gs.info() |
| 2017-01-19 16:26:21 | Warning | This message is from gs.warn() |

13:39:15.267: Global 🎨 ==> 'Scoped Logging Methods' on incident:INC0000015
 13:39:15.270:: This message is from gs.info()
 13:39:15.270:: This message is from gs.warn()
 13:39:15.272:: This message is from gs.error(): no thrown error
 13:39:15.272:: [DEBUG] This message is from gs.debug()
 13:39:15.273: Global 🎨 <== 'Scoped Logging Methods' on incident:INC0000015

Navigate to **System Logs** to review log message output

gs.debug() log messages are located at the bottom of a form when **Session Debugging** is turned on

Scoped GlideSystem logging methods are available for scripts in private application scopes.

- **debug()** – sets the log Level to 'Debug'. Use to log informational messages useful for debugging.
- **error()** – sets the log Level to 'Error'. Use to log messages that might still allow the system to continue running.
- **info()** – sets the log Level to 'Information'. Use to log informational messages describing progress.
- **warn()** – sets the log Level to 'Warning'. Use to log potentially harmful messages.
- **isDebugging()** – determines if debugging is active for a specific scope.

Navigate to **System Logs > System Log > All** to locate *gs.error()*, *gs.warn()*, and *gs.info()* output. These messages along with messages from *gs.debug()* are also located at the bottom of a form when System Debugging is turned on.

Example Using Logging Methods

now.

```
(function executeRule(current, previous /*null when async*/) {
    try {
        if(current.u_city.nil()) {
            current.u_city.setValu(current.u_attendee.location.city + ", " +
current.u_attendee.location.state)
        }
    } catch(err) {
        gs.error("This error occurred at runtime: " + err);
    }
})(current, previous);
```

Spelling Error

Navigate to **System Logs > System Log > Errors**

The screenshot shows the ServiceNow System Log interface. At the top, there are navigation buttons: a menu icon, 'Log' (which is selected and highlighted in blue), 'New', 'Created', 'Search', 'Grid', and 'Split'. Below the buttons, a table displays a single log entry. The entry includes a checkbox, a timestamp ('2017-09-12 14:02:20'), a severity level ('Error'), and a message ('This error occurred at runtime: TypeError: Cannot find function setValu in object.'). The message is highlighted with a red border.

If using logging methods in production, always select the appropriate log type for the event or log message (*error, info, warn or debug*).

GlideSystem: Cloud Dimensions Requirements

now.

User Interface Requirements

- | | |
|--|---|
| 1 Confirm Major Incident process is followed before a P1 is submitted | ✓ |
| 2 Enforce mandatory Incident requirements if State is Resolved or Closed | ✓ |

Database Requirements

- | | |
|--|---|
| 3 Identify Incident records with RCA documentation | ✓ |
| 4 Populate Change's CAB date field with next CAB meeting date | ✓ |
| 5 Prevent Problems from re-opening if closed for more than 30 days | ✓ |
| 6 Update Problem and Child Incidents with RCA details from parent Incident | |
| 7 Implement SLA targets and identify Incidents in danger of breaching them | |
| 8 Automatically assign Incidents to Assignment group members | |

Security Requirements

- | |
|------------------------|
| 9 Track Impersonations |
|------------------------|

Lab 6.1 fulfills requirement 4.

Lab 6.2 fulfills requirement 5.

Good Practices

- Use the appropriate object methods based on your scope
 - GlideSystem
 - Scoped GlideSystem
- Write validation scripts for forms with dates
 - Start date validation
 - End date validation
- Use the GlideSystem date methods rather than JavaScript date methods for convenience
- Always check the ServiceNow Developer's portal for the latest information on the **GlideSystem** and **Scoped GlideSystem** APIs and their methods

Module Recap: GlideSystem

now.

Core Concepts

The GlideSystem and Scoped GlideSystem APIs provide methods to access system-level information

GlideSystem methods fall into three categories:

- User
- General
- Date/Time

Scoped GlideSystem methods fall into three categories:

- User
- General
- Logging

Methods execute server-side

Real World Cases

• **Why** would you use these capabilities?

• **When** would you use these capabilities?

• **How often** would you use these capabilities?

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Module 7: GlideRecord

now.

| |
|--------------------|
| Scripting Overview |
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Explore the GlideRecord API
- Learn how to write a GlideRecord query
- Determine if records are returned from a query
- Take action on returned records

Labs

- Lab 7.1 Two GlideRecord Queries
Lab 7.2 RCA Attached: Problem and Child Incidents
Lab 7.3 addEncodedQuery()

PLEASE NOTE:

- This module demonstrates how to use various GlideRecord methods using Business Rules.
- Once you learn how to use these Class methods, **they can be included in any server-side script.**

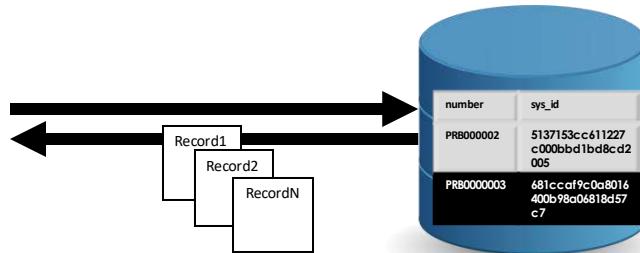
What is a GlideRecord?

now.

- Used for database operations instead of writing SQL queries
- An object containing zero or more records from the same table (*ordered list*)
- Contains records (rows) and fields (columns)
- Executes server-side

```
var problems = new GlideRecord('problem');
problems.addQuery('rfc', '=', me.sys_id);
problems.query();

while (problems.next()) {
    problems.problem_state.setValue(4);
    var notes = problems.close_notes.getValue();
    notes = notes + '\n' + msg;
    problems.close_notes.setValue(notes);
    problems.update();
}
```



Any table row from any table can become a GlideRecord.

GlideRecord queries can be called client-side but this should be avoided due to performance impact. The `getReference()` method previously discussed in *Module 3 – Client Scripts* performs a GlideRecord query.

Working with GlideRecord Queries

now.

Strategy for creating a new GlideRecord query

1. Create a GlideRecord object for the table of interest
2. Build the query condition(s)
3. Execute the query
4. Process returned records with script logic



This strategy outlines the necessary steps for creating and populating a GlideRecord object, then processing any returned records.

Create New

now.

Step 1: Create a GlideRecord Object for the Table of Interest

- To query a table, first create an object for the table
- Object is called a GlideRecord
- Only parameter needed is the name of the table

```
var myObj = new GlideRecord('table_name');           ← Syntax  
var myObj = new GlideRecord('change_request');      ← Example
```

select * from change_request

The example creates a variable called myObj which is a GlideRecord object for the Change Request [change_request] table.

addQuery()

Step 2: Build the Query Condition(s)

- **addQuery()** builds a SQL select statement (not seen by the user)
- Each addQuery() call adds a new "where" clause to the select statement
- addQuery() calls are automatically AND'ed

```
var myObj = new GlideRecord('change_request');
myObj.addQuery('category', '=', 'Hardware');
myObj.addQuery('priority', '!=', 1);

$$\uparrow \downarrow$$

select * from change_request where category = 'Hardware' AND priority != 1;
```

The example shown only builds the select statement; it does not execute it.



TIP FROM THE FIELD

In a SQL statement , the “where” clause has a direct effect on how long it takes to return data. Since the query conditions generate a SQL statement, use the appropriate query conditions and the right number of them to generate the correct “where” clause. If you do not use the correct query conditions, performance of a query can be degraded.

addQuery() Operators

- Used to search the database for matching records

| Numeric | String | | |
|---------|--------|------------|----------------|
| = | != | = | != |
| > | >= | STARTSWITH | ENDSWITH |
| < | <= | CONTAINS | DOESNOTCONTAIN |

- Must be in quotes

```
myObj.addQuery('priority','!=',1);
myObj.addQuery('work_notes','CONTAINS','Fixed');
```

- In the absence of an operator, method assumes condition to test is equality

```
myObj.addQuery('category','Hardware');
```

Is the same as

```
myObj.addQuery('category','=', 'Hardware');
```

Numeric comparison operators:

- > field must be greater than the value supplied
- >= field must be equal or greater than the value supplied
- < field must be less than the value supplied
- <= field must be equal or less than the value supplied

String comparison operators:

STARTSWITH – field value must start with the value supplied

ENDSWITH – field value must end with the value supplied

CONTAINS – field value must contain the value supplied somewhere in the text

DOES NOT CONTAIN – field value must not have the value supplied anywhere in the text

Both numeric and string comparison operators:

- = field must be equal to the value supplied
- != field must not equal the value supplied

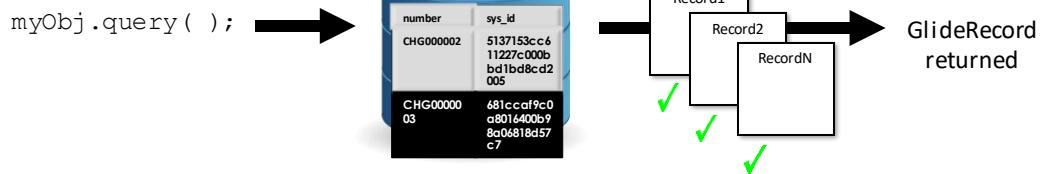
See ServiceNow's Product Documentation for information on additional operators.

Query

Step 3: Execute the Query

- Use the **query()** method to execute the query
- Returns zero or more rows
- Each record returned is a GlideRecord object

```
var myObj = new GlideRecord('change_request');
myObj.addQuery('category', '=', 'Hardware');
myObj.addQuery('priority', '!=', 1);
myObj.query();
```



The **_query()** method provides the same functionality as the **query()** method. It is intended to be used on tables where there is a column named 'query', which would interfere with using the **query()** method.

Process Returned Record

now.

Step 4: Process Returned Records With Script Logic

- **next()** moves to the next record in the GlideRecord object
 - **while()** iterates through all returned records

```
while (myObj.next ()) {  
    // script logic  
    // Reference current record as <object name>.<property> (e.g. myObj.state)  
}
```
 - **if()** process only the first record returned

```
if (myObj.next ()) {  
    //script logic  
}
```
- **hasNext()** steps through returned records and determines if there are any more records

```
if (myObj.hasNext ()) {  
    //script logic  
}
```

.hasNext() returns true when there is a next record, but does not load that record into the object for processing. Select this method to execute script logic when a next record simply exists, but access to the record's data is not needed.

By default queries with invalid field names run but ignore the invalid condition. A true/false *glide.invalid_query.Returns_no_rows* System Property can be added to the System Property table with its value set to true to instead have invalid queries return no rows in the GlideRecord object.

The method **_next()** has the same functionality as *next()*. It is intended to be used on tables where there is a column named 'next', which would interfere with using the *next()* method.

Update

now.

- Use the **update()** method to save changes to the records in the GlideRecord
- If record does not exist, it is inserted

```
var incGR = new GlideRecord('incident');
incGR.addQuery('problem_id', '=', current.sys_id);
incGR.query();

while (incGR.next()) {
    incGR.work_notes = "Related Problem " + current.number + "
        closed with the following Close Notes:\n\n" +
        current.close_notes;
incGR.update();
}
```

In the example shown, the `work_notes` fields in the returned Incident records are updated with the value currently in the related Problem's `close_notes` field.

Put it All Together

now.

```
//Step 1: Create a GlideRecord object for the table of interest
var myObj = new GlideRecord('table_name');

//Step 2: Build the query condition(s)
myObj.addQuery('field_name', 'operator', 'value');
myObj.addQuery('field_name', 'operator', 'value');

//Step 3: Execute the query
myObj.query();

//Step 4: Process returned record(s) with script logic
while(myObj.next ()) {
    //Script logic
    //Reference current record as <object name>.<property> (e.g. myObj.state)
    //Use myObj.update() to save changes made to records
}
```

To automatically insert GlideRecord syntax containing an *addQuery()* method, use the Syntax Editor Macro **vargr**. This macro inserts an **if()** at end of the code stub, in most cases you will want to change this to a **while()** so you can execute the script logic on all returned rows and not just the first one.

To insert GlideRecord syntax containing an *addOrCondition()* method, use the script macro **vargor**.

addOrCondition()

now.

- Use the **addOrCondition()** method to add a new condition to a select statement using **OR**
- Works with **addQuery()**

```
var myObj = new GlideRecord('change_request');
var q1 = myObj.addQuery('category', '=', 'Hardware');
q1.addOrCondition('priority', '!=', 1);
↑
select * from change_request where category = 'Hardware' OR priority != 1;
```

Use an object variable and **addQuery()** to add the first condition and **addOrCondition()** for the second condition.

addOrCondition()

Mixing ANDs and ORs

- Create "groups" to represent ()
- (A or B) and (C or D)

```
var myObj = new GlideRecord('incident');
var q1 = myObj.addQuery('state','<',3);
q1.addOrCondition('state','>',5);
var q2 = myObj.addQuery('priority',1);
q2.addOrCondition('priority',5);
myObj.query();
```



```
select * from incident where (state < 3 OR state > 5) AND (priority = 1 OR priority = 5);
```

Each 'group' in the select statement is its own object in the script.

get()

Query for a Single Record

- **get(Object name, Object value)**
- Used to query for a single record
- Returns
 - **True** if a record is found
 - **False** if a record is not found

```
if(!current.parent_incident.nil()) {  
    var myObj = new GlideRecord('incident');  
    myObj.get(current.parent_incident);  
    myObj.u_rca = true;  
    myObj.update();  
}
```



Unique value

Defines a GlideRecord based on the specified expression of **name = value**. If value is not specified, then the expression used is **sys_id = name**.

This method performs a *next()* operation before returning.



TIP FROM THE FIELD

If you are not 100% sure the *get()* method will return a record, consider using an *if()* condition to first check if you retrieved a GlideRecord. If not, script logic can be written to handle that scenario.

Confirm Records Returned

Did My Query Return GlideRecords?

- **getRowCount()** method

```
var myObj = new GlideRecord('incident');
myObj.addQuery('caller_id',current.caller_id);
myObj.query();
var retRows = myObj.getRowCount();
gs.log("Returned number of rows = " + retRows);
```

- **GlideAggregate** object and methods

```
var count = new GlideAggregate('incident');
count.addQuery('caller_id',current.caller_id);
count.addAggregate('COUNT');
count.query();
var incidents = 0;
if (count.next()){
    incidents = count.getAggregate('COUNT');
}
gs.log("Returned number of rows = " + incidents);
```

Both of these options return the number of rows meeting the query criteria.

The first option, **getRowCount()**, is a lot less code to write but has a significantly larger performance impact and is not recommended for use with tables with large amounts of data. This strategy retrieves the rows one by one and increments a counter.

The second method, **GlideAggregate()** is more work for the developer, but results in a faster execution. This strategy executes a true COUNT select statement: select COUNT(*) from table_name; This strategy is always recommended for cases where the returned number of rows is greater than 100 and for tables that grow continuously. If in doubt, use this strategy.

See the ServiceNow Developer site for more information on the [GlideAggregate API](#).

addEncodedQuery()

now.

- Builds SQL select statement (not seen by the user)
- Passes all query where clauses as a single argument
- Best option for complex queries

```
glideRecordObject.addEncodedQuery('where clauses');           ← Syntax  
myObj.addEncodedQuery('approval=approved^assigned_to=  
681b365ec0a80164000fb0b05854a0cd^Orassigned_to=f298d2  
d2c611227b0106c6be7f154bc8^ed_dateONThis  
year@javascript:gs.beginningOfThisYear()@javascript:g  
s.endOfThisYear()');
```

← Example

Although this example looks overwhelmingly complex, it is easy to build the where clause(s) using the Condition builder.

addEncodedQuery()

Build an Encoded Query

1. Open the list for the table of interest
2. Build the query using the Condition Builder
3. Run the query to ensure correct results
4. Right-click the breadcrumbs and select **Copy query**



Remember, you can type `<table name>.list` into the Application Navigator's Search field to quickly open the list view for a table (e.g. entering `sys_user.list` will open the list view of the User [sys_user] table in the content pane).

When right-clicking the breadcrumbs and selecting **Copy query**, the query copied to the clipboard will contain only those parts of the query from where you clicked and to the left. In this example, if you right-click on *Approval=Approved*, you will only have that part of the where clause. If you right-click on *Planned end date on This year*, you will copy the entire set of where clauses.



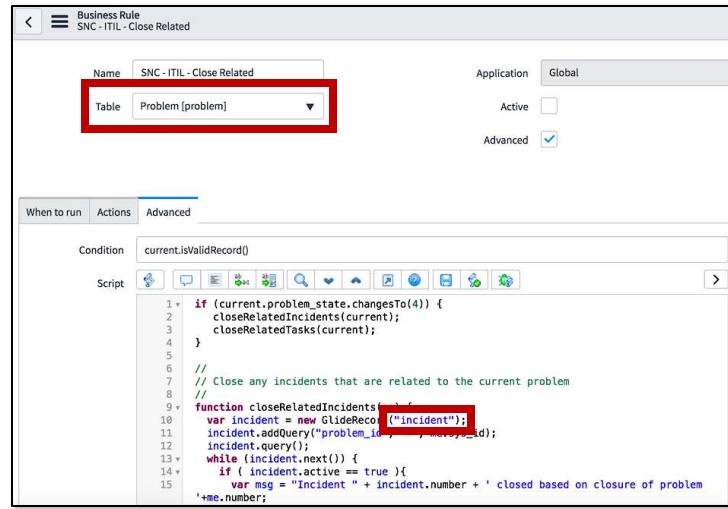
IMPORTANT

Always test queries on a development instance prior to deploying them to a production instance. An incorrectly constructed encoded query (e.g. including an invalid field name), produces an invalid query. When the invalid query executes, the invalid part of the query condition is dropped, and the results are based on the valid part of the query, which may return all records from the table. Using an **insert()**, **update()**, **deleteRecord()**, or **deleteMultiple()** method on bad query results can result in data loss.

Updating Records

Update Records On Another Table

- Table specified in the GlideRecord object can be different from the table identified in the Business Rule



For example, the baseline **SNC - ITIL - Close Related** Business Rule closes all related Incident records when a Problem record closes. The Business Rule identifies the Problem table in its trigger, and the GlideRecord query updates the appropriate Incident records.

Documentation

GlideRecord Methods

The screenshot shows the ServiceNow developer documentation interface. At the top, there's a navigation bar with links for PROGRAM, LEARN, API, COMMUNITY, EVENTS, and BLOG. On the right side of the header are links for FEEDBACK, LOG IN, REGISTER, and MORE SITES, along with a search bar. Below the header, there are two tabs: 'SCOPE' and 'LEGACY'. The 'LEGACY' tab is selected. A search bar is located above the main content area. The main content area has a title 'GlideRecord' with a subtitle 'GlideRecord is used for database operations.' It describes the class as being used for both records and fields, mentioning its inheritance from GlideRecordSecure and its enforcement of ACLs. It also notes that invalid queries on a sub-production instance will return no records if the system property 'glide.invalid_query_returns_no_rows' is set to true. Below this, there's a section for the 'addActiveQuery()' method, which adds a filter to return active records. It includes a table for 'Return' with columns 'Type' and 'Description', showing 'QueryCondition' as the type and 'Filter to return active records.' as the description. There's also a note about 'Scoped equivalent' and an example code snippet:

```
var inc = new GlideRecord("incident");
inc.addActiveQuery();
inc.query();
```

The complete set of **GlideRecord** methods can be found on the ServiceNow Developer site.

1. Select **API**.
2. Select **Server**.
3. Select **LEGACY**.
4. Locate and expand the **GlideRecord** category on the left.

- Class inherited from GlideRecord
- Performs the same functions as GlideRecord **AND enforces ACLs**

Non-writable Fields

- Are set to NULL when trying to write to the database
- `canCreate()` on the column is replaced with `canWrite()`
- If it returns false, the column value is set to NULL

Checking for NULL Values

- If an element cannot be read because an ACL restricts access, a NULL value is created in memory for that record
- You do NOT have to check for read access
`if (!grs.canRead()) continue;`
- The `next()` method simply moves to the next record in the GlideRecord object

GlideRecord can handle a variety of data manipulation tasks, but can it restrict data to certain users? Yes, the GlideSystem methods discussed in Module-6, like `hasRole()`, `canRead()`, and `canWrite()` used in conjunction with GlideRecord will restrict data to users that have the correct roles. However, you must include the appropriate conditional checks every time data is queried. This strategy is NOT considered best practice.

GlideRecord example

```
var count = 0;
var gr = new GlideRecord('mytable');
gr.query();
while (gr.next()) {
    if (!gr.canRead()) continue;
    if (!gr.canWrite()) continue;
    if (!gr.val.canRead() || !gr.val.canWrite())
        gr.val = null;
    else
        gr.val = "val-" + gr.id;
    if (gr.update())
        count++;
}
```

GlideRecordSecure example

```
var count = 0;
var grs = new GlideRecordSecure('mytable');
grs.query();
while (grs.next()) {
    grs.val = "val-" + grs.id;
    if (grs.update())
        count++;
}
```

Module Labs

- **Lab 7.1**

- **Time:** 20-25m
- Two GlideRecord Queries
 - Practice using Business Rules to locate records from one table and update another table

- **Lab 7.2**

- **Time:** 20-25m
- RCA Attached: Problem and Child Incidents
 - Create a Business Rules to update records and related records

- **Lab 7.3**

- **Time:** 5-10m
- addEncodedQuery()
 - Practice creating an Encoded Query and using addEncodedQuery() in a GlideRecord



Two Glide Record Queries

Lab Summary

You will achieve the following:

- Write a GlideRecord query to locate a set of Incidents
- Write a second GlideRecord that:
 - query to locate a set of users.
 - Trigger the Business Rule from one table, but make updates to records on a different table.

A. Script 1: Find Active SAP Incidents

1. Create a new Business Rule.

Name: **Lab 7.1 SAP Incidents**
Table: **Incident [incident]**
Active: **Selected (checked)**
Advanced: **Selected (checked)**
When: **after**
Order: **150**
Insert: **Selected (checked)**
Update: **Selected (checked)**

2. Switch to the **Advanced** tab.
3. Within the **executeRule()** function in the Script field, type **vargr** followed by the **<tab>** key to insert the standard GlideRecord syntax (*stored in the baseline Syntax Editor Macro*).
4. Select the **Format Code** icon on the Syntax Editor toolbar to properly format the script.

5. Update the Script Macro to suit your current requirements.
 - a) Update the name of the GlideRecord object to something more meaningful (considered good practice).
 - b) Replace the word "if" with **while**.
 - c) *Optional: Insert a blank line between the query() statement and the while() statement for easy readability.*
 - d) Compare your script with this code. Make any adjustments if necessary.

```
(function executeRule(current, previous /*null when async*/) {  
  
    var sapIncs = new GlideRecord("");  
    sapIncs.addQuery("name", "value");  
    sapIncs.query();  
  
    while (sapIncs.next()) {  
  
    }  
  
})(current, previous);
```

6. Update the GlideRecord query to find all **active Incidents** where **Short Description contains SAP**.

```
(function executeRule(current, previous /*null when async*/) {  
  
    var sapIncs = new GlideRecord("incident");  
    sapIncs.addActiveQuery();  
    sapIncs.addQuery("short_description","CONTAINS","SAP");  
    sapIncs.query();  
  
    while (sapIncs.next()) {  
  
    }  
  
})(current, previous);
```

7. Now update the Script to identify the Incidents matching the query criteria by their Number. You can use this script as a guide, or any other identification strategy of your choice.

```
(function executeRule(current, previous /*null when async*/) {  
  
    var sapIncs = new GlideRecord("incident");  
    sapIncs.addActiveQuery();  
    sapIncs.addQuery("short_description", "CONTAINS", "SAP");  
    sapIncs.query();  
  
    var myLog = "";  
  
    while (sapIncs.next()) {  
        myLog += sapIncs.number + ", "  
    }  
  
    gs.addInfoMessage("These records are active SAP Incidents: " + myLog);  
})(current, previous);
```

8. Select **Submit**.

B. Test Your Work

1. Type **incident.list** in the Application Navigator's Search field, and press <Enter> on your keyboard to open the list view of the Incident table.
2. Use the Condition Builder to build the same query you just scripted.
3. Run the query and record the number of returned rows here: _____.
4. Create a new Incident and populate the mandatory fields with values of your choice.
5. **Save** the record, remain on the form.
6. Did your script locate the expected records? If not, debug and re-test your Business Rule.
7. Make the Lab 7.1 SAP Incidents Business Rule **inactive**.

C. Script 2: Update Executive User Records

1. Using the GlideRecord strategies, you have learned so far, create, test, and debug a GlideRecord query on your own.
 - a) Find all users with Vice, VP, or Chief in their title.
 - b) Make the users VIPs.

Hint: *VIP is a Boolean field on the User [sys_user] table. The VIP field is not displayed on the User records unless the form/list is configured.*

Hint: *Use the GlideRecord update() method to modify the User records.*

Note: *Experienced scripters are encouraged to do this without looking at provided code, however, if you are new to scripting and require assistance, you will find an example of a Business Rule that can be used as a guide on the next page.)*

2. Did your script locate and update the expected records? If not, debug and re-test your Business Rule.
3. Make the Business Rule **inactive**.

Lab Completion

Great job! You thoroughly understand how to query the database using GlideRecord methods. If you were able to complete the second Business Rule on your own, bonus points for you.

D. Business Rule Example for Script 2

1. Create a new Business Rule.

Name: **Lab 7.1 VIP Users**
Table: **Incident [incident]**
Active: **Selected (checked)**
Advanced: **Selected (checked)**
When: **after**
Order: **100**
Insert: **Selected (checked)**
Update: **Selected (checked)**

Script:

```
(function executeRule(current, previous /*null when async*/){  
  
    var makeVIP = new GlideRecord('sys_user');  
    q1 = makeVIP.addQuery('title','CONTAINS','VP');  
    q1.addOrCondition('title','CONTAINS','Vice ');  
    q1.addOrCondition('title','CONTAINS','Chief');  
    makeVIP.query();  
  
    while(makeVIP.next()){  
        makeVIP.vip = true;  
        gs.log("ADMIN: " + makeVIP.name + " with title: " +  
makeVIP.title + " is now a VIP");  
        makeVIP.update();  
    }  
  
})(current, previous);
```

2. Test the Script

- a) Use the Condition Builder on the Users [sys_user] table to build the same query you just scripted. Make a mental note of the number of returned rows.
- b) Create a new Incident.
- c) **Submit** the record.
- d) Navigate to **System Logs > Script Log Statements** to review the log message.
- e) Did your script locate the expected records? If not, debug and re-test your Business Rule.
- f) Make the Business Rule **inactive**.

Lab 07.02

⌚20-25m

RCA Attached: Problem and Child Incidents

Lab Summary

You will achieve the following:

- Create a Business Rule to update related Problem and Child Incident records. When this lab is complete, you will mark a Cloud Dimensions' Phase II requirement complete.



Business Problem

The Problem Management team currently updates Problems and Child Incidents manually with the Root Cause Analysis (RCA) details from Parent Incident records. The team is requesting automation be put in place to relieve them of the tedious and time-consuming updates.

Project Requirement

Create a Business Rule to update Problem and Child Incident records with RCA details from the Parent Incident. To ensure the RCA details are finalized, ensure the Business Rule only executes when the State of the Parent Incident changes to Closed.

A. Create a Business Rule

1. Create a new Business Rule.

Name: **Lab 7.2 RCA Update PRB and Child INCs**
Table: **Incident [incident]**
Active: **Selected (checked)**
Advanced: **Selected (checked)**
When: **after**
Order: **100**
Insert: **Selected (checked)**
Update: **Selected (checked)**
Condition: **current.state.changesTo(IncidentState.CLOSED)**

2. Examine the pseudo-code for the script you will write:

- When the value of State changes to Closed
 - If the Problem field has a value
 - Create a new GlideRecord object for the Problem table
 - Get the current Problem into the new GlideRecord object
 - Update the Problem's Work notes with the RCA source and current value of RCA
 - Create a new GlideRecord object for the Incident table
 - Query for incidents where the value of Parent Incident is the same as the current Incident
 - Query the database
 - While there are records returned
 - Update the Child Incident's RCA field to indicate the RCA source and current value of RCA

3. Write the script:

```
(function executeRule(current, previous /*null when async*/) {  
  
    if(!current.problem_id.nil()){  
        var prbRecord = new GlideRecord('problem');  
        prbRecord.get(current.problem_id);  
        prbRecord.work_notes += "\n\nRCA from " + current.number + ": " + current.u_rca;  
        prbRecord.update();  
    }  
  
    var childIncs = new GlideRecord("incident");  
    childIncs.addQuery("parent_incident", current.sys_id);  
    childIncs.query();  
  
    while (childIncs.next()) {  
        childIncs.u_rca += "RCA from " + current.number + ": " + current.u_rca;  
        childIncs.update();  
    }  
})(current, previous);
```

4. Select **Submit**.

B. Test Your Work

1. Create a new Incident.

- a) Configure the record:

Caller: <Select any User of your choice>

Short description: **THIS IS MY PARENT INCIDENT**

Problem (*Related Records tab*): <Select any "active" Problem record>

- b) Write the Problem number you chose here: _____.

- c) Write the Incident number here: _____.

- d) Select **Submit**.

2. Create another new Incident.

- a) Configure the record:

Caller: <Select any User of your choice>

Short description: **This is Child Incident #1**

Set **Parent Incident** to the Incident from step 1.

- b) Select **Submit**.

3. Create another new Incident.
 - a) Configure the record:

Caller: <Select any User of your choice>
Short description: **This is Child Incident #2**
Set **Parent Incident** to the Incident from step 1.
 - b) Select **Submit**.
4. Open the Parent Incident from step 1.
5. Navigate to the **Child Incidents** Related List and confirm both of your Child Incidents appear in the list. If not, review your Child Incident records and ensure the Parent Incident is identified in both of the records.
6. Update these fields:

RCA: **Parent Incident RCA field contents**
State: **Closed**
Resolution code: **Solved (Permanently)**
Resolution notes: **Resolution notes from my Parent Incident**
7. Select **Update**.
8. Open the Problem you recorded in step 1b. Do you see the Root Cause Analysis data in the Activity list? If not, debug and re-test.
9. Open each of the Child Incident records. Are the RCA fields populated? If not, debug and re-test.
10. Make the Business Rule **inactive**.

Lab Completion

Well done! You have successfully updated records on two different tables using one Business Rule.

addEncodedQuery()

Lab
07.03

5-10m

Lab Summary

You will achieve the following:

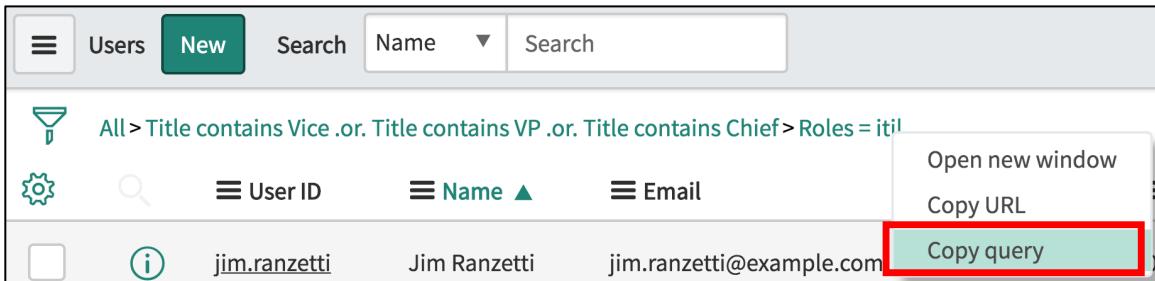
- Re-write and augment a previous lab's GlideRecord query using the addEncodedQuery() method.

Lab Dependency: You must have the **Lab 7.1 VIP Users Business** complete to do this lab.

A. Preparation

1. Open the VIP Business Rule you wrote in Lab 7.1 (users with Vice, VP, or Chief in their title).
2. Rename the Business Rule **Lab 7.3 addEncodedQuery**.
3. Make the script **Active**.
4. Select **Insert** on the form's Context menu.
5. Navigate to **User Administration > Users**.
6. Use the Condition Builder to build a query looking for users with Vice, VP, or Chief in their title.
7. Add an additional query to search for users who have the itil role.
8. Run the query.
9. Record the number of records meeting the criteria here: _____.

10. Right-click the right-most condition of the query in the breadcrumbs and select **Copy query**.



B. Modify the Script

1. Re-write the *Lab 7.3 addEncodedQuery* Business Rule script to use the **addEncodedQuery()** method.
 - a) Delete any script statements that are no longer needed.
 - b) Add the **addEncodedQuery()** method to the script.
 - c) Paste the copied query from your clipboard into the **addEncodedQuery()** method as the parameter.

```
(function executeRule(current, previous /*null when async*/) {  
  
    var makeVIP = new GlideRecord("sys_user");  
    makeVIP.addEncodedQuery("titleLIKEVice^0RtitleLIKEVP^0RtitleLIKEChief^roles=itil");  
    makeVIP.query();  
  
    while (makeVIP.next()) {  
        makeVIP.vip = true;  
        gs.log(makeVIP.name + " with the title: " + makeVIP.title + ", is now a VIP");  
        makeVIP.update();  
    }  
  
})(current, previous);
```

2. Select **Update**.

C. Test Your Work

1. Force the *Lab 7.3 addEncodedQuery* Business Rule to execute.
2. Was the expected number of records returned? If not debug and re-test your Business Rule.
3. Make the Business Rule **inactive**.

Lab Completion

Good job! You have successfully used the `addEncodedQuery()` GlideRecord method.

Scoped GlideRecord

now.

- You must use Scoped APIs when developing scripts for scoped applications
- The **Scoped GlideRecord** API provides methods for database operations in scoped applications
- The complete set of methods is available on the ServiceNow developer site



Follow these steps to locate Scoped GlideRecord information ServiceNow Developer site.

1. Select **API**.
2. Select **Server**.
3. Select **SCOPED**.
4. Locate and expand the GlideRecord category on the left.

Client Scripts: Cloud Dimensions Requirements

now.

User Interface Requirements

- | | |
|--|---|
| 1 Confirm Major Incident process is followed before a P1 is submitted | ✓ |
| 2 Enforce mandatory Incident requirements if State is Resolved or Closed | ✓ |

Database Requirements

- | | |
|--|---|
| 3 Identify Incident records with RCA documentation | ✓ |
| 4 Populate Change's CAB date field with next CAB meeting date | ✓ |
| 5 Prevent Problems from re-opening if closed for more than 30 days | ✓ |
| 6 Update Problem and Child Incidents with RCA details from parent Incident | ✓ |
| 7 Implement SLA targets and identify Incidents in danger of breaching them | |
| 8 Automatically assign Incidents to Assignment group members | |

Security Requirements

- | |
|------------------------|
| 9 Track Impersonations |
|------------------------|

Lab 7.2 fulfills requirement 6.

Good Practices

- Use the appropriate object methods based on your scope
 - GlideRecord
 - Scoped GlideRecord
- Use GlideAggregate instead of getRowCount() for large tables and production instances
- Perform all record queries in server-side scripts for improved performance
- Use _next() and _query() for tables containing next and query columns
- When deleting/inserting/updating records with GlideRecord methods, practice writing the list of records to a log to insure you do not accidentally modify records you still need before enabling the methods that modify the database (***there is no undo!***)

Module Recap: GlideRecord

now.

| Core Concepts | Real World Cases |
|---|--|
| GlideRecords are rows returned from the database | • Why would you use these capabilities? |
| A GlideRecord query can be created for any table | • When would you use these capabilities? |
| The query() method executes a select statement against a database table | • How often would you use these capabilities? |
| Use a while loop to iterate over returned GlideRecords | |
| GlideAggregate() and getRowCount() tell how many rows are returned to a GlideRecord | |
| Use encoded queries for complex query conditions | |

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Module 8: Script an Event

now.

| |
|--------------------|
| Scripting Overview |
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Define what it means to be an Event
- Know how to generate an Event
- Create Script Actions to respond to Events
- Determine if an Event is generated

Labs

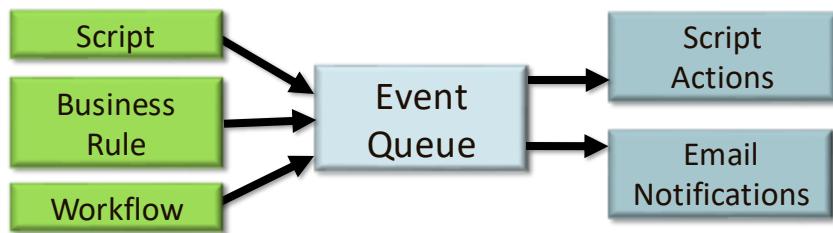
- Lab 8.1 Tracking Impersonations
Lab 8.2 Incident State Event

Script an Event: Overview

now.

What is an Event?

- Indication **something notable has occurred in the system**
- Recorded in the **Event Queue** when the event occurs
- Examples
 - A record is inserted, updated, deleted, or queried
 - User has logged in
 - Log information, warning, or error
 - Impersonation started



Although any server-side script can generate an event, most will be generated by Business Rules or Workflows.

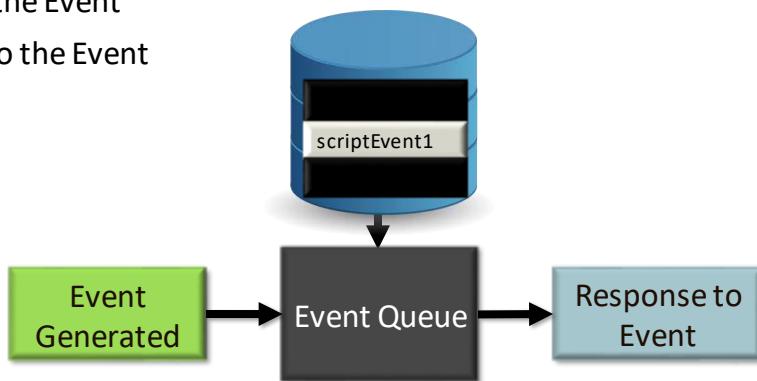
Events are not actions but rather an entry in the Event Queue.

Logic must be written to respond to entries in the Event Queue and take action.

Working With Events

now.

- To create an Event
 1. Register the Event
 2. Create a queue (*optional*)
 3. Create a script to generate the Event
 4. Create a script to respond to the Event



The **Event Registry** table records the Events that can be called and responded to. Although scripts can call any Event, if it is not registered, no response will occur.

Unless otherwise specified, all user-created events appear in the default queue.

Create an Event

Step 1: Register the Event

- Open System Policy > Events > Registry
- Select New
- Create the Event

The screenshot shows the 'Event Registration' screen with a 'New record' button. The form contains the following fields:

- Event name:** problem.myInsertedProblem
- Application:** Global
- Table:** Problem [problem] (with a dropdown arrow)
- Fired by:** New Problem Business Rule
- Description:** This event is fired when a new problem is created
- Queue:** cloud_dimensions (labeled as 'Optional')

Event name – name used in script logic to generate the event. By convention the event name is <table name>.<event name>. The table_name is not automatically added to the Event name and must be added manually.

Table – database table for the event.

Fired By – list of scripts calling this event. This is used for troubleshooting only and not for event processing. Whenever you add an event to a script, update this field.

Description – documents the event purpose. Be sure to put a meaningful description for future reference. If you alter the meaning of an event, update this field.

Application – identifies the scope containing the event.

Queue – the event processing queue which manages this event. If left blank, the event is processed by the default queue. User-defined queues should be all lowercase and use underscores to separate words.

ServiceNow cannot respond to events unless they have been registered. The method used to generate events can be passed any string as the event parameter. Events generated with unregistered event names will be inserted in the event queue but no ServiceNow processes will respond to the event queue entry.

Create a New Event

Step 2: Create a Queue (optional)

- Open System Scheduler > Scheduled Jobs > Scheduled Jobs
- Open the **text index events process** Scheduled Job
 - IMMEDIATELY select **Insert and Stay** to make a copy of the record
- Modify the parameters of the new record
 1. Change the value in the **Name** field
 2. Pass the user-defined queue name in the **Job context** field



Schedule Item
cloud dimensions events process

| | | | |
|-------------|---|--------|--------------|
| Name | cloud dimensions events process | Job ID | RunScriptJob |
| Next action | 2017-01-25 11:42:00 | State | Ready |
| Calendar | | Parent | |
| Job context | #Wed Jan 25 09:40:33 PST 2017 fcScriptName=javascript\GlideEventManager('cloud_dimensions').process(); | | |
| System ID | -- None -- | | |



CAUTION!

Do not overwrite the **text index events process** Schedule Item!

Overwriting the text index events process Schedule Item removes ServiceNow's ability to process that baseline queue. Use **Insert and Stay** as soon as possible to save your new Schedule Item.

When you are creating a queue, the value passed to the *GlideEventManager()* method must match the value of the Queue field in the Event Registry.

By default, the scheduler checks for messages in this queue every 30 seconds. (Trigger type = Interval and Repeat = 30 seconds.)

Sending an event to the **text index events process** queue could potentially delay processing of higher priority events. For example, if an event is generated every time a record is imported, a large Import Set could delay processing of the more important Priority1Event.

Use this feature only when there is a possibility of long Script Action Processing times or rapid generation of events causing high volumes in the queues.

Create a New Event

now.

Step 3: Create a Script to Generate the Event

- Use the **gs.eventQueue()** GlideSystem method in any server-side script
- Syntax:

```
gs.eventQueue("<event_name>", object, parm1, parm2);
```

- Example:

```
gs.eventQueue("problem.myInsertedProblem", current, "Number: " +  
current.number, "Created by: " + gs.getUserDisplayName());
```

event_name – name of Event in Registry.

object – current, previous, or a GlideRecord Object.

parm1 – optional parameter used to pass a string. If you do not need to pass this parameter the convention is to pass `gs.getUserID()`.

parm2 – optional parameter used to pass a string. If you do not need to pass this parameter the convention is to pass `gs.getUserName()`.

parm3 – optional parameter can be used to pass the name of a queue. This parameter overwrites a value provided by the Queue field on the Event's Registry record. Syntax example:

```
gs.eventQueue("problem.myInsertedProblem", current, current.number,  
gs.getUserDisplayName(), "js_queue");
```

parm1 and parm2 can be strings ("hello world"), properties (`current.assigned_to`), variables (`myString`), or methods (`gs.getUserID()`).

The example shown calls the registered event `problem.myInsertedProblem` and passes it the current object, the number of the current record, and the value of `gs.getUserDisplayName()`.

Create a New Event

now.

Generate the Event at a Fixed Time

- Use the `gs.eventQueueScheduled()` GlideSystem method
- Syntax:

```
gs.eventQueueScheduled("<event_name>", object, parm1, parm2, time);
```

- Example:

```
gs.eventQueueScheduled("problem.reminder", current, "Event time: " +
    current.uReminder, gs.getUserName(),
    current.uReminder);
```

The time parameter is set to a record field containing a Date/Time value. The event is generated at the time passed in this field.

The example shown calls the registered event and passes it the current object, the string "Event time:" + the value in the *current.u_reminder* field, the value of *gs.getUserName()*, and the date/time for the event to be generated.

Did the Event Occur?

now.

Checking for Events

- Check for generated events in **System Policy > Events > Event Log**
 - **Created** is the time when the event was added to the queue
 - **Parm1** and **Parm2** fields show resolved values
 - **Processed** is the time when the event was processed
 - **Processing duration** shown in milliseconds
 - If the **Queue** field is blank, the event was processed by the default queue



| Events | | | | | | |
|------------------------|--------------------------|--|---|--|--|--|
| | New | Created | Search | Grid | Split | |
| All > Created on Today | | | | | | |
| | <input type="checkbox"/> | <input type="checkbox"/> Created | <input type="checkbox"/> Name | <input type="checkbox"/> Parm1 | <input type="checkbox"/> Parm2 | <input type="checkbox"/> Table |
| | <input type="checkbox"/> | 12:53:50 | problem.myInsertedProblem | Number: PRB0040010 | Created by: System Administrator | problem |
| | | | | | 2021-08-14 12:54:08 | 2 cloud_dimensions |

Created – date and time of the event for the locale of the machine running the ServiceNow instance.

Name – name of the event.

Parm1 – event-specific value dependent on the event and the recipient.

Parm2 – event-specific value dependent on the event and the recipient.

Table – database table acted on by this event.

Processed – date and time the event was processed. This time reflects the locale of the machine running the ServiceNow instance.

Processing duration – time taken to process this event in milliseconds.

Queue – processor queue name.

The **Processed** time should be within 30 seconds (or whatever the Interval is for the Schedule Item) of the *Created* time.

Script a Response

Step 4: Create a Script to Respond to the Event

- **Script Actions** respond to Events
- Triggered when an event is recorded in the Event queue
- Has access to two objects:
 1. The **current** object passed in from `gs.eventQueue()`
 2. The **event** object

The screenshot shows the 'Script Action' configuration page. The 'Name' field is set to 'Check for Existing PRB'. The 'Event name' is 'problem.myInsertedProblem'. The 'Execution Order' is '100'. The 'Application' is 'Global'. The 'Active' checkbox is unchecked. In the 'Script' field, there is sample code:

```

1 //Script logic goes here. The Script Action has access to parameters passed in by the
2 //gs.eventQueue() method: the object, event.parm1 and event.parm2
3 //
4 //Regardless of which object is passed in from the calling script, the object is now
5 //referenced as 'current' in the Script Action

```

New Script Actions are NOT 'Active' by default

The 'object' passed in from `gs.eventQueue()` is now called 'current'

Scripts can be set to execute whenever a particular activity occurs in the system, rather than at a particular time. Navigate to **System Policy > Events > Script Actions** to create a new Script Action.

Name – name of the Script Action.

Event name – registered event to use for this Script Action.

Order – order in which the Script Action executes.

Application – the scope containing the Script Action.

Active - selected the check box (true) to enable the Script Action. **Note:** not selected by default.

Condition script – script the conditions under which the Script Action should execute.

Script – script the logic to execute when the Script Action runs.

The **event** and **current** objects do not exist in the **Condition script** field. Testing for conditions using these objects must be done in the **Script** field. Example: `if (event.parm1 == 'itil')`.



TIP FROM THE FIELD

Use this code to see the event object properties.

```

var output = "";
for (var property in event) {
    output += property + ': ' + event[property] + ', ';
}
gs.log("Event Object Properties: " + output);

```

Module Labs

- **Lab 8.1**

- **Time:** 10-15m
- Tracking Impersonations
 - Practice writing a Script Action that reacts to an event.

- **Lab 8.2**

- **Time:** 10-15m
- Incident State Event
 - Register, generate, and respond to an event



Tracking Impersonations

Lab
08.01

⌚10-15m

Lab Summary

You will achieve the following:

- Write a Script Action to the baseline impersonation.start event.
- Mark a Cloud Dimensions' Phase II requirement complete.



Business Problem

At Cloud Dimensions, IT Security is in place across all the systems in the Enterprise, due in large part to many of the systems containing credit card data. It is the long-term goal for their ServiceNow instance to also contain credit card data.

IT Security must show they are prepared to investigate security breaches. One of the requirements is to be able to track when System Administrators impersonate other users in their Production instance.

Project Requirement

Create an event to track impersonations and log the information. The IT Security team will use the records to build a report.

A. Review the Event Registry Record

1. Review the **impersonation.start** Event Registration record.
 - a) Navigate to **System Policy > Events > Registry**.
 - b) Locate and open the **impersonation.start** record.
 - c) Review the value in the **Description** field:
 - i. Record the value passed in event.parm1 here: Impersonating UserName
 - ii. Record the value passed in event.parm2 here: Impersonated UserName
2. Close the record.

B. Create a Script Action

1. Create a new Script Action.
 - a) Open **System Policy > Events > Script Actions**.
 - b) Select **New**.
 - c) Configure the trigger:

Name: **Lab 8.1 Tracking Impersonations**
Event name: **impersonation.start**
Active: **Selected (checked)**
2. Examine the pseudo-code for the script you will write:

Log the name of the user doing the impersonation and the user being impersonated.
3. Write the script:

```
gs.log("<YOUR INITIALS> Script: user " + event.parm1 + " impersonated user " + event.parm2);
```
4. Select **Submit**.

C. Test Your Work

1. Impersonate **Beth Anglin**.
2. While impersonating Beth Anglin, impersonate **Fred Luddy**.
3. End the impersonation. You should be logged in as the **System Administrator** when this step is complete.
4. Open **System Policy > Events > Event Log**.
5. Locate the two **impersonation.start** events.
6. Examine **Parm1** for the two events. What value does Parm1 have? Record the value here:

7. Is this the value you expected?

8. Why are there only two **impersonation.start** events in the last minute or two?

9. Open **System Logs > System Log > Script Log Statements**. Locate the two log messages your Script Action wrote. Did the Parm1 and Parm2 values resolve correctly? If not, debug and re-test.

Lab Completion

Good job! You have successfully created an event to track impersonations and log the information.

Incident State Event

Lab
08.02

⌚10-15m

Lab Summary

You will achieve the following:

- Register, generate, and respond to an event.
- Trigger the event by changes to the Incident State.

A. Register a New Event

1. Navigate to **System Policy > Events > Registry**.
2. Select **New**.
3. Configure the record:

Event name: **incident.state.changed**
Table: **Incident [incident]**
Fired by: **Lab 8.2 Incident State Changed Business Rule**
Description: **Fires when the value in the State field changes**

4. Select **Submit**.

B. Trigger an Event

1. Create a new Business Rule.

Name: **Lab 8.2 Incident State Changed**
Table: **Incident [incident]**
Active: **Selected (checked)**
Advanced: **Selected (checked)**
When: **after**
Order: **300**
Insert: **Selected (checked)**
Update: **Selected (checked)**
Condition: **current.state.changes()**

2. Examine the pseudo-code for the script you will write:

Generate the incident.state.changed event and pass it the current object, the previous incident state's display value, and the current incident state's display value.

3. Write the script.

```
(function executeRule(current, previous /*null when async*/) {  
    gs.eventQueue('incident.state.changed', current,  
    previous.state.getDisplayValue(), current.state.getDisplayValue());  
})(current, previous);
```

4. Select **Submit**.

C. Respond to the Event

1. Create a new Script Action.

- a) Open **System Policy > Events > Script Actions**.
- b) Select **New**.
- c) Configure the trigger:

Name: **Lab 8.2 Incident State Changed**
Event Name: **incident.state.changed**
Active: **Selected (checked)**

- d) Examine the pseudo-code for the script you will write:

Store parm1's value in a variable.
Store parm2's value in a variable.
Log the old and new Incident States in a warning message.

- e) Write the script:

```
var oldVal = event.parm1;  
var newVal = event.parm2;  
  
gs.log("<YOUR INITIALS> Script: The Incident's State changed from " +  
oldVal + " to " + newVal + ".");
```

- f) Select **Submit**.

D. Test Your Work

1. Open and modify an existing Incident.
 - a) Record the value of State here: _____.
 - b) Change the State to a different value.
 - c) Record the modified value of State here: _____.
 - d) Select **Update**.
2. Did the event occur?
 - a) Open **System Policy > Events > Event Log**.
 - b) Locate the **incident.state.changed** event in the *Name* column.
 - c) Are the parm1 and parm2 values what you expected? If not, debug and re-test the Business Rule.
3. Was the event responded to?
 - a) Open **System Logs > System Log > Script Log Statements**.
 - b) Locate your Script Action message.
 - i. Change the Search field type to **Message**.
 - ii. Enter **<YOUR INITIALS> Script** in the *Search* field.
 - iii. Press the **<enter>** key on your keyboard.
4. If needed, debug and re-test the Script Action.

Lab Completion

Well done! You have successfully registered, generated, and responded to an event. Use any server-side scripting techniques to respond to any event you create.

Script an Event: Cloud Dimensions Requirements

now.

User Interface Requirements

- | | |
|--|---|
| 1 Confirm Major Incident process is followed before a P1 is submitted | ✓ |
| 2 Enforce mandatory Incident requirements if State is Resolved or Closed | ✓ |

Database Requirements

- | | |
|--|---|
| 3 Identify Incident records with RCA documentation | ✓ |
| 4 Populate Change's CAB date field with next CAB meeting date | ✓ |
| 5 Prevent Problems from re-opening if closed for more than 30 days | ✓ |
| 6 Update Problem and Child Incidents with RCA details from parent Incident | ✓ |
| 7 Implement SLA targets and identify Incidents in danger of breaching them | |
| 8 Automatically assign Incidents to Assignment group members | |

Security Requirements

- | | |
|------------------------|---|
| 9 Track Impersonations | ✓ |
|------------------------|---|

Lab 8.1 fulfills requirement 9.

Script an Event

now.

Good Practices

- When Registering Events always complete:
 - Fired by
 - Description
- Use parm1 and parm2 to pass parameters to a Script Action
- Use the Event Log to see the values of parm1 and parm2 in runtime and to check when/if the event was processed
- Create queues for Events generated frequently
- Ensure new Script Actions are active

Module Recap: Script an Event

now.

| Core Concepts | Real World Cases |
|---|--|
| Events must be registered | • Why would you use these capabilities? |
| Events are simply notifications and do not take actions | • When would you use these capabilities? |
| Use the <code>gs.eventQueue()</code> method to generate events | • How often would you use these capabilities? |
| Script Actions respond to Events | |
| Script Actions have access to whatever object is passed to it, as well as the event object itself | |

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Module 9: Script Includes

now.

| |
|------------------------|
| Scripting Overview |
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Define what it means to be a Script Include
- Determine when to use Script Includes
- Write, test, and debug Script Includes
- Extend the AbstractAjaxProcessor Class

Labs

- Lab 9.1 Classless Script Include
- Lab 9.2 Create a New Class
- Lab 9.3 HelloWorld GlideAjax
- Lab 9.4 Number of Group Members
- Lab 9.5 JSON Object

In this module you will write, test, and debug Script Includes.

Script Includes Overview

Store One Classless Function

Define a New Class

Extend an Existing Class

Application Scope

Overview: What is a Script Include?

now.

- Stores reusable JavaScript for execution on the server
- Must be called to run
- Can be client-callable



Script Includes load only on demand and do not impact performance.

FAQ's

What Exists Baseline?

- More than 1,100 Script Includes exist baseline

When do they execute?

- Run only when called

Where can they be found?

- Navigate to **System Definition > Script Includes** to create or modify Script Includes.



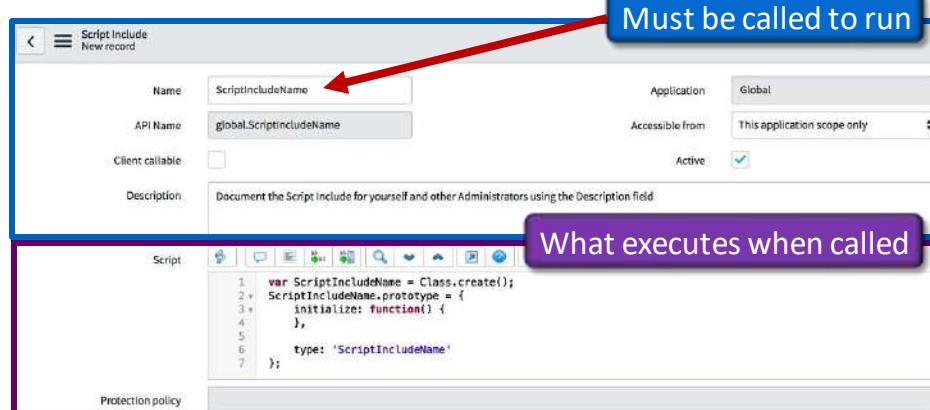
IMPORTANT

You should not modify the behavior of baseline Script Includes.

Overview: Script Include Trigger

now.

- Not many configuration options as a Script Include runs only when called
- Runs server-side
- No default objects
- Data (*parameters*) passed in from the calling script



Name – name of Script Include. No spaces, no special characters. Format is very important!

API Name – The API name of the script, including the scope and script name.

Client Callable – select this option if client-side scripts can call the Script Include.

Application – identifies the scope of the Script Include.

Accessible from – sets the accessibility of the Script Include.

- All application scopes: sets the Script Include to be global, and can be accessed from any script.
- This application scope only: sets the Script Include to be private and can only be accessed by other scripts in the same scope.

Active – select this option if the Script Include is executable. If this option is not selected the Script Include will not run even if called from another script.

Description – (optional but highly recommended) documentation explaining the purpose and function of the Script Include.

Script – script logic to execute.



IMPORTANT

A Script Include will NOT execute if there are spaces or special characters in its name.

Overview: Three Types of Script Includes

now.

1. Store One Classless Function
 - Stores **one re-usable function**
 - Used **server-side only**
 - Can also be referred to as **On Demand**
2. Define a new Class
 - **Collection of functions**
 - Can be called by client-side scripts
 - Standard to include "Utils" in the name
 - e.g. TravelRequestUtils
3. Extend an existing Class
 - **Adds functionality to an existing Class**
 - Can be called from client-side scripts



If a script is to be used multiple times, it should be stored as a Script Include method.

Script Includes: Topics

now.

Script Includes Overview

Store One Classless Function

Define a New Class

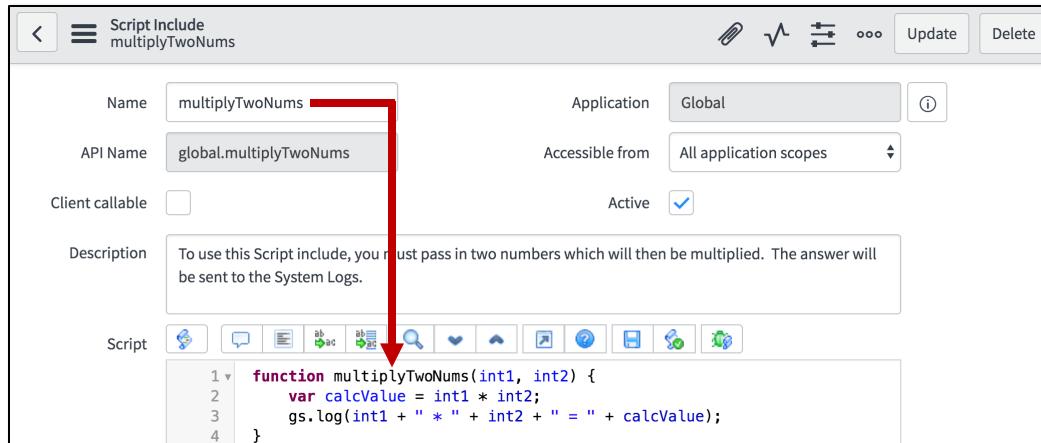
Extend an Existing Class

Application Scope

Classless/On Demand: Script Includes

now.

- Stores **one function**
- Script Include name **MUST** be identical to the function name



Classless/on-demand Script Includes are not callable from the client-side even if the Client callable option is selected.

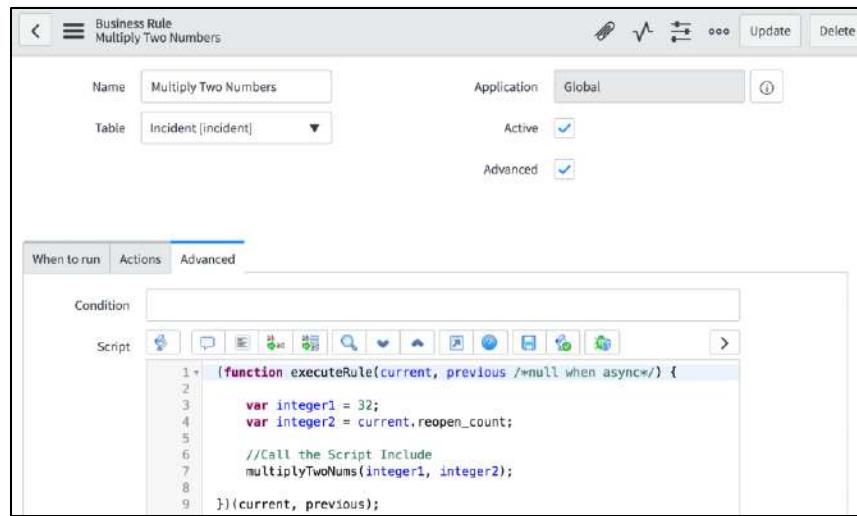
Delete the code template automatically inserted in the Script field after the Script Include is named.

If a second function is included in the script, it is only available for use after the first function has executed. This strategy is not considered best practice. If you wish to create a script with multiple functions, it is better to define a new Class or extend an existing Class.

Classless/On Demand: Use a Classless Script Include

now.

- Call the function from any server-side script



Most Script Includes, when called server-side, are called from Business Rules.

To use a Classless Script Include from a Business Rule, simply reference the function.

You will see later in this module how Class-based Script Includes are handled differently.

Module Labs

- **Lab 9.1**
 - **Time:** 10-15m
 - Classless Script Include
 - Create, test, and debug a classless Script Include



Classless Script Include

Lab
09.01

⌚10-15m

Lab Summary

You will achieve the following:

- Create, test, and debug a classless Script Include to write logging messages for server-side scripts.

A. Create a Classless Script Include

1. Navigate to **System Definition > Script Includes**.
2. Select **New**.
3. Configure the record:

Name: **logPropertyValues**
Accessible from: **All application scopes**
Description: **Lab 9.1 server-side logging**

4. Delete the Script Include template code stub currently in the *Script* field.
5. Examine the pseudo-code for the script:
 - Create a new function that will be passed a string when it is called.
 - Set the `debug` property on the object to `true`.
 - Set the `debugPrefix` property on the object to `<YOUR INITIALS>`:
 - If the `debug` property is `true`
 - Write a log message containing the `debugPrefix` property and the passed in string.
 - 6. Write the script:

```
function logPropertyValues(str) {  
    this.debug = true;  
    this.debugPrefix = "<your_intials>: ";  
    if (this.debug) {  
        gs.log(this.debugPrefix + str);  
    }  
}
```

7. Select **Submit**.

B. Use the Classless Script Include

1. Create a new Business Rule.

Name: **Lab 9.1 Script Include Logging**
Table: **Incident [incident]**
Active: **Selected (checked)**
Advanced: **Selected (checked)**
When: **After**
Insert: **Selected (checked)**
Update: **Selected (checked)**

2. Examine the pseudo-code for the script:

- For the properties in the current object
 - If the property has a value
 - Call the Logging Script Include function with the concatenated string.
 - Set the property and the property value.

3. Write the script:

```
(function executeRule(current, previous /*null when async*/) {  
  
    for (var property in current) {  
        if (current[property]) {  
            logPropertyValues(property + ", " + current[property]);  
        }  
    }  
})(current, previous);
```

4. Select **Submit**.

C. Test Your Work

1. Create a new Incident and populate many of the fields with values of your choice.
2. Select **Submit**.
3. Open **System Logs > System Log > Script Log Statements**. Do the Incident's properties and values appear here? If not, debug and re-test.

4. Modify the *Script Include* by setting **this.debug = false**.
5. Create a new Incident and populate the fields with the values of your choice.
6. **Submit** the Incident.
7. Open **System Logs > System Log > Script Log Statements**. Do the new Incident's properties and values appear here? Should they appear? Explain your reasoning:

8. Explain why a Script Include to write log data with an "on/off" parameter is useful for debugging scripts:

9. Make the Lab 9.1 Script Include Logging Test Business Rule **inactive**.
10. Do you need to make the **logPropertyValues** Script Include inactive? Why or why not? Is there a performance impact if it remains active? Explain your reasoning:

Lab Completion

Well done! You have successfully created a classless Script Include and called it from a Business Rule.

Script Includes: Topics

now.

Script Includes Overview

Store One Classless Function

Define a New Class

Extend an Existing Class

Application Scope

Define: a New Class

now.

- Stores **multiple functions**
- After the record is named, a Script Editor Macro is inserted in the Script field to help guide you
- Script Include name MUST be an EXACT match to the Class name

1. Create a new Class → var HelloWorld = Class.create();
2. Create an Object → HelloWorld.prototype = {
3. Populate the Object with Methods and other Properties → initialize: function() {
},
type: 'HelloWorld'
};

Once a Script Include is named, a Script Editor Macro is automatically inserted in the Script field and the Name is propagated throughout the script. If you change the Name of the Script Include, the script will automatically update to reflect the change.



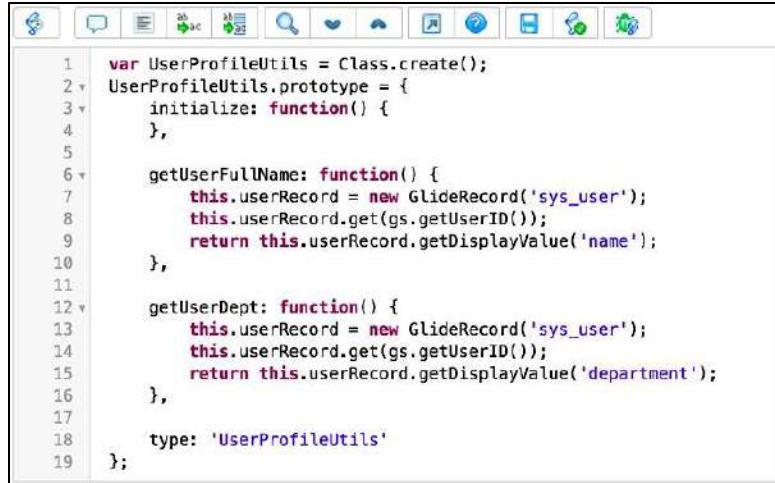
TIP FROM THE FIELD

Script Includes/Class names often include a table name (e.g. IncidentUtils, ChangeUtilsSNC, KBUtils) as a way to group/manage functions. In the case of smaller applications with minimal tables, you could instead include the application name in the Script Includes/Class name (e.g. TravelMgmtUtils).

Define: Functions

now.

- Each function definition begins with the function **name** followed by a colon
- Then the **function** keyword and parentheses
- Function arguments/parameters may be added inside the parentheses as per any function definition
- Write the script to execute between the curly braces



```
1 var UserProfileUtils = Class.create();
2 UserProfileUtils.prototype = {
3     initialize: function() {
4         },
5
6     getUserFullName: function() {
7         this.userRecord = new GlideRecord('sys_user');
8         this.userRecord.get(gs.getUserID());
9         return this.userRecord.getDisplayValue('name');
10    },
11
12    getUserDept: function() {
13        this.userRecord = new GlideRecord('sys_user');
14        this.userRecord.get(gs.getUserID());
15        return this.userRecord.getDisplayValue('department');
16    },
17
18    type: 'UserProfileUtils'
19};
```

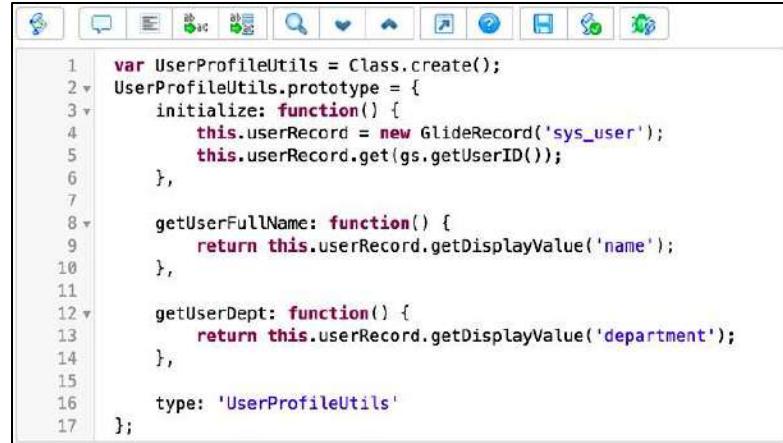
Functions are separated by commas.

Type is for allowing any developer using your Script Include to identify the type of object it is. Search Community for more information on the topic, quite a few blog articles have been written by ServiceNow experts on this advanced subject.

Define: Global Properties

now.

- Declared once at the beginning of the script in the **initialize()** function
- Available to every function in the Class
- Cleaner code



```
1 var UserProfileUtils = Class.create();
2 UserProfileUtils.prototype = {
3     initialize: function() {
4         this.userRecord = new GlideRecord('sys_user');
5         this.userRecord.get(gs.getUserID());
6     },
7
8     getUserFullName: function() {
9         return this.userRecord.getDisplayValue('name');
10    },
11
12    getUserDept: function() {
13        return this.userRecord.getDisplayValue('department');
14    },
15
16    type: 'UserProfileUtils'
17};
```

You can also define variables and objects in the `initialize()` function as well.

In this example, the logged-in user's *User [sys_user]* record was retrieved.

Module Labs

now.

- **Lab 9.2**

- **Time:** 20-25m
- Create a new Class
 - Create, test, and debug a Server-Side Script Include



Create a New Class

Lab
09.02

⌚20-25m

Lab Summary

PLEASE READ: The concept for this lab comes from a requirement many customers request when implementing Asset Management. Every organization has their specific Asset Management strategy, but for the purpose of this lab, this is the scenario you will practice using existing demo data:

The Asset Location picker should only show relevant locations based on Asset Manager roles:

- Local-level Asset Managers can assign Configuration Items to locations in their assigned Regions.
- Company-level Asset Managers can assign Configuration Items to their own corporate locations.
- System Administrators can assign Configuration Items to all locations.

This lab presumes the following regarding demo data in **Locations [cmn_location]** table:

- Corporately owned locations are identified using an ACME organization in the **Company [company]** field. Non-corporate locations listed in the *Locations* table are never identified as an ACME company.
- Corporate locations are grouped by Regions. The **Parent [parent]** field in the *Location* table is used to identify a location's Region. Non-corporate locations listed in the *Locations* table are never assigned to a Region.
- Only users with the proper permissions have 'edit' access to the Location field in Configuration Item records.

These new Groups and Roles have already been created in the platform for this lab. Users have also been added to each group for testing purposes.

Group: **Asset Managers – New York**
Role: **asset_mgr_local**
User: **Luke Wilson**

Group: **Asset Managers – Corporate**
Role: **asset_mgr_corporate**
User: **Howard Johnson**

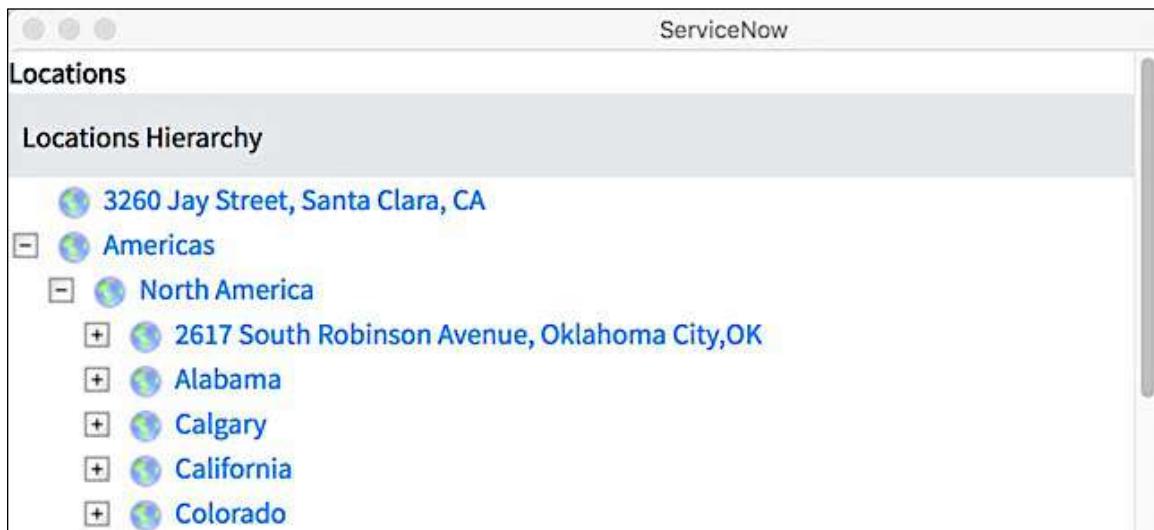
A. Preparation

1. Navigate to **Configuration > Base Items > Printers**. (Consider saving this module as a **Favorite** as you will be navigating here often throughout this lab.)
2. Select **New**.
3. Add the **Location** field to the form.

The screenshot shows the 'Printer' form in 'New record' mode. The fields visible are Name, Asset tag, Manufacturer, Asset, Company, Location, Serial number, Model ID, and Assigned to. The 'Location' field is highlighted with a red box.

4. Select the magnifying glass to the right of the **Location** field.

Note: Notice every location is available for selection. (Close this pop-up window when you are ready to proceed.)



B. Write the Script Include to Restrict Location Selections

1. Create a new Script Include.

Name: **LocationsByRole**
Accessible from: **All application scopes**
Description: **Lab 9.2 Restrict Location Options**

2. Write the script.

```
var LocationsByRole = Class.create();
LocationsByRole.prototype = {
    initialize: function() {
        //Retrieve the logged-in user's 'User [sys_user]' record
        this.loggedInUser = new GlideRecord('sys_user');
        this.loggedInUser.get(gs.getUserID());
    },

    forCMDB: function() {
        var refQual = "";
        if(gs.hasRole('admin')) {
            //Admins may assign CIs to All locations
            refQual += "^EQ";
        }
        if(gs.hasRole('asset_mgr_local')) {
            //Users with this role may assign CIs to local locations
            refQual += "parent=" + this.loggedInUser.location.parent + "^EQ";
        }

        if(gs.hasRole('asset_mgr_corporate')) {
            //Users with this role may assign CIs to corporate locations
            refQual += "company=" + this.loggedInUser.location.company + "^EQ";
        }

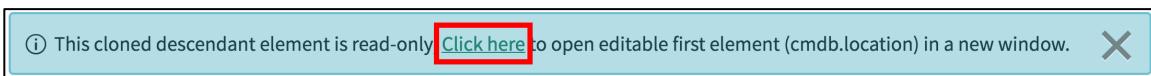
        return refQual;
    },
    type: 'LocationsByRole'
};
```

3. Select **Submit**.

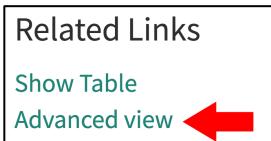
C. Configure the Location's Reference Qualifier to Call the Script Include

1. Navigate to **Configuration > Base Items > Printers**.
2. Select **New**.
3. Right-click the *Location* field's label and select **Configure Dictionary**.
4. The record is read-only as the *Location [location]* field does not exist in the Printer [cmdb_ci_printer] table. The hierarchy for this table in the platform is **Base Configuration Items [cmdb] > Configuration Items [cmdb_ci] > Printer [cmdb_ci_printer]**.

Select the **Click here** link in the InfoMessage at the top of the form to open the *Dictionary Entry* record for the *Location [location]* field in the *Base Configuration Items [cmdb]* table. The record opens in a new browser window.



5. Select the **Advanced view** Related Link.



6. Configure the fields on the **Reference Specification** tab.

User reference qualifier: **Advanced**

Reference qual: **javascript:new LocationsByRole().forCMDB()**

A screenshot of the "Reference Specification" tab configuration. The tab has several tabs at the top: "Reference Specification" (selected), "Choice List Specification", "Dependent Field", "Calculated Value", and "Default Value".

The "Reference Specification" tab content includes:

- A note: "The Reference field specifies what table this field displays values from."
- A section for "Reference": "Reference" dropdown set to "Location", a search icon, and an info icon.
- A section for "Use reference qualifier": "Use reference qualifier" dropdown set to "Advanced".
- A section for "Reference qual": A text input field containing "javascript:new LocationsByRole().forCMDB()", which is highlighted with a red box.

Note: The Scope name is not required if the Script Include is in the same scope as the Dictionary Entry record. **javascript:new global.LocationsByRole().forCMDB()** works as well.

7. While you are in the Location's Dictionary Entry record, take a minute to also disable the legacy Tree picker. (The List view is a better strategy for locating Locations quickly as it offers the use of the Condition builder to query the list.)

On the **Attributes** Related List, double-click the Tree picker's Value field and update it to **false**.



8. Select **Update**.
9. Close the browser tab you are currently on and return to the original browser tab displaying the full view of the platform.

D. Test Your Work

1. Navigate to **Configuration > Base Items > Printers**.
2. Select **New**.
3. Select the magnifying glass to the right of the **Location** field.

Notice a *List view* replaces the *Tree picker* view.

Scroll to the bottom of the window. Record the total number of records: _____.

If you did not add any Locations to the baseline demo data, there should be 425 records in the list.

4. Impersonate **Luke Wilson**. A member of the *Asset Managers – New York* Group with the *asset_mgr_local* Role, and his Parent Location (Region) is *New York*.
5. Navigate to **Configuration > Base Items > Printers**.
6. Select **New**.

7. Select the magnifying glass to the right of the **Location** field.

Notice only the two New York locations are available for selection. If not, debug and re-test.

The screenshot shows a search results page for 'Locations'. The top navigation bar includes 'Locations', 'Search' (with a dropdown menu), and a search input field. Below the header are filters for 'Name' (sorted ascending), 'City', 'State / Province', and 'Country'. The main table lists two records:

| Name | City | State / Province | Country |
|--|----------|------------------|---------|
| 322 West 52nd Street, New York, NY | New York | NY | USA |
| 450 Lexington Avenue, New York, NY | New York | NY | USA |

8. Impersonate **Howard Johnson**. A member of the *Asset Managers – Corporate Group* with the *asset_mgr_corporate* Role, and his Company is *ACME North America*.
 9. Navigate to **Configuration > Base Items > Printers**.
 10. Select **New**.
 11. Select the magnifying glass to the right of the **Location** field.
- Notice only North America corporate locations are available for selection. To be sure, quickly scan through a few pages to confirm.
- Also record the total number of records in the list now: _____. If you did not add any Locations to the baseline demo data, there should now be 372 records in the list. If not, debug and re-test.
12. End the impersonation. You should be logged in the System Administrator when this step is complete.

Lab Completion

Great job! You have successfully created and used a server-side Script Include. This strategy for restricting Locations was selected because it pushes the data to the client before the user even opens a form, thus eliminating a round-trip visit to the server and back an onLoad() Client Script would require.

Script Includes: Topics

now.

Script Includes Overview

Store One Classless Function

Define a New Class

Extend an Existing Class

Application Scope

Extend an Existing Class

now.

- Create a new Class to store new functions
- Reference an existing Class using the **extendsObject()** method

```
var MyNewUtil = Class.create();
MyNewUtil.prototype = Object.extendObject(ExistingClassNameGoesHere, {
    type: 'MyNewUtil'
});
```



My new Class includes all of the functionality in this Class, plus any new script logic I write

Extending a Class means to add functionality (typically in the form of methods) to an existing Class without modifying the original script.

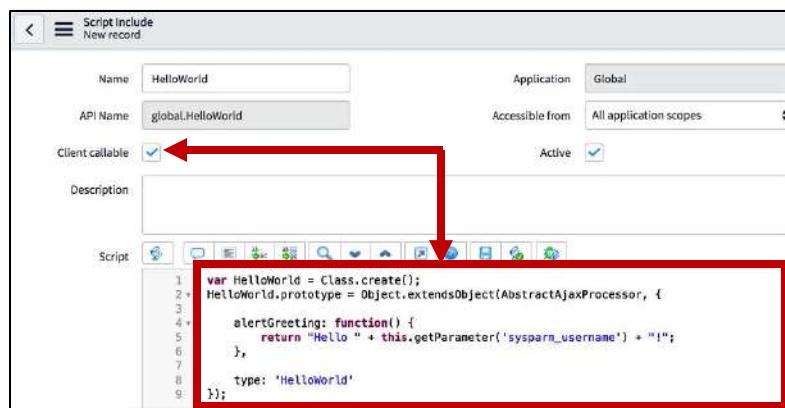
Create a New Script Include/Class, reference an existing Class using the `extendsObject()` method to include all its functionality, add script logic

Commonly extended ServiceNow Classes:

- **AbstractAjaxProcessor**: makes AJAX calls from Client Scripts.
- **LDAPUtils**: used by LDAP integration to ServiceNow (*e.g. adding Managers to users, managing group membership, debug logging*).
- **Catalog***: set of Classes used for Service Catalog management (*e.g. UI building, Form processing*).

Extend an Existing Class: Extending AbstractAjaxProcessor now.

- ServiceNow's baseline **AJAX processor Class**
- Executes server-side but is called client-side
- Code automatically inserted in the Script field when **Client callable** selected



AJAX – Asynchronous JavaScript and XML

Set of web development methods used client-side to send data to and receive data from the server.

In this example, the `alertGreeting()` method is created. When it is called, it returns the string “Hello <name passed in from the client> !” to the calling script.

Extend an Existing Class: Calling a Script Include Client-side

now.

- Use the **GlideAjax** Class

- Enables Client Scripts and UI Policies to call server-side code in Script Includes
- Pass parameters to Script Includes
- Use the returned data from the server

1. Create a new GlideAjax object for the **Script Include** of interest
2. Pass **parameters** to the Script Include
 - All parameters begin with **sysparm_**
 - Reserved parameter **sysparm_name** identifies the **method**
3. Make an asynchronous call to the server
4. Callback function processes **returned data**

```
var gaDesc = new GlideAjax('HelloWorld');
gaDesc.addParam('sysparm_name','alertGreeting');
gaDesc.addParam('sysparm_user_name','Ruth');
gaDesc.getXML(HelloWorldParse);

function HelloWorldParse(response) {
    var answerFromXML = response.responseXML;
    documentElement.getAttribute("answer");
    alert(answerFromXML);
}
```

The code shown here would be wrapped in a function (onLoad, onSubmit, onChange) depending on the type of Client Script.

In this example the GlideAjax object is used to call the Script Include and pass in parameters. The reserved parameter **sysparm_name** tells GlideAjax which function in the Script Include to use. All passed in parameters must start with **sysparm_**.

The *getXML()* method is passed the name of a callback function which parses properties and values from the XML returned from the server. (*Data is often returned as XML but can also be returned as a JSON object.*)

The *getXMLAnswer()* method can be used instead of *getXML()*. This method automatically extracts the answer variable value from the returned XML. Example:

```
var objDesc = new GlideAjax('HelloWorld');
objDesc.addParam('sysparm_name','alertGreeting');
objDesc.addParam('sysparm_var1','Ruth');
objDesc.getXMLAnswer(HelloWorldParse);
```

Extend an Existing Class: Returned to the Client from the Server

now.

- Example of XML returned from a Script Include:

```
<?xml version="1.0" encoding="UTF-8"?><xml answer="Hello Ruth!"  
sysparm_max="15" sysparm_name="alertGreeting"  
sysparm_processor="HelloWorld"/>sysparm_processor="AJAXGlideRecord"  
sysparm_type="query"><item  
sys_id="f298d2d2c611227b0106c6be7f154bc8"><active>true</active><building/  
><calendar_integration>1</calendar_integration><city/><company>31bea3d537  
90200044e0bfc8bcbe5dec</company><cost_center>d9d07bddc0a80a647cf932056ed2  
4652</cost_center><country/>  
/xml>
```

The XML in this example was captured using Firebug and has been edited to fit on the slide.

Extend an Existing Class: Putting it Together

now.

- Script Includes Script

```
var HelloWorld = Class.create();
HelloWorld.prototype = Object.extendsObject(AbstractAjaxProcessor, {
    alertGreeting: function() {
        return "Hello " + this.getParameter('sysparm_user_name') + "!";
    }
});
```

- Client-side Script

```
var greeting = new GlideAjax('HelloWorld');
greeting.addParam('sysparm_name','alertGreeting');
greeting.addParam('sysparm_user_name','Ruth');
greeting.getXML(HelloWorldParse);

function HelloWorldParse(response) {
    var answerFromXML = response.responseXML.documentElement.getAttribute("answer");
    alert(answerFromXML);
}
```

Review the two scripts side-by-side.

The client-side code shown here would be wrapped in a function (*onLoad()*, *onSubmit()*, or *onChange()*) depending on its type.

Extend an Existing Class: Another Example

now.

- Script Includes Script

```
var IdentifyTwoUsers = Class.create();
identifyTwoUsers.prototype = Object.extendsObject(AbstractAjaxProcessor, {

    userNames: function() {
        var id1 = this.getParameter('sysparm_user_id1');
        var id2 = this.getParameter('sysparm_user_id2');
        var u1  = new GlideRecord('sys_user');
        var u2  = new GlideRecord('sys_user');

        if (!u1.get('employee_number', id1))
            return '';
        if (!u2.get('employee_number', id2))
            return '';

        return u1.name + ':' + u2.name;
    },
});
```

In this example, the Script Include extends the AbstractAjaxProcessor Class.

The IdentifyTwoUsers object has a function called userNames which is passed two values:

- sysparm_user_id1
- sysparm_user_id2

The Script Include retrieves the records from the database for both *sysparm_user_id1* and *sysparm_user_id2*.

If the records both have a value in the employee_number field, the Script Include concatenates the two employee numbers with a : (colon) and returns the string to the calling script.

Extend an Existing Class: Another Example continued...

now.

- Client-side Script

```
var twoUsers = new GlideAjax('IdentifyTwoUsers');
twoUsers.addParam('sysparm_name', 'userNames');
twoUsers.addParam('sysparm_user_id1', g_form.source_id);
twoUsers.addParam('sysparm_user_id2', g_form.target_id);
twoUsers.getXML(userParse);

function userParse(response) {
    var answerFromXML = response.responseXML.documentElement.getAttribute("answer");
    var user = answerFromXML.split(':');
    var confMerge = confirm('Merge ' + user[0] + ' --> ' + user[1] + '\nAre you sure?');

    if(confMerge){
        <additional script logic would go here...
    }
}
```

The client-side code shown here would be wrapped in a function (*onLoad()*, *onSubmit()*, or *onChange()*) depending on the type of client-side script.

In this example, the *answerFromXML* variable from the response contains multiple pieces of information separated by a colon. The statement *var user = answerFromXML.split(':')* breaks the answer variable into an array called *user* with each element being whatever is between each set of colons. In this case, *user[0]* contains the value from the beginning of the answer string until the first colon, and *user[1]* contains the value from the colon to the end of string. Assume *answer=Matt:Miranda*. In this case *user[0] = "Matt"* and *user[1] = "Miranda"*.

Module Labs

- **Lab 9.3**
 - **Time:** 10-15m
 - HelloWorld GlideAjax
 - Create a Client Script that calls a Script Include
- **Lab 9.4**
 - **Time:** 20-25m
 - Number of Group Members
 - Create a Client Script and Script Include that returns information to the client



HelloWorld GlideAjax

Lab
09.03

⌚10-15m

Lab Summary

You will achieve the following:

- Create a Script Include which extends the AbstractAjaxProcessor class.
- Create a Client Script to call the Script Include.

A. Create a Script Include

1. Create a new Script Include.

Name: **HelloWorld**

Client Callable: **Selected (checked)**

Accessible from: **All application scopes**

Active: **Selected (checked)**

Description: **Lab 9.3 Extends the AbstractAjaxProcessor class**

2. Examine the pseudo-code for the script:

- Create a new class called HelloWorld.
- Create an object from the new class with properties inheritable by other objects and which extends the AbstractAjaxProcessor class.
 - Add a method to the new object called alertGreeting.
 - Return "Hello" + parameter containing user's name + "!".

3. Write the Script:

```
var HelloWorld = Class.create();
HelloWorld.prototype = Object.extendsObject(AbstractAjaxProcessor, {

    alertGreeting: function(){
        return "Hello " + this.getParameter('sysparm_user_name') + "!";
    },

    type: 'HelloWorld'
});
```

4. Select **Submit**.

B. Call the Script Include from a Client Script

1. Create a new Client Script.

Name: **Lab 9.3 HelloWorld Client Script**

Table: **Incident [incident]**

Type: **onLoad**

Active: **Selected (checked)**

Global: **Selected (checked)**

2. Examine the pseudo-code for the script:

- Create an instance of the GlideAjax object called HelloWorld.
- Add a param to call the alertGreeting method.
- Add a param to pass the user name Ruth.
- Use the getXML method and the callback function HelloWorldParse to execute the Script Include.
- Pass the response returned from the Script Include to the callback function.
- Locate the answer variable in the returned XML and store the value in a variable.
 - Generate an alert to display the value of the answerFromXML variable.

3. Write the script:

```
function onLoad() {  
    var ga = new GlideAjax('HelloWorld');  
    ga.addParam('sysparm_name','alertGreeting');  
    ga.addParam('sysparm_user_name','<YOUR NAME>');  
    ga.getXML(HelloWorldParse);  
  
    function HelloWorldParse(response) {  
        var answerFromXML = response.responseXML.documentElement.getAttribute('answer');  
        alert(answerFromXML);  
    }  
}
```

4. Select **Submit**.

5. The first parameter in the script you just wrote is **sysparm_name**, it is a reserved parameter name. What information does it pass to the Script Include?

-
6. Other than beginning with a **sysparm_** prefix, do additional parameters have to use reserved parameter names?
-

C. Test Your Work

1. Open any Incident.
2. Did an alert appear? Was it populated correctly? If not, debug and re-test.
3. Make the *Lab 9.3 HelloWorld Client Script* **inactive**.

Lab Completion

Great job! You have successfully extended the existing AbstractAjaxProcessor class. The alertGreeting() method you created in the HelloWorld Script Include expects a value to be passed in when it is called. It then uses that value to build a string and return it to the client.

You then used the GlideAjax class in a Client Script to call the alertGreeting() method in the new Script Include.

Number of Group Members

Lab
09.04

⌚20-25m

Lab Summary

You will achieve the following:

- Create a Script Include which:
 - Extends the AbstractAjaxProcessor
 - Return the number of members of a group.
- Create a Client Script that will:
 - Call a Script Include.
 - Pass a Group name to the Script Include.
 - Parse the response from the Script Include.

A. Create a Script Include

1. Create a new Script Include.

Name: **AssignmentGroupUtils**

Client Callable: **Selected (checked)**

Accessible from: **All application scopes**

Active: **Selected (checked)**

Description: **Lab 9.4 extends the AbstractAjaxProcessor class. Returns the number of members in a Group.**

2. Examine the pseudo-code for the script:
 - Create a new class called AssignmentGroupUtils.
 - Create an object from the new class with properties inheritable by other objects and which extends the AbstractAjaxProcessor class.
 - Add a method to the new object called countGroupMembers.
 - Create a new variable to store the group name.
 - Store the message the group has no members in the variable message.
 - Store the sys_id of a group from a Client Script in the variable groupID.
 - Create a new GlideRecord object for the sys_user_grmember table.
 - Query the sys_user_grmember table to return records for all members of the group passed in from a Client Script.
 - If there are records returned
 - Store the name of the group in the variable grpName.
 - If the value of grpName is not empty
 - Overwrite the current value in the variable message with a string containing the name of the group and number of group members in the variable message
 - Return the variable message to the calling Client Script.

3. Write the script:

```
var AssignmentGroupUtils = Class.create();
AssignmentGroupUtils.prototype = Object.extendsObject(AbstractAjaxProcessor, {
    countGroupMembers: function() {
        var grpName = "";
        var message = "There are no members in this Assignment Group";
        var groupID = this.getParameter('sysparm_group_id');

        var grpMems = new GlideRecord('sys_user_grmember');
        grpMems.addQuery('group.sys_id', groupID);
        grpMems.query();

        if(grpMems.next()){
            grpName = grpMems.getDisplayValue('group');
        }

        if(grpName != "") {
            message = "There are " + grpMems.getRowCount() + " members in
the " + grpName + " group";
        }

        return message;
    },
    type: 'AssignmentGroupUtils'
});
```

4. Select **Submit**.

B. Call the Script Include from a Client Script

1. Create a new Client Script.

Name: **Lab 9.4 Number of Group Members**

Table: **Incident [incident]**

Type: **onChange**

Field name: **Assignment group**

Active: **Selected (checked)**

Global: **Selected (checked)**

2. Examine the pseudo-code for the script:

- Create an instance of the GlideAjax object called AssignmentGroupUtils.
- Add a param to call the countGroupmembers method.
- Add a param to pass the sys_id of the Assignment group value to the Script Include.
- Use the getXML method and the callback function membersParse to execute the Script Include.
- Pass the response returned from the Script Include to the callback function.
 - Locate the answer variable in the returned XML and store the value in a variable.
 - Generate an alert to display the value of the answer variable.

3. Write the script:

```
function onChange(control, oldValue, newValue, isLoading, isTemplate) {  
    if (isLoading || newValue === '') {  
        return;  
    }  
  
    var membersGA = new GlideAjax('AssignmentGroupUtils');  
    membersGA.addParam('sysparm_name','countGroupMembers');  
    membersGA.addParam('sysparm_group_id',g_form.getValue('assignment_group'));  
    membersGA.getXML(membersParse);  
  
    function membersParse(response) {  
        var myObj = response.responseXML.documentElement.getAttribute('answer');  
        alert(myObj);  
    }  
}
```

4. Select **Submit**.

C. Test Your Work

1. Navigate to **User Administration > Groups**.
2. Select a group with members assigned to it. Record the group you selected here:

3. Open any Incident record.
4. Change the value in the **Assignment group** field to the group you selected in step 2.
5. Did an alert appear? Was it populated correctly? If not, debug and re-test.
6. Change the value in the Assignment group field to **IT Securities**. (The IT Securities group has no members baseline.)
7. Did an alert advise there are no members in this group? If not, debug and re-test.

D. Reuse the Script Include

1. Can you think of other tables that could use a similar Client Script to call the same Script Include? Write the name of two of those tables here:

2. Open the **Lab 9.4 Number of Group Members** Client Script.

3. Update the Client Script trigger:

Name: **Lab 9.4 Number of Group Members (PRB)**
Table: **Problem [problem]**

4. Select **Insert** on the form's Context menu.
5. Open any Problem record.
6. Change the value in the **Assignment group** field to the group you selected earlier in the lab.

7. Did the alert appear? Was it populated correctly? If not, debug and re-test.
8. Make both the Lab 9.4 Number of Group Members and Lab 9.4 and Number of Group Members (PRB) Client Scripts **inactive**.

Lab Completion

You have completed the lab and successfully extended the existing AbstractAjaxProcessor class. The new countGroupMembers() method you created in the AssignmentGroupUtils Script Include uses the passed in Assignment group's sys_id in a GlideRecord query to count the members of that Assignment group and return the information to the client.

You used the GlideAjax class in a Client Script to call the new method and use the value returned from the server in an alert to inform the user how many members are in their selected group.

JSON: JSON Script Include

```

var CallerIncidents = Class.create();
CallerIncidents.prototype = Object.extendsObject(AbstractAjaxProcessor, {
    getCallersIncs: function(){
        var incArray= []; //Create an empty array
        var callersRecords = new GlideRecord('incident');
        callersRecords.addQuery('caller_id',this.getParameter('sysparm_callerID'));
        callersRecords.query();
        while(callersRecords.next()) {
            //Build the array of objects
            var incDetails = {};
            incDetails.number = callersRecords.number.toString();
            incDetails.priority = callersRecords.priority.getDisplayValue();
            incDetails.shortDesc = callersRecords.short_description.toString();
            incArray.push(incDetails);
        }
        //Create a string from a JSON object and pass it back to the calling script
        return JSON.stringify(incArray);
    },
    type: 'CallerIncidents'
});

```

In this example, when the Script Include is called, the script first creates an empty array called **incArray**.

It then uses the passed in parameter **sysparm_callerID** in a GlideRecord query to gather all of the user's Incident records.

The while loop builds an **incDetails** object using properties/values from the GlideRecord query and puts that object into the **incArray** array.

The **JSON.stringify()** method is used to create a string from a JSON object (*incArray*) that is returned to the calling Client Script.

JSON: What is Returned?

now.

- JSON object is contained in an XML wrapper
- Returns the answer variable and administrative information

```
<xml answer='[{"number": "INC0000014", "priority": "2 - High", "shortDesc": "missing my home directory"}, {"number": "INC0000016", "priority": "4 - Low", "shortDesc": "Rain is leaking on main DNS Server"}, {"number": "INC0000038", "priority": "4 - Low", "shortDesc": "my PDF docs are all locked from editing"}, {"number": "INC0000041", "priority": "4 - Low", "shortDesc": "My cubicle phone does not work"}]'>  
sysparm_max="15" sysparm_name="callerIncidents"  
sysparm_processor="CallerIncidents"></xml>
```

In the example shown, four objects are returned in the array sent back.

JSON: JSON Client-Side Script

now.

```
//Create a GlideAjax object and call the Script Include
var incGA = new GlideAjax('CallerIncidents');
incGA.addParam('sysparm_name', 'getCallersIncs');
incGA.addParam('sysparm_callerID', g_form.getValue('caller_id'));
incGA.getXMLAnswer(CallerIncidentsParse);

function CallerIncidentsParse(response){
    //Build an array of objects from the JSON object and take action
    var myObj = JSON.parse(response);
    g_form.addInfoMessage('User has raised the following incidents:');
    for (var i=0; i< myObj.length; i++){
        g_form.addInfoMessage(myObj[i].number + ':' + myObj[i].shortDesc);
    }
}
```

The client-side code shown here would be wrapped in a function (*onLoad()*, *onSubmit()*, or *onChange()*) depending on the type of client-side script.

The **JSON.parse()** method creates an object from a JSON formatted string.

Module Labs

now.

- **Lab 9.5**

- **Time:** 25-30m
- JSON Object
 - Create a Client Script that calls a Script Include that will return a JSON Object



JSON Object

Lab
09.05

25-30m

Lab Summary

You will achieve the following:

- Use the Script Include and a copy of the Client Script you wrote in the previous lab.
- Modify the Script Include to return a JSON object containing an array of Group members
- Modify the Client Script to dynamically select a group member from the JSON object to assign to an Incident.
- Complete a Cloud Dimensions' Phase II requirement.



Business Problem

A few IT analysts are reporting the same individuals are continually being assigned Incidents on their teams. They would like to ensure support responsibilities are spread across the team, as everyone is busy working on other items in addition to their support responsibilities.

After some discussions, management has agreed to implement a strategy that will randomly assign analysts to Incidents, ensuring everyone has a turn.

Project Requirement

To implement this strategy, two scripts will be required:

1. First, create a Script Include to randomly select an analyst from the identified Assignment group, and then return that name to the calling Client Script.

2. Create a Client Script that calls the Script Include method whenever the value in the Assignment group changes. The name of the randomly selected analyst is returned from the server, should then populate the Assigned to field.

Lab Dependency: This lab builds on the scripts created in **Lab 9.4 Number of Group Members**. Please complete the scripts in that lab before proceeding.

A. Add a New Method to an Existing Script Include

1. Open the **AssignmentGroupUtils** Script Include.
2. Place your cursor after the comma at the end of the countGroupMembers function. Press **Enter <return>** on your key board two times to create some blank lines. (*The blank lines will leave some space between functions for easy readability.*)

```
16 v      if(grpName != "") {  
17          message = "There are " + grpMems.getRowCount() + " members in the "  
+ grpName + " group";  
18          }  
19          return message;  
20      },  
21  
22      | ←  
23      type: 'AssignmentGroupUtils'  
24      );  
25  
26      27
```

3. Examine the pseudo-code for the new method you will ADD to the Script Include:
 - Add a method to the object called assignAnalyst.
 - Store the sys_id of a group from a Client Script in the variable groupID.
 - Create a new array.
 - Query the sys_user_grmember table to return records for all members of the group passed in from a Client Script.
 - While there are returned records
 - Create a new object to store member information
 - Add the sys_id for the member to the new object
 - Add the Display Value for the member to the new object
 - Push the object into the array
 - Return the array of members as a JSON object.

4. Write the highlighted script after the blank lines you inserted in step 2.

```
var AssignmentGroupUtils = Class.create();
AssignmentGroupUtils.prototype = Object.extend(AbstractAjaxProcessor, {
    countGroupMembers: function() {
        var grpName = "";
        var message = "There are no members in this Assignment Group";
        var groupID = this.getParameter('sysparm_group_id');

        var grpMems = new GlideRecord('sys_user_grmember');
        grpMems.addQuery('group.sys_id', groupID);
        grpMems.query();

        if(grpMems.next()){
            grpName = grpMems.getDisplayValue('group');
        }

        if(grpName != "") {
            message = "There are " + grpMems.getRowCount() + " members in
the " + grpName + " group";
        }

        return message;
    },
    assignAnalyst: function(){
        var groupID = this.getParameter('sysparm_group_id');
        var membersArray = [];

        var membersGR = new GlideRecord('sys_user_grmember');
        membersGR.addQuery('group.sys_id', groupID);
        membersGR.query();

        while(membersGR.next()) {
            var member = {};
            member.sys_id = membersGR.user.sys_id.toString();
            member.name = membersGR.user.getDisplayValue();
            membersArray.push(member);
        }

        return JSON.stringify(membersArray);
    },
    type: 'AssignmentGroupUtils'
});
```

5. Select **Update**.

B. Call the Script Include from a Client Script

1. Open the **Lab 9.4 Number of Group Members** Client Script.
2. Select **Insert and Stay** on the form's Context menu to make a copy of the record and remain on the form.
3. Re-configure the new Client Script trigger:

Name: **Lab 9.5 Assign Analyst**
Active: **Selected (checked)**
4. Examine the pseudo-code for the Client Script:
 - Create an instance of the GlideAjax object called AssignmentGroupUtils.
 - Add a param to call the assignAnalyst method.
 - Add a param to pass the sys_id of the Assignment group value to the Script Include.
 - Use the getXML method and a callback function in your Client Script to execute the Script Include.
 - Pass the response returned from the Script Include to the callback function
 - Locate the answer variable in the returned XML and store the value in a variable
 - Create an object from the JSON formatted string and store it in the variable named members
 - Find the length of the array
 - If the array has elements
 - Generate a random number between zero and one less than the number of array elements
 - Set the assigned_to field to the array element selected by the random number
 - Else
 - Clear any value currently in the assigned_to field
 - Generate an alert that says the selected Assignment group has no members

5. Much of the code for this Client Script already exists from the previous lab. Using the image on the next page as a guide, make the following modifications:

- a) Modify the statement calling the method name to call the **assignAnalyst** method.
- b) In the callback function, immediately after the var myObj = response.responseXML... statement
 - i. Delete the alert(myObj); statement.
 - ii. Use the JSON.parse() method to create an object from the returned JSON formatted string.
 - iii. Add an 'if' statement to check if the array has elements. Generate a random number and make the assignment.
 - iv. Add an else statement to clear the assigned_to field and advise if the group has no members.

```
function onChange(control, oldValue, newValue, isLoading, isTemplate) {  
    if (isLoading || newValue === '') {  
        return;  
    }  
  
    var membersGA = new GlideAjax('AssignmentGroupUtils');  
    membersGA.addParam('sysparm_name','assignAnalyst');  
    membersGA.addParam('sysparm_group_id',g_form.getValue('assignment_group'));  
    membersGA.getXML(membersParse);  
  
    function membersParse(response) {  
        var myObj = response.responseXML.documentElement.getAttribute('answer');  
        var members = JSON.parse(myObj);  
  
        if(members.length > 0) {  
            var randomNum = Math.floor(Math.random() * members.length);  
            g_form.setValue('assigned_to', members[randomNum].sys_id,  
members[randomNum].name);  
        }  
        else {  
            g_form.setValue('assigned_to', "");  
            alert("The assignment group has no users assigned to it");  
        }  
    }  
}
```

6. Select **Update**.

C. Test Your Work

1. Open any Incident record.
2. Examine the value of the **Assigned to** field. Record the value here:

3. Was the **Assigned to** field populated with the name of an Assignee from that group? If not, debug and re-test.
4. Record the name of the analyst currently in the **Assigned to** field here:

5. Change the value in the Assignment group field to **IT Securities**. (*The IT Securities group has no members baseline.*)
6. Did the value in the **Assigned to** field clear? If not, debug and re-test.
7. Did an alert advise there are no members in this group? If not, debug and re-test.
8. Make the *Lab 9.5 Assign Analyst Client Script* for this lab **inactive**.

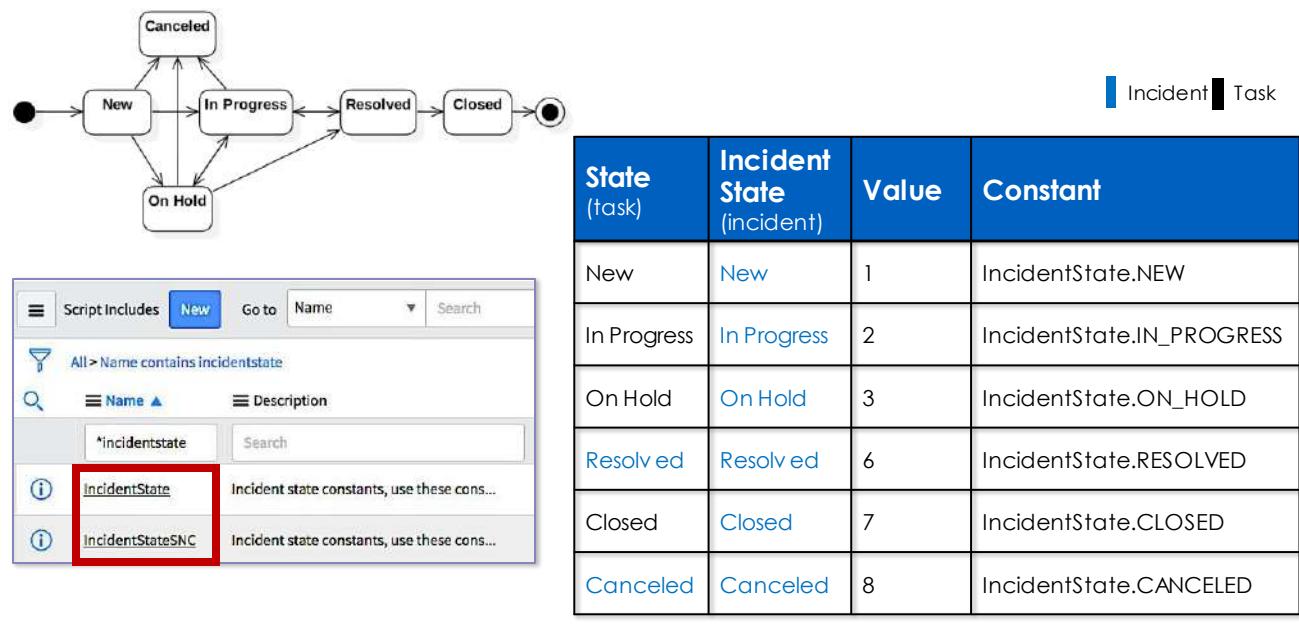
Lab Completion

Fantastic! You have successfully added a second method to an existing Script Include. The assignAnalyst() method uses the passed in Assignment group's sys_id in a GlideRecord query to build an array containing all the sys_ids of the group members. It then returns the object to the client.

You used the GlideAjax class in a Client Script to call the new method and use the returned object to randomly assign incidents to group members ensuring the group's work is shared fairly.

State Model: Baseline Script Includes and the State Model

now.



The **IncidentStateSNC** and **IncidentState** Script Includes provide State constants for use in scripts.

- Script Includes with **SNC** in the Name are meant to be read-only. (*This ensures the SNC Script Includes are updated during upgrades.*)
- To impose pre-requisites or limits for moving from one state to another, you can incorporate new logic in the **IncidentState** Script Include.
- To override a function defined in an **SNC** Script Include, copy that function to the paired version of the Script Include that does NOT contain SNC. Paste and customize the function there.

It is considered best practice to refer to States in your scripts using constants
`current.state.changesTo(IncidentState.CLOSED)` rather than values
`current.state.changesTo(7)`.

State Models currently exist for the **Incident** and **Change** applications. Visit docs.servicenow.com and search for the following articles for more information on the State Model.

- State model mapping
- Add a state to the state model
- Scripts modified with Incident Management – Core
- Script includes installed with Incident Management – Core
- Installed with Change Management – State Model
- Update change request states

Server Lookup Best Practices: Minimize Server Lookups

now.

- You have reviewed many ways to get information from the server in class
- The recommended methods are
 - **g_scratchpad**
 - Sent once when the form loads
 - **GlideAjax**
 - Dynamically triggered when the client requests information from the server



As you learned in previous modules, **GlideRecord** and **g_form.getReference()** are also available for retrieving server information. They have a slightly higher performance impact. Both methods **retrieve all fields** in the requested GlideRecord when most cases only require one field.

Script Includes: Topics

now.

Script Includes Overview

Store One Classless Function

Define a New Class

Extend an Existing Class

Application Scope

Application Scope: Scope Matters in a Script Include

now.

- Identify if the Script Include is:
 - **Public** and accessible from **All application scopes**
 - **Private** and accessible to artifacts in **This application scope only**



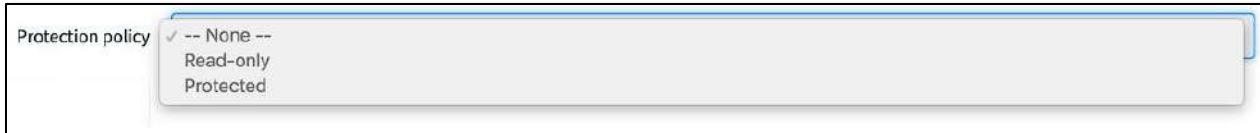
The screenshot shows the 'Script Include' configuration page. The 'Name' field contains 'HelloWorld'. The 'API Name' field contains 'global.HelloWorld'. The 'Client callable' checkbox is unchecked. The 'Application' dropdown is set to 'Global'. The 'Accessible from' dropdown is open, showing two options: 'All application scopes' and 'This application scope only'. The 'This application scope only' option is selected and highlighted with a blue border. The 'Active' checkbox is checked.

Use the **Accessible from** field to make a Script Include accessible from all scopes.

Application Scope: Protection Policy

now.

- Only applies when applications are installed from the ServiceNow App Store
- Sets the level of protection for the application file
- Protection policies safeguard intellectual property by making the file **read-only** or **Protected** (*not visible*)



Protection Policy options include:

- **None** – Script Include logic is viewable and editable by anyone.
- **Read-only** – Script Include logic is viewable but not editable.
- **Protected** – Script Include logic is not viewable.

Protection policies do not prevent other developers on the application development instance from viewing or editing a Script Include.

Protection policies are not applied when applications are published and migrated to an instance using an Update Set.

For the instance on which an application is developed, use Access Controls and/or roles to restrict User ability to see and edit Script Includes. Protection policies do not apply in this case.

Application Scope: Calling a Script Include

now.

- The syntax to call a Script Include depends on scope

- A scope prefix is not required to call a Script Include in the same scope

```
var conflict = new ItineraryConflict();
```

- To call a Script Include in another scope, prepend the Script Includes name with its unique scope namespace to distinguish it from other Script Includes with the same name

```
var travSched = new x_cld_travel.ItineraryConflict();
```

```
var switches = new global.ArraysUtils();
```

Every custom application has an application scope that is defined by its namespace.

Application Scope: Script Include Scope

now.

Effects of making a Script Include **public** or **private**

| | |
|---|---|
| <p>Application: Travel Management Scope: x_cld_travel</p> <p>Script Include</p> <p>Name: VendorUtils Application: Travel Management Accessible from: All application scopes</p>  | <p>Application: Loaner Equipment Scope: x_cld_loaner</p> <p>Business Rule</p> <p>Name: ReserveProjector Application: Loaner Equipment</p> <p>Script Include use allowed</p> <pre>if (current.state == 7) { //Out for repair var vLookup = new x_cld_travel.VendorUtils(); vLookup.confirmVendor(); <additional logic...></pre> <p>Script Include use rejected</p> <pre>if (current.type == 'purchase_new') { var reimburse = new x_cld_travel.ExpenseUtils(); reimburse.submitExpense(); <additional logic...></pre>   |
|---|---|

In the example on the left, two Script Includes are developed for the Travel Management application.

- **VendorUtils** – methods for managing vendors, set to be accessible from all application scopes.
- **ExpenseUtils** – methods for managing and processing travel expenses, set to be accessible only to other scripts in the Travel Management scope.

On the right, a Business Rule named ReserveProjector has been developed for a Loaner Equipment application:

- The Business Rule script instantiates the *VendorUtils* Script Include in the Travel Management scope to access its *confirmVendor* method. As this Script Includes' *Accessible from* value is set to **All application scopes**, the Business Rule is permitted to use the method.
- Later in the Business Rule script, there is an 'if' statement to determine if the hardware loaned to employees needs to be replaced with new equipment. The Business Rule instantiates the *ExpenseUtils* Script Include to access its *submitExpense* method. As this Script Includes' *Accessible from* value is set to **This application scope only**, access is denied.

Script Includes: Cloud Dimensions Requirements

now.

User Interface Requirements

- | | |
|--|---|
| 1 Confirm Major Incident process is followed before a P1 is submitted | ✓ |
| 2 Enforce mandatory Incident requirements if State is Resolved or Closed | ✓ |

Database Requirements

- | | |
|--|---|
| 3 Identify Incident records with RCA documentation | ✓ |
| 4 Populate Change's CAB date field with next CAB meeting date | ✓ |
| 5 Prevent Problems from re-opening if closed for more than 30 days | ✓ |
| 6 Update Problem and Child Incidents with RCA details from parent Incident | ✓ |
| 7 Implement SLA targets and identify Incidents in danger of breaching them | ✓ |
| 8 Automatically assign Incidents to Assignment group members | ✓ |

Security Requirements

- | | |
|------------------------|---|
| 9 Track Impersonations | ✓ |
|------------------------|---|

Lab 9.4 fulfills requirement 8.

Script Includes

now.

Good Practices

- Prepend the Script Include name with the unique scope namespace when calling Script Includes from different scopes
- GlideAjax can be called any time from a client-side script
- When passing multiple pieces of data from the server back to the client
 - Use character delimited strings
 - Use JSON objects
- Create a debugging Script Include on your instance to enable/disable debugging by setting a flag

Module Recap: Script Includes

now.

| Core Concepts | Real World Cases |
|--|--|
| Script Includes contain reusable code | • Why would you use these capabilities? |
| Script Includes execute only when called | • When would you use these capabilities? |
| Classless Script Includes are callable only from server-side scripts | • How often would you use these capabilities? |
| Use GlideAjax when passing data between the client and server | |
| JSON objects pass data as arrays or objects | |
| Privately scoped Script Includes can only be called by other scripts in the same scope | |

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Module 10: UI Actions

now.

| |
|--------------------|
| Scripting Overview |
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Describe and use Client UI Actions
- Describe and use Server UI Actions
- Write, test, and debug UI Action Scripts

Labs

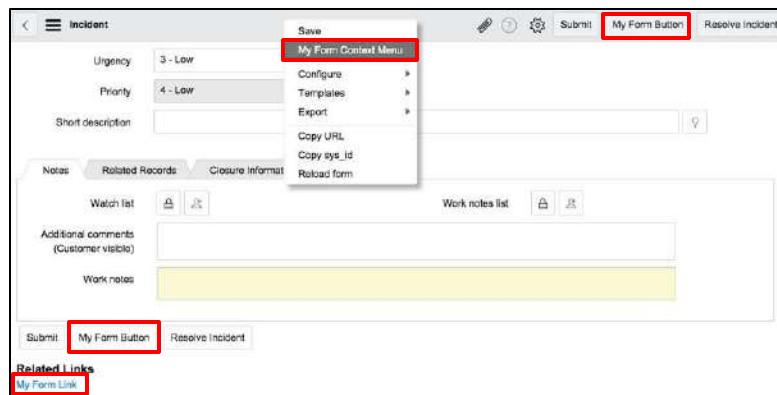
- Lab 10.1 Client UI Action – Priority 1 Incident
- Lab 10.2 Server UI Action – URL Redirect
- Lab 10.3 Client and Server UI Action – Update Problem

In this module you will practice writing, testing, and debugging UI Actions.

What are Form UI Actions?

Add to forms:

- Form buttons
- Form context menu items (right-click in the header)
- Form links (Related links section)



UI Actions put buttons, links, and context menu items on forms and lists making the UI more interactive, customizable, specific to user activities, and applicable to business requirements. Scripts take action when buttons, links, or context menu items are selected.

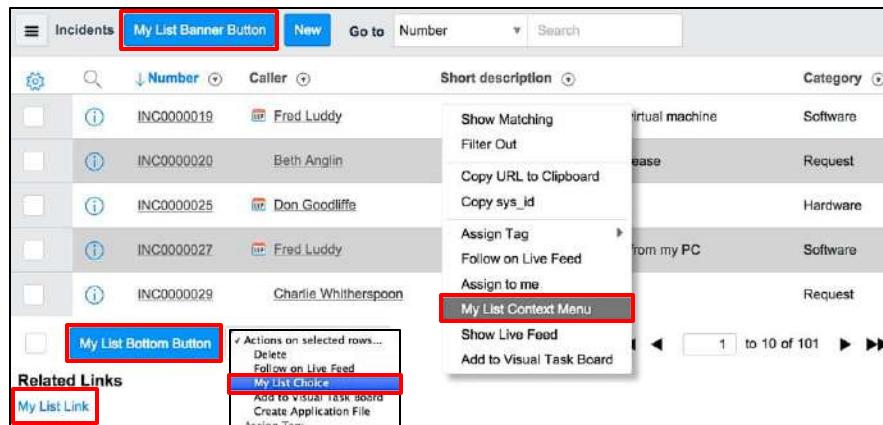
To prevent a button from appearing at the bottom of a form, set the `glide.ui.buttons_bottom` system property to **false**.

What are List UI Actions?

now.

Add to lists:

- List bottom button
- List banner button
- List context menu items (right-click a record)
- List choices (at the bottom of list)
- List links (Related Links at the bottom of list)



Add UI actions to forms and lists in the same scope, as well as to forms and lists that allow UI Actions from another scope to run on them.

FAQ's

What Exists Baseline?

Over 1300 UI Actions exist baseline

When does this execute?

Run whenever UI Action trigger conditions met

Where can this be found?

Navigate to the **System Definition > UI Actions** module to create or modify UI Actions.



IMPORTANT

If you think you need to modify a baseline records, be sure to think about the following, ask yourself a few questions, and act accordingly.

- Carefully consider the customization
- Will no code changes work?
- Is customization necessary?
- Modify the baseline record
- Review and revert the skipped file after an upgrade

UI Actions Trigger

now.

- Determines when UI Action runs
- Specifies which buttons, links, and context menu items to display

The screenshot shows the 'UI Action' configuration page. At the top, it says 'New record'. On the left, there's a list of settings:

- Name: Mobile Phone Tech Support
- Table: Incident [incident]
- Order: 100
- Action name: techSupportURL
- Active: checked
- Show insert: checked
- Show update: checked
- Client: checked
- List v2 Compatible: checked
- List v3 Compatible: unchecked
- Overrides: (empty text field)

On the right, there are several dropdown menus and checkboxes:

- Application: Global
- Form button: checked
- Form context menu: unchecked
- Form link: unchecked
- Form style: -- None --
- List banner button: unchecked
- List bottom button: unchecked
- List context menu: unchecked
- List choice: unchecked
- List link: unchecked
- List style: -- None --

A 'Submit' button is at the top right.

Name – the text that appears on the widget.

Table – table for which the UI Action applies. Global is for all tables.

Order – the order of display of the UI Action. Lower numbers are higher on the list/farthest to the left.

Action name – name by which this UI Action can be called in a script.

Active - select this option to enable the UI Action.

Show insert/update – displays the widget on new records/existing records.

Client – UI Action executes in the browser.

List v2 Compatible - Specifies that the UI action is can be used with v2 lists and will display on v2 List.

List v3 Compatible - Specifies that the UI action is can be used with v3 lists and will display on v3 List.

Application – the scope containing the UI Action.

Form button/context menu/link – if selected enables the widget on a form.

List bottom button/banner button/context menu/choice/link – if selected enables the widget on a list form.

Show multiple update – (does not appear baseline and must be added through personalization): Select this option to display the widget on multiple record update forms.

Show query – (does not appear baseline and must be added through personalization): Select this option to display the widget on Search forms.

UI Actions Trigger

now.

- The Condition test executes on the server even if the UI Action runs client-side
- The Onclick field appears only when the Client option is selected

| | |
|-----------|---|
| Comments | Adds a link and a button to the Incident form for the Mobile Phone Technical Support web site if the CI on an Incident form is a Mobile Phone |
| Hint | Click here for Mobile Phone Technical Support web site |
| Onclick | <code>addTsUrl();</code> |
| Condition | <code>current.cmdb_ci.name == "Mobile Phone"</code> |

Comments – documentation about what a UI Action does, who wrote it etc.

Hint – text that appears when a user hovers over the widget.

Onclick – function to call when widget is selected.

Condition – restricts when a UI Action applies. Has access to current.

UI Actions Script

now.

- Usable syntax depends on where the script runs
 - Client-side
 - Server-side
- Runs when
 - Trigger criteria is met AND
 - Condition field returns true



The screenshot shows a ServiceNow UI Actions script editor window. The title bar says "Script". Below it is a toolbar with various icons. The main area contains a code editor with the following content:

```
1 v  function addTsUrl() {  
2 v    try {  
3       var myURL = "http://www.mobilephone.com/support/";  
4       window.open(myUrl);  
5     }  
6 v     catch (err) {  
7       jslog("An error occurred at runtime: " + err);  
8     }  
9   }
```

If the condition field has no value, the script will execute whenever the trigger criteria is met.

The example shown works with a condition to offer a user the opportunity to launch the RIM Tech Support website in a new browser window if an Incident's CI is Blackberry (see preceding slides for trigger and condition).

Protection Policy

now.

- Protection policies safeguard intellectual property by making UI Action logic read-only or not visible
- Only applies when applications are installed from the ServiceNow App Store
- Protection policies do not prevent other developers on the application development instance from viewing or editing a UI Action



Protection policies are not applied when applications are published and migrated to an instance using an Update Set.

The Protection Policy options are:

None – UI Action logic is viewable and editable.

Read-only – UI Action logic is viewable but not editable.

Protected – UI Action logic is not viewable.

For the instance on which an application is developed, use Access Controls and/or roles to restrict User ability to see and edit UI Actions. Protection policies do not apply in this case.

Controlling Access to buttons, links, and context menu items

now.

A button should not be available at all times for all users

- Restrict with List Control
- Restrict by View using UI Action Visibility
- Restrict with Condition field

| | Update | Resolve Incident | Delete |
|------------------|---------------------|---------------------------------|---------------------------------|
| Opened by | Joe Employee | <input type="button" value=""/> | <input type="button" value=""/> |
| Opened | 2015-04-17 16:07:12 | <input type="button" value=""/> | <input type="button" value=""/> |
| Contact type | Email | <input type="button" value=""/> | <input type="button" value=""/> |
| State | Active | <input type="button" value=""/> | <input type="button" value=""/> |
| Assignment group | Network | <input type="button" value=""/> | <input type="button" value=""/> |
| Assigned to | Howard Johnson | <input type="button" value=""/> | <input type="button" value=""/> |

| | Update | Close Incident | Delete |
|------------------|---------------------|---------------------------------|---------------------------------|
| Opened by | Joe Employee | <input type="button" value=""/> | <input type="button" value=""/> |
| Opened | 2015-04-17 16:07:12 | <input type="button" value=""/> | <input type="button" value=""/> |
| Contact type | Email | <input type="button" value=""/> | <input type="button" value=""/> |
| State | Resolved | <input type="button" value=""/> | <input type="button" value=""/> |
| Assignment group | Network | <input type="button" value=""/> | <input type="button" value=""/> |
| Assigned to | Howard Johnson | <input type="button" value=""/> | <input type="button" value=""/> |

The image shows two side-by-side incident forms. The left form has an 'Open' state, while the right form has a 'Resolved' state. Red arrows point from the 'Resolved' state text to the 'Close Incident' button in both forms, indicating that the button is only visible when the incident is resolved.

For example, a Close Incident button might be available only to group managers or should appear only on Incident forms if the state is resolved.

UI Action Visibility and Conditions

now.

- Use the UI Action Visibility option to restrict access based on View

The screenshot shows a list view titled "UI Action Visibility (1) Versions (1)". There is one item listed:

| UI Action Visibility | Visibility | View | Updated by |
|---------------------------|------------|--------------|------------|
| Mobile Phone Tech Support | Include | Self Service | admin |

At the bottom, there is a button labeled "Actions on selected rows..." with a dropdown arrow.

- Use the Condition field to restrict access based on scripted criteria

| | |
|-----------|--|
| Condition | current.state < 3 && current.approval != requested |
|-----------|--|

If the Condition field has no value the script will execute whenever the trigger criteria is met.

Client-Side UI Actions

now.

1. Select Client trigger option

The screenshot shows the 'Client-Side UI Actions' configuration page. The 'Client' checkbox is checked. The 'Onclick' field contains the function name 'addTsUrl();'. The 'Script' field contains the following JavaScript code:

```
1+ function addTsUrl() {
2+   try {
3+     var myURL = "http://www.mobilophone.com/support/";
4+     window.open(myURL);
5+   }
6+   catch (err) {
7+     jsllog("An error occurred at runtime: " + err);
8+   }
9+ }
```

2. Enter function name in Onclick field

3. Write function in Script field

Use client-side methods when scripting for execution within the browser. The script has no access to server-side objects, such as current, or methods, such as gs.log.

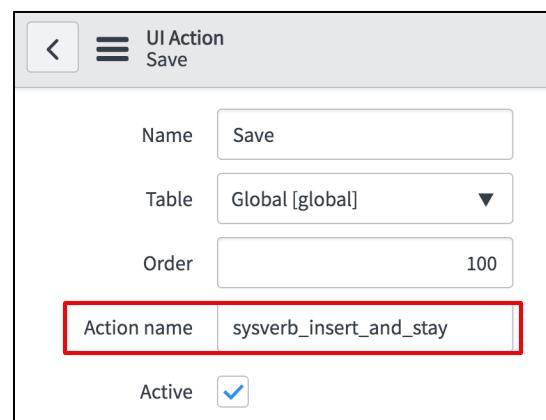
Common uses for client-side UI Actions are:

- Show workflow
- Launch email client
- Print preview
- Redirect to another URL

Calling another UI Action - Client

now.

- gsftSubmit method calls client-side UI Action
- gsftSubmit(gel('<UI Action Action Name>'))
- Example (inserts a new record into the database):
gsftSubmit(gel('sysverb_insert'))



Only UI Actions included on a form can be called. For example, the Edit Homepage UI Action's link does not appear on the Incident form and could not be called from a UI Action for the Incident form.

The gsftSubmit() method can also be called as follows:

```
gsftSubmit(null,g_form.getFormElement(),'<UI Action Action Name>');
```

The first parameter is the control (will have the value null when calling UI Actions), the second is the form, and the third is action name.

For example, gsftSubmit(null,g_form.getFormElement(),'sysverb_insert');

Server-Side UI Actions

now.

- Execute entirely on server-side
- Can use GlideSystem and GlideRecord methods
- Have access to current
- Server-side execution is the baseline behavior (Client not selected)

The screenshot shows the 'UI Action Insert' screen. The form fields are as follows:

- Name: Insert
- Table: Global [global]
- Order: 100
- Action name: sysverb_insert
- Active:
- Show insert:
- Show update:
- Client: (This field is highlighted with a red rectangular box.)

UI Actions run on the server-side when the Client option is not selected.

Use server-side UI Actions when:

- You do not need access to the form.
- You want improved performance.
- You do not need to call client-side UI Actions.
- You do not need access to form elements.
- You do not access to server-side methods.

Client and Server UI Actions

now.

Executes partially on client and partially on server

1. Write client code and call UI Action again

2. Ensure client code execution is complete and call the server-side code

3. Write server code

```
//Client-side Onclick function
function runClientCode() {
    //Client-side code here.
    //Call the UI Action again and skip
    //the Onclick function. MUST call
    //'Action name' set in this UI Action.
    gsftSubmit(null, g_form.getFormElement(),
        'button_action_name');
}

//Code that runs without Onclick.
//Ensures call to server-side
//function with no browser errors.
if(typeof window == 'undefined') {
    runServerSideFunction();
}

//Server-side function
function runServerSideFunction() {
    //Server-side code here
}
```

When the UI Action is called a second time the Onclick function will not execute because the button/link/menu item was not physically selected a second time. Any code not enclosed in a function is executed. In this case, the if(typeof window == 'undefined') statement runs and the server-side function is called.

Module Labs

- **Lab 10.1**

- **Time:** 15-20m
- Client UI Action – Priority 1 Incident
 - Create, test, and debug a UI Action

- **Lab 10.2**

- **Time:** 15-20m
- Server UI Action – URL Redirect
 - Create, test, and debug a server-side UI Action

- **Lab 10.3**

- **Time:** 15-20m
- Client and Server UI Action – Update Problem
 - Create, test, and debug a UI Action that executes on both the client and server-sides



Client UI Action – Priority 1 Incident

Lab
10.01
⌚15-20m

Lab Summary

You will achieve the following:

- Create a UI Action requiring confirmation before submitting a Priority-1 Incident.

A. Creating a UI Action

1. Open the UI Action list: **System Definition > UI Actions**.
2. **Create** a new UI Action.
3. **Configure** the UI Action trigger.

Name: **Priority 1**

Table: **Incident [incident]**

Active: **Selected (checked)**

Show insert: **Selected (checked)**

Show update: **Not Selected (unchecked)**

Client: **Selected (checked)**

Application: **Global**

Form button: **Selected (checked)**

Onclick: **confirmPriOne();**

Condition: **current.canCreate() && current.isNewRecord()**

4. Here is the pseudo-code for the script:
 - Get the priority from the Incident form field.
 - If the priority is not 1, insert the record into the database.
 - If the priority is 1, display a JavaScript confirmation dialog asking if the user really wants to submit the Incident with this priority.
 - If the answer is OK, insert the record into the database.

5. Write the UI Action script:

```
function confirmPriOne() {  
    var pri = g_form.getValue('priority');  
    if (pri != 1) {  
        gsftSubmit(null, g_form.getFormElement(), 'sysverb_insert');  
    }  
    else {  
        var confText = "Are you sure you want to submit a Priority 1 incident?";  
        if (confirm(confText)) {  
            gsftSubmit(null, g_form.getFormElement(), 'sysverb_insert');  
        }  
    }  
}
```

6. Save the UI Action.

B. Testing

1. Create a new Incident.
2. Set the Priority to Critical by setting the Urgency and Impact fields each to **High**.
3. Select the **Priority 1** form button.
4. Did the confirmation dialog box appear? If not, debug and re-test.
5. Select the **OK** button in the confirmation dialog box. Was the incident submitted? If not, debug and re-test.
6. Create another new Incident and set the Urgency and Impact to **High**.
7. Select the **Priority 1** button.
8. Select the **Cancel** button in the confirmation dialog box. Was the Incident submitted?
9. Make the Priority 1 UI Action **inactive**.

Lab Completion

Congratulations on completing the lab!

Server UI Action – URL Redirect

Lab
10.02
⌚15-20m

Lab Summary

You will achieve the following:

- Create a server-side UI Action accessible only from the Self Service view that allows a caller to verify their user information is correct.

A. Creating a UI Action

1. **Create** a new UI Action.
2. **Configure** the UI Action trigger.

Name: **Check User Info**
Table: **Incident [incident]**
Active: **Selected (checked)**
Show insert: **Selected (checked)**
Show update: **Selected (checked)**
Application: **Global**
Form button: **Selected (checked)**
Form link: **Selected (checked)**
Condition: **current.canCreate()**

3. Examine the pseudo-code for the script:
 - Add/modify the record in the database.
 - Try:
 - Save the user page URL in a variable.
 - Redirect to the URL+the caller's sys_id.
 - When finished with the user form return to the current Incident form.
 - Catch:
 - Log any errors to the Script Log Statements.

4. Write the UI Action script:

```
current.update();
try {
    userURL = "sys_user.do?sys_id=";
    action.setRedirectURL(userURL + current.caller_id);
    action.setReturnURL(current);
}
catch (err) {
    gs.error("<your initials>", err);
}
```

5. **Save** (not Submit) the UI Action.
6. Select the **Edit** button in the **UI Action Visibility** Related List at the bottom of the UI Action form.
7. Select the **Self Service** view from the Collection slushbucket and Add it to the **UI Action Visibility** list.
8. Select the **Save** button.

B. Testing

1. Create and Save a new Incident.
 2. Do the Check User Info button and link appear on the form? Should they? Explain your reasoning.
-
-

3. Navigate to **Self-Service > Incidents**, select the **New** button.
4. In the Incident form, set the Caller to **Olga Yarovenko**.
5. Select the **Check User Info** button or link.
6. Did Olga's user record open? If not, debug and re-test.

7. Change Olga's title:
 - a) Select the light bulb next to the Title field.
 - b) Select **Sales Executive** from the suggestions list.
 8. Select the **Update** button to update the User record.
 9. What page loads after the User record is updated? Is it the User list page? Why or why not? Explain your reasoning.
-
-

10. Make the Check User Info UI Action **inactive**.

Lab Completion

Congratulations on completing the Server UI Action – URL Redirect lab!

Client and Server UI Action – Update Problem

Lab
10.03
15-20m

Lab Summary

You will achieve the following:

- Create a UI Action that executes on both the client and server-sides, retrieves the problem_id from an Incident, and updates the related Problem's Work notes.

A. Creating a UI Action

1. Create a new UI Action.
2. Configure the UI Action trigger.

Name: **Update Problem**
Table: **Incident [incident]**
Action name: **update_problem**
Active: **Selected (checked)**
Show update: **Selected (checked)**
Client: **Selected (checked)**
Application: **Global**
Form button: **Selected (checked)**
Onclick: **updateProblem();**
Condition: **current.canCreate()**

3. Examine the pseudo-code for the script:
 - In the client-side function
 - Try:
 - If the Incident form does not have a value in the Parent field
 - Alert the user using the strategy of your choice
 - Else
 - Execute the UI Action a second time
 - Catch:
 - Log any errors to the JavaScript log
 - Check the Client Script has completed
 - Call the server-side function

- In the server-side function
 - Try:
 - Create the string you will put in the Work notes
 - Create a new GlideRecord for the problem table
 - Get the Parent (Problem Record) for the current Incident
 - Add a Work note to the Problem record
 - Update the Problem record
 - Catch:
 - Log any errors to the Script Log Statements
4. Write the UI Action script. If needed, refer to the "Client and Server UI Actions" slide's stub code.

```

function updateProblem() { //Client Side Function
    try {
        var problemID = g_form.getValue('problem_id');
        if (problemID == "") {
            alert('Problem ID is empty');
        } else {
            gsftSubmit(null, g_form.getFormElement(), 'update_problem'); //call Action Name
        }
    } catch (err) {
        jslog("Client <your initials: " + err);
    }
}

if(typeof window == 'undefined')
    serverSideFunction();

function serverSideFunction() { //Server Side Function
    current.update();
    try {
        var problemNumber = current.problem_id;
        var workNotesMessage = "<your initials> - Problem " + problemNumber.getDisplayValue() +
            " updated from Incident " + current.number + " UI Action";
        var gr = new GlideRecord("problem");
        gr.addQuery("sys_id", current.problem_id);
        gr.query();
        if (gr.next()) {
            gr.work_notes = workNotesMessage;
            gr.update();
        }
        gs.addInfoMessage(workNotesMessage);
    } catch (err) {
        gs.error("Server <your initials>",err);
    }
    action.setRedirectURL(current);
}

```

5. **Save** the UI Action.

B. Testing

1. Create or Open an Incident.
2. Located the Related Records Tab and Assign a Problem to the Problem ID field. Note the Problem number here:

3. Select the **Update Problem** button.
4. Open the Problem you select in step 2.
5. Examine the Problem's Activity history. Does your message appear in the history? If not, debug and re-test.
6. Make the Update Problem UI Action **inactive**.

Lab Completion

Congratulations on completing the lab!

UI Actions

now.

Good Practices

- Add buttons, links, and context menu items to forms and lists as appropriate
- Use appropriate widget type
 - Buttons are highly visible
 - Context menus are more subtle
- Call other UI Actions like `sysverb_insert_and_stay` rather than re-creating UI Action logic
- If a UI Action opens another page, use the `action.setRedirectURL()`
- To set which form loads when a UI Action completes, use `action.setReturnURL()`

Module Recap: UI Actions

now.

| Core Concepts | Real World Cases |
|---|---|
| <p>UI Actions add buttons, links, and context menu items to forms and lists</p> <ul style="list-style-type: none">• Client• Server | <p>• Why would you use these capabilities?</p> |
| <p>Use <code>gsftSubmit</code> to call a Server UI Action from a Client UI Action</p> | <p>• When would you use these capabilities?</p> |
| <p>UI Action Visibility restricts access based on views</p> <p>Protection policy options can protect privately scoped scripts installed from the ServiceNow App Store</p> | <p>• How often would you use these capabilities?</p> |

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Module 11: Flow Designer Scripting

now.

| |
|----------------------|
| Scripting Overview |
| Client Scripts |
| UI Policies |
| Catalog Scripting |
| Business Rules |
| GlideSystem |
| GlideRecord |
| Events |
| Script Includes |
| UI Actions |
| Flow Designer |

Module Objectives

- Discuss where to use scripts in Flow Designer
- Describe and use Flow Designer objects
- Write, test, and debug a script in Action Script Step
- Describe the role of Flow Designer Contexts in Flow Designer testing

Labs

- Lab 11.1 Build a Flow
- Lab 11.2 Trigger a Flow with a Script
- Lab 11.3 Add a Script to a Flow

Flow Designer Scripting: Topics

now.

Flow Designer Overview

Script-Triggered Subflow

Action Designer Script Step

Overview: Flow Designer Vs. Workflow

now.

Use Flow Designer When:

- Instance running Kingston or later version
- Process owners need to use natural language and low code way to automate approvals, tasks, notifications, and record operations
- New logic needs to be developed and it has not been created in Workflow
- Business logic needs to use the library of **reusable** actions across multiple flows.

Use Workflow When:

- Instance running Jakarta or earlier version
- Process requires complex flow logic or parallel processing not yet supported in Flow Designer
- Existing logic already developed using Workflow
- Steps required do not exist yet in Flow Designer and require unsupported protocols

Flow Designer and Workflow can be used at the same time in an instance. Understanding what process fits the needs of your company can aid you in a decision. As of the London release, below are some scenarios for Flow Designer or Workflow. Remember that even though it is suggested here to use one method or another, you may still be able to accomplish a task using Flow Designer or Workflow.

Flow Designer

Service Catalog Action Plan

- Approvals - Flow Designer has reimagined Approvals for a much more usable authoring experience. Flow Designer is a good fit for approval flow associated to a catalog item
- Flow with Stages to report progress status to user
- Standard task creation or record updates

Record updates

- When you need to interact with a record, or related set of records, and make updates. Flow Designer provides very intuitive ways to handle record updates, and working with related lists
- The new approval authoring method for Adding Approvals to a record (Service Catalog item, Incident, or any) in Flow Designer is much improved way of dealing with approvals

IntegrationHub

- Slack, Teams, Active Directory, Azure AD, and more... (More integrations in the future.)

Action Designer – alternative to Script Include

- When you want to build a reusable component(s) (build once, reuse in flows)

Workflow

- Complex branching and parallel processing – e.g. complex catalog workflows
- Service Level Management (SLA Timer)
- Passwords must be masked
- Existing orchestration integrations

Overview: What is Flow Designer?

now.

- A design environment that **Creates** flows
- Flows consist of a *trigger* and one or more actions
- Triggers start a flow
- Actions are steps that automate approvals, tasks, notifications, or record operations
- **Process owners and Developers** use natural language to create a flow

Triggers – What causes a flow to start

TRIGGER
Incident Created
Trigger: Created
Table: Incident [incident]
Condition: All of these conditions must be met
Priority is 1-Critical
Actions:
1. Create Freeform VTB
2. Look Up Visual Task Board Lane Record where (Board is 1-Board, and Name is To Do)
3. Create VTB Card

Actions – What Occurs when the flow starts

Flow Designer is a Now Platform feature that enables rich process automation capabilities in a consolidated design environment. It enables process owners to use natural language to automate approvals, tasks, notifications, and record operations without having to code.



IMPORTANT

Permission to use Flow Designer and view flow execution is controlled by assigning the `flow_designer` role to users. Before you give a process owner this role, **understand that you are giving the user access to all tables and database operations which is equal to an admin role.**

Overview: Flow Designer Landing Page

now.

- Navigate to **Flow Designer > Designer**
- Use the navigation links to work on each part of a flow

Landing Page Link

| Name | Internal name | Application | Status | Active | Updated | Updated by |
|------------------------------------|------------------------------------|-------------------------------|-----------|--------|---------------------|------------|
| Benchmark Recommendation Evaluator | benchmark_recommendation_evaluator | Benchmarks Spoke | Published | true | 2018-11-14 13:42:30 | system |
| VTB Sample Flow | vbt_sample_flow | Visual Task Board (VTB) Spoke | Draft | false | 2018-11-14 13:42:30 | system |

New Flow,
Subflow,
or
Action

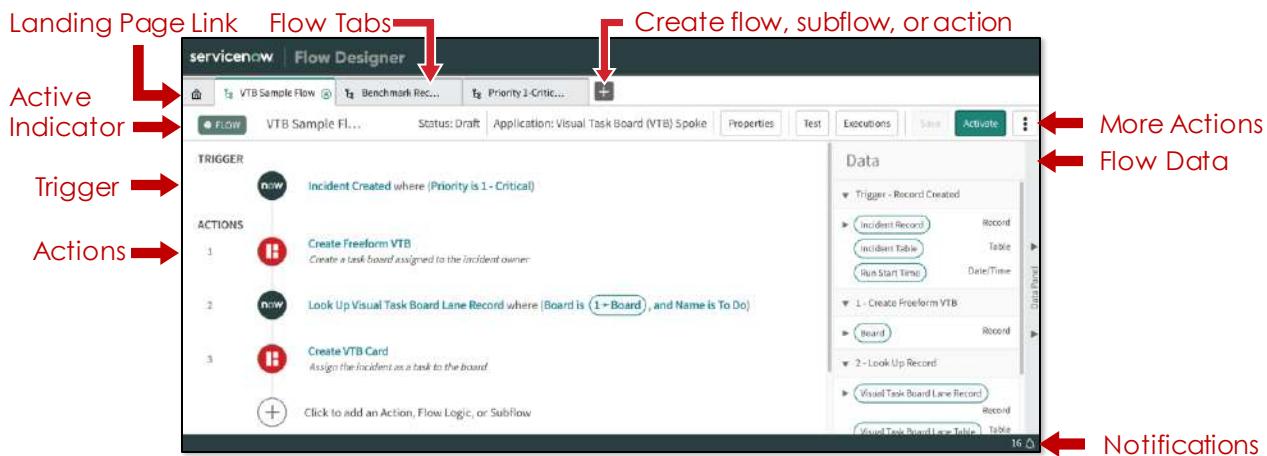
The welcome page has links to Flows, Subflows, Actions, Flow Executions, Help, and New.

- A **Flow** is an automated process consisting of a sequence of actions and a trigger. Flows automate business logic for a particular application or process.
- A **Subflow** is a sequence of reusable actions that can be started from a flow, subflow, or script.
- An **Action** is a reusable operation that enables process analysts to automate Now Platform features without having to write code.
- **Flow execution** is a list of associated execution detail pages. Each record is created when a flow runs. Process analysts can view the information the system stores about the configuration and runtime values produced.
- The **Help** tab has resource links to Documentation, Videos, and Community Discussion.
- The **New** link uses a dropdown to allow you to create a New Flow, New Subflow, or New Action.

Overview: Flow Designer Editor Page

now.

The flow editor uses **properties**, a **trigger**, a sequence of **actions**, and the **data** collected or created to create a flow.



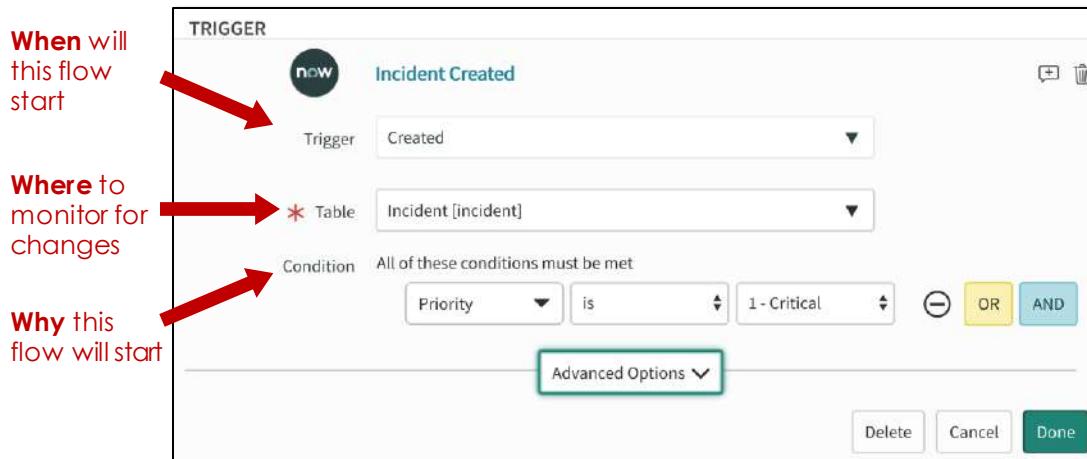
This example shows the Flow Editor after a Flow has been selected.

- **House Icon** - returns you to the Landing page.
- **Flow Tabs** – allows you to work on multiple flows; you can select the tab that you want to work on.
- **Plus Icon** – selecting the icon will allow you to create a new Flow, Subflow, or Action.
- **Indicator** – The text will tell you if you are working on Flow, Subflow, or Action and if the component of a flow is read only. Flows will have a gray (inactive) or green (active) dot.
- **Title** – Shows the flow you are currently working on.
- **Status** – Indicates if the flow is Draft.
- **Scope** – It is encouraged to use flow designer in a scoped application.
- **Properties** - Edit flow properties e.g. description and protection policy.
- **Test** - Simulate a trigger and execute the flow.
- **Executions** - List executions associated with this flow to open operations.
- **Save** - Save a draft of the Flow, Subflow, Action.
- **Activate** - Activate this flow and make the trigger live.
- **Three Dots** - open more actions that are specific to the flow, subflow, or action you are working with.
- **Trigger** - specifies the conditions that start the flow. When the trigger condition is true, the system starts the flow.
- **Actions** – The section groups sets of reusable business logic that produces a specific outcome when provided with its input values.
- **Flow Data** - any data gathered or generated as variables will be available in the data pane. Each variable has its own pill. The pill can be dragged and dropped on an action input or output.
- **Notifications** – Previous activity information.

Overview: Flow Designer Trigger

now.

The trigger specifies the conditions that start the flow. When the trigger condition is true, the system starts the flow.



Creating a trigger for a flow will require **When** the flow will start, **Where** this flow will monitor for changes, and **Why** this flow will start.

1. The **Trigger** dropdown has several Trigger Types to choose from.
 - **Record Trigger Type** - flows start when a record is newly created, updated, or both and if the transaction meets the condition filter. The **Updated** Trigger Type can run always or only once.
 - **Scheduled Trigger Type** starts a flow at specific time of the day, week, or month. Also, a flow can start at a specific time and it will **Run Once**. If you need to set a flow to run at predefined intervals (example: every 3 days), Use the **Repeat** choice.
 - **Application Trigger Type** has a MetricBase choice and a Service Catalog choice. MetricBase will start a flow by tracking time series data and can monitor when a threshold is reached, when a trend is detected, or when a system stops reporting data. Service Catalog Trigger Type monitors all catalog item requests for a specific item.

Unlike a Scheduled and Application Trigger Types, a Record Trigger Type requires:

1. **Table to monitor** - In this example we have chosen the incident table as the place to look at for conditional changes. Choose the appropriate table.
2. **Condition that starts the flow** - Use the condition builder the same way you filter for a list. All conditions created in the condition builder must be met.



TIP FROM THE FIELD

You may not be able to use Application Trigger Type. The MetricBase Trigger Type requires a separate subscription. On the Service Catalog Trigger Type a plugin (com.glideapp.servicecatalog.flow_designer) will need to be activated. Contact HI Customer Service for assistance.

Overview: Actions

now.

- A group of **reusable** operations that automate the Now Platform features without having to write code.
- Process analysts can use Actions, Flow Logic, or Subflows
 - Actions are used to add functionality to a flow.
 - Flow Logic controls whether or not actions run with **If** statements and **For Each** loops
 - Subflows automate generic business logic that can be applied to multiple applications or processes.



Actions can be anything from creating tasks, looking up records, or sending email. Flow designer has a number of core actions that can be used and custom actions that can be created.

- A core action is a ServiceNow-provided action available to any flow that cannot be viewed or edited from the Action Designer design environment.
- Custom actions can be viewed or edited from the Action Designer design environment. The custom actions always use the same configuration when added to a flow.

Flow Logic is a way to control what action steps are included or bypassed based on your criteria.

- The **If** conditional statements check if a condition is met before executing or skipping an action.
- The **For Each** Flow Logic loops through a list of records, supplied by flow data, to apply actions for each record that meets the condition specified. Creating a Nested For Each loop requires you to understand how many flow transactions this will generate and its duration. Nested For Each loops are discouraged as they may cause the flow transaction quota rule to stop a flow after an hour.

Subflows are a sequence of reusable actions that can be started from a Flow, Subflow, or Script. Subflows are very similar to flows, but there are some differences.

- A trigger is not used to start a subflow.
- Subflows can use inputs from a parent flow.
- Subflows can return outputs to the parent flow when the subflow ends.



TIP FROM THE FIELD

Flows are set to allow 50 actions or less. You can increase the number of actions with the **sn_flow_designer.max_actions** system property, but performance may be impacted. Also, consider using Subflows for repetitive tasks used across multiple flows.

Overview: Flow Data

- Flows store any data gathered or generated as variables in the **Data** panel.
- Each variable has its own Pill that Flow designers can use to drag-and-drop the variable value to an action input or output.

The variable with Trigger data is stored as a Pill in the Data pane

The Pill can be dragged to the action and used

TRIGGER

Incident Created or Updated where (Priority is 1 - Critical)

ACTIONS

1

Action: Update Incident Record

* Record: Trigger->Incident Record

* Table: Incident [incident]

* Fields: Assignment group

+ Add Field Value

Data

- Trigger - Record Created or Updated
 - Incident Record (Record)
 - Incident Table (Table)
 - Run Start Time (Date/Time)
- 1 - Update Record
 - Incident Record (Record)
 - Incident Table (Table)

As you create additional actions, additional Pills will be created. You can use the Pills in Actions, Flow Logics, and Subflows.

Above, a pill being used with an Action. Below is an example of Pills being used with Flow Logic and Subflows:

Flow Logic

The Trigger Data Pill is used in an If conditional to check the trigger incident record's priority.

3 If (If Priority 1-Critical) then

Condition Label: If Priority 1-Critical

* Condition 1: Trigger->Incident Record->Priority is 1 - Critical

Subflow

The Subflow is using the Pill from a previous Action as input. Subflows should be created for reuse. This modular design can save you from creating repetitive Actions across several flows.

4 Subflow 1

Subflow: Subflow 1

* Record Number: 1->Change Request Record

Module Labs

now.

- **Lab 11.1**

- **Time:** 15-20m
- Flow Designer: Build a Flow
 - Build a simple flow



Build a Flow

Lab
11.01

⌚15-20m

Lab Summary

You will achieve the following:

- Build a flow that responds to the change in priority of an incident.
- Test the flow within the Flow Designer.

Scenario

When an incident is created or updated with a priority of 1 - Critical, management would like for the incident to be assigned to the Incident Management group first.

Pre-Requisite

Since this is your first flow and you will test the result in flow designer, these system properties will enable flow engine debug messages and show system logs within the Flow Execution Details page.

1. Type **sys_properties.list** in the filter navigator.
2. Update the value for **com.glide.hub.flow_engine.debug** to true.
3. Update the value for **com.glide.hub.flow_engine.listener_trace** to true.

A. Build the Flow

1. To open the Flow Designer in a new tab, navigate to the **Flow Designer -> Designer** in the Filter Navigator.
2. On the right side of the screen, Select the **+ New** button.
3. From the drop-down, select the **New Flow** option.

4. In the Flow Properties window, add the following information:

- Name: **Incident Changed to P1**
- Application: **Global**
- Description: **Monitor Incidents for P1 Activity**

5. Select **Submit**

6. After the flow is saved successfully, add a Trigger by selecting the **+ Symbol** or **Click to add a Trigger** link.

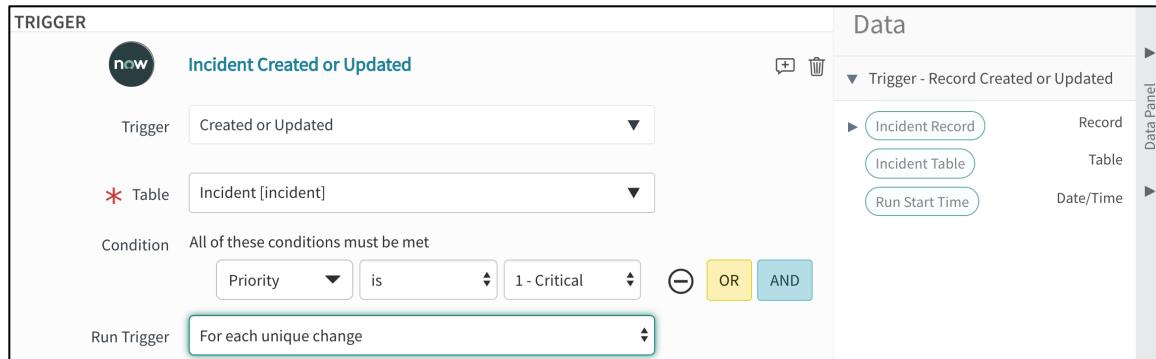
7. From the Trigger menu, select **Created or Updated**.

8. Select the Table drop down, type **Incident** in the search field, and select the **Incident [incident]** table.

9. To the right of Condition, select the **+ Add Filters** and set the drop down to the following:

Priority - is - 1-Critical

10. Select **For each unique change** on the Run Trigger drop-down.



11. Select **Done**.

12. In the Actions section, select the **+ symbol** or **Click to add an Action, Flow Logic, or Subflow** link.

13. From the buttons Action, Flow Logic, and Subflow, select the **Action** button.

Note: The Action menu has various Action Categories and the corresponding Actions. As you look at each Action, information will be shown in the description section. Be sure to read the description, so you choose the correct Action for your flow.

14. With the **ServiceNow Core** category highlighted, select the **Update Record** Action.



TIP FROM THE FIELD:

A good practice is to always create flows in a scoped application. This will categorize your content and make it easier to maintain and release.

15. Specify the record to update by dragging the pill labeled **Incident Record** from the **Data -> Trigger – Record Created or Updated** section on the right side of the screen to the **Record** field in the middle of the screen.

16. Move down to **Fields** and select the **+ Add Field Value** button.

17. Set the **Select a Field** dropdown fields to the following:

Assignment Group – Incident Management

18. Select **Done** to close the action.

19. Select the **Save** button in the upper right corner.

20. Select the **Activate** button and Select the **OK** button when the **Confirm Flow Active** dialog box is shown.

B. Test Your Work

You can test the flow with updates to the record that triggers the flow to execute. However, testing can be done within the Flow Designer.

Note: You should only test a flow in a Dev or Test instance, not a Production instance. Record changes caused by the test of a flow or integration call are real.

1. With the Incident Changed to P1 flow open, select the **Test** button.
2. In the Test Flow dialog box, select the **Record** field and choose an incident.
3. Select the **Run Test** button.
4. After the flow process completes, open the Execution Details tab with the **Flow has been executed. To view the flow, click here** link.
5. The Execution Details tab provides data and links about what occurred during the test. Review the items below to understand what occurred with its execution.
 - **Trigger and Actions** - Select the Trigger or the Actions to expand the sections. Review additional Information about what occurred when the trigger or action processed.
 - **Open Flow Logs** – View the flow logs of what occurred during the test.
 - **Open Current Record** – Open the form of the record to see the changes.
 - **Logs** – The drop-down shows the logs that were recorded during the flow execution.

6. Review the **Execution Details** screenshot below and select each link, indicated by the arrows, in your instance.

The screenshot shows the ServiceNow Flow Designer Execution Details page. The flow is titled "Incident Changed to P1". It contains one trigger step, "Incident Created or Updated", which has one action step, "Update Record". The "Update Record" step has a configuration section showing variable assignments and output data. A red arrow points to the "Open Flow Logs" link in the header. Another red arrow points to the "Open Current Record" link in the header. A third red arrow points to the "Update Record" link in the list of steps. A fourth red arrow points to the "No Logs" link at the bottom of the configuration section.

Note: The large amount of detailed information, gathered in one place, about the flow execution is convenient and helpful. Benefits of the Execution Detail become more apparent with larger flows.

7. Was the Test Run completed the way you expected? Why or why not?
-

8. Select the on the **Execution Details** Tab to close it.
9. On the flow, select the to close Test Flow dialog box.
10. Select the **Deactive** button and select **Ok** to Confirm Flow Deactivate.

Lab Completion

Good job! You have successfully created a flow and tested the execution of that flow.

Flow Designer Scripting: Topics

now.

Flow Designer Overview

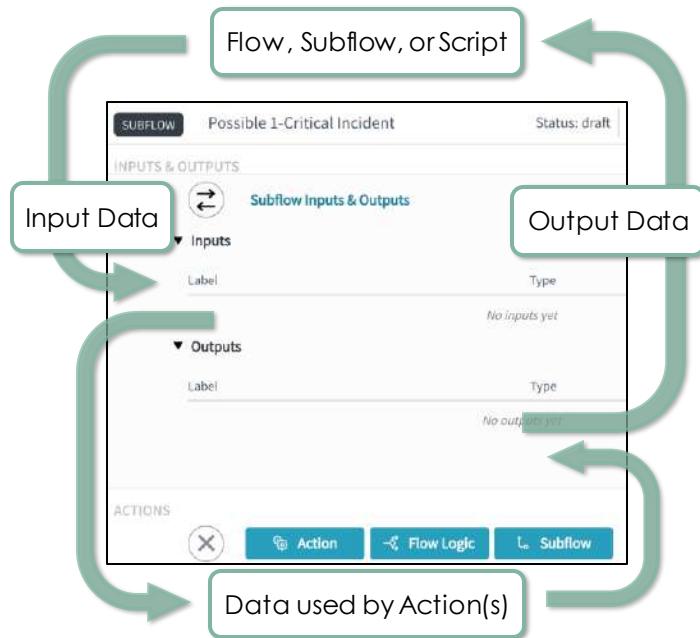
Script-Triggered Subflow

Action Designer Script Step

Script-Triggered Subflow: Subflows

now.

- Subflows are a sequence of reusable actions that can be started from a Flow, Subflow, or Script.
- Subflows do not require a trigger.
- Subflows contain inputs to make data available to Actions.
- Actions can use the data
- Data can be returned via the Outputs to be used by the module that called the Subflow



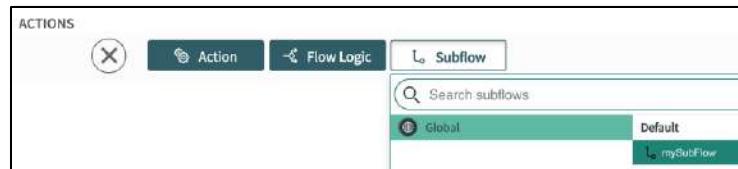
A subflow is a container of automated steps that a process analyst can use in a flow. A process analyst may need to post a message in Slack, query a Now platform table, or create approvals with emails, but they may not be familiar with the details to accomplish these tasks. By selecting a subflow with multiple steps, intensive calculations, or complicated connection strings, a process analyst can quickly take advantage of the power of a subflow and then incorporate it within the flow they are creating.

Subflows that automate generic business logic that helps process analysts and low code admins to not only integrate the capabilities of a Subflow into a flow, but into every additional flow that is created. The ability to reuse the functionality over and over in multiple applications or processes increases the speed at which a flow is built and its ease of creation.

Script-Triggered Subflow: Making a Subflow Run

now.

- Flow and Subflow Actions can call a **published** subflow



- Server Scripts (Business Rules, UI Actions, Script Includes) and Client Scripts can call a **published** subflow with code.
- The Server-side APIs can be executed in a Non-Blocking mode or Blocking with Timeout mode.
- The Client-side JavaScript library (GlideFlow) uses promise objects that provide an eventual result of an asynchronous operation
- There is a Code Snippet Generator!**

Subflows called from a script requirements:

- Create the subflow before you create the script to call it.
- Subflows must be published. Unpublished Subflows will not show up in your actions section and can not be called from code.
- The script you use to call a subflow can be a Server script or Client script.
- Each API has different functionality for the environment that it was designed for, so use the Code Snippet Generator to select the API for your application
- The Internal name of a subflow will be used. This is created by the designer and can be found on the Landing Page -> Subflow list. The internal name is a lowercase, no-spaces version of your subflow name
- Inputs should be Name-value pairs. If the subflow calls for a reference type value, use a GlideRecord object as the value.

Can a Flow or Actions be called by a script?

Yes. Flows, SubFlows, and Actions can be executed from script.



IMPORTANT

With all the new ways to call a Flow/Subflow, the following calls have been deprecated:

- sn_fd.Flow
- sn_fd.Subflow

Script-Triggered Subflow: Making a Flow, Subflow, or Action Run from a Server Side Script

now.



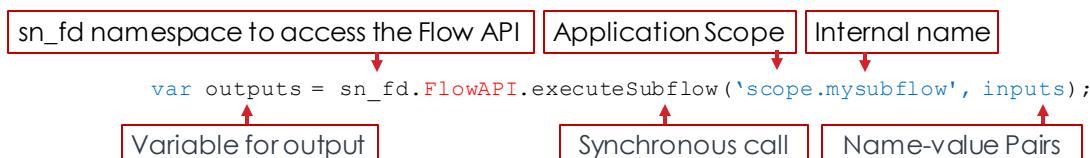
Server API – Non Blocking

```
sn_fd.FlowAPI.startAction(scope.myflow, inputs);  
sn_fd.FlowAPI.startSubflow (scope.mysubflow, inputs);  
sn_fd.FlowAPI.startFlow(scope.myAction, inputs);
```



Server API – Blocking

```
sn_fd.FlowAPI.executeAction(<scoped_name>, inputs,timeout_in_ms);  
sn_fd.FlowAPI.executeFlow(<scoped_name>, inputs,timeout_in_ms);  
sn_fd.FlowAPI.executeSubflow(<scoped_name>, inputs,timeout_in_ms);
```



Server API – Non Blocking

Pros: Immediately returns execution to the user. “Fire and forget” API.

Cons: No access to the outputs

Server API – Blocking

Pros: Outputs of Subflows / Actions will be immediately returned if no errors occurred during execution

Cons: Users need to wait for the call to complete

Note: The timeout for the blocking mode is the maximum time this Flow / Action will run before it is cancelled. If you do not add a timeout, the system default value is used which is determined by the property `com.glide.hub.flow_api.default_execution_time`. The default value is 5000 ms.



IMPORTANT

The Input map keys need to include data that the flow's record triggers are expecting.

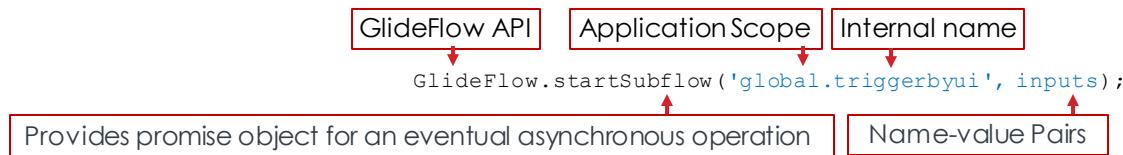
Script-Triggered Subflow: Making a Flow, Subflow, or Action Run from a Server Side Script

now.



Client-side API

```
GlideFlow.startFlow(String <scoped_name>, Object inputs);  
GlideFlow.startSubflow(String <scoped_name>, Object inputs);  
GlideFlow.startAction(String <scoped_name>, Object inputs);
```



Client-side API

Pros: Allows front-end access to Flows / Subflows / Actions

Cons: Not immediate as JavaScript promise objects are returned which represents the eventual result of an asynchronous operation.

Note: Server-side script is rigidly controlled compared to the Access to Client-side script. When using Client side script to execute a Flow, Subflow, or Action, there must be a new "***client_callable_flow_object***" ACL type assigned to it. If this is not done, by default access will be denied. All ACL management is handled with the "Manage Security" menu in Flow Designer.

Module Labs

now.

- **Lab 11.2**

- **Time:** 20-25m
- **Flow Designer:** Trigger a Subflow with a Script
 - Build a UI action that will trigger a subflow



Trigger a Subflow with a Script

Lab
11.02
20-25m

Lab Summary

You will achieve the following:

- Build a Subflow that can be triggered by a UI action.
- Create a UI Action that triggers the Subflow.
- Execute the UI Action that calls a Subflow and creates an Incident Task with values from the Incident.

Scenario

The Incident Management group has noticed some incidents did not need to be marked as a Priority 1-Critical. Since all critical Incidents are taken seriously and are reviewed by upper management, the service desk and the Incident Management group has decided that service desk employees needed another option. Incidents that are a possible candidate for Priority 1-Critical should be reviewed before being set to the highest Priority. Here is the solution:

- An Incident Task will be created and assigned to the Incident Management group.
- Since the incident could be time-sensitive, a UI action will quickly create an Incident task record and populates the form.

A. Build the Subflow

1. To open the Flow Designer in a new tab, navigate to the **Flow Designer -> Designer** in the Filter Navigator.
2. On the right side of the screen, Select the **+ New** button.
3. From the drop-down, select the **New Subflow** option.

4. In the Subflow Properties window, add the following information:
 - Name: **triggerbyui**
 - Application: **Global**
 - Description: **Subflow that creates an Incident Task for Incident Management to review**
5. After the Subflow is saved successfully, locate the **INPUTS & OUTPUTS** section, and select the **+ Click to create the inputs & outputs of your subflow.**
6. Under the **Inputs** section, select the  symbol.
7. Select the **Label** Field that is marked **variable** and change the wording to **number**.
8. Select the **Type drop down**. Choose the **Reference > Type** and Complete the Reference by selecting the **Incident [incident]** table.
9. Select the  symbol twice, and add the following Inputs with the information:
 - Label: **short_description**
 - Type: **String**
 - Label: **description**
 - Type: **String**



| Label | Type | Mandatory |  |
|-------------------|--------------------|-------------------------------------|---|
| number | Reference.Incident | <input checked="" type="checkbox"/> |  |
| short_description | String | <input type="checkbox"/> |  |
| description | String | <input type="checkbox"/> |  |

10. Select **Done**.
11. Locate the **Actions** section and select the **+ Click to add an Action, Flow Logic, or Subflow.**
12. Select **Action**.

13. For the Action, select **ServiceNow Core -> Create Record**.

14. For the Table Name, add **Incident Task [incident_task]**.

15. Select the **+ Add Field Value** and add the following fields and values

| Field | Value |
|-------------------|---|
| Incident | Select the number pill from the Data Panel -> Subflow Inputs and drag the pill to the empty field. |
| Priority | Select 3 – Moderate |
| Short Description | Type Possible Priority 1-Critical : Select the short_description pill from the Data Panel -> Subflow Inputs and drag the pill to the field. |
| Assignment Group | Select the Incident Management group |
| Description | Select the description pill from the Data Panel -> Subflow Inputs and drag the pill to the empty field |

The screenshot shows the subflow configuration interface. On the left, under 'ACTIONS', there is a step labeled '1 now Create Incident Task Record'. This step has an 'Action' set to 'Create Record', a 'Table Name' set to 'Incident Task [incident_task]', and a 'Fields' section. The 'Fields' section contains five dropdowns: 'Incident' (selected value: 'Input->number X'), 'Priority' (selected value: '3 - Moderate'), 'Short description' (selected value: 'Possible Priority 1-Critical Input->short_description X'), 'Assignment group' (selected value: 'Incident Management X'), and 'Description' (selected value: 'Input->description X'). Below these fields is a button '+ Add Field Value'. To the right of the actions is a 'Data' panel. The 'Subflow Inputs' section lists three fields: 'number' (Record), 'short_description' (String), and 'description' (String). Under '1 - Create Record', it shows the table 'Incident Task Record' (Record) and the table 'Incident Task Table' (Table). At the bottom right of the data panel is a 'Data Panel' button.

16. Select **Save**.

17. Select **Publish**.

B. Test the Subflow

1. With the Subflow page open, select **Test**.
2. With the **Possible 1-Critical Incident** Subflow open, select the **Test** button.
3. Add the following values to the Test Subflow dialog box.
 - **number:** **Select any Incident from the dropdown**
 - **short_description:** **Short Description Subflow Test**
 - **description:** **Description Subflow Test**

4. Select **Run Test**.
 5. Wait for the “Processing Test Subflow” to complete and then select the **Subflow has been executed. To view the subflow, click here**.
 6. In the **Execution Details**, select the **Subflow Inputs & Outputs** and the **Create Record** to expand those sections and review test results.
 7. Was the Test Run completed the way you expected? Why or why not?
-

8. Locate the Subflow Input Variable Name **number** and select the **link** to incident number you chose.

| | | |
|--------|-----------|--------------|
| number | Reference | INC0010016 ⓘ |
|--------|-----------|--------------|

9. Select **Open Record** on the preview incident section.



10. Add the Incident Task related list to the form by selecting the **Additional actions menu** button.

11. Select **Configure -> Related Lists**.

12. On the Configuring related lists on Incident form, scroll down the **Available** slush bucket until you locate the **Incident Task -> Incident** table, select the table to highlight the table, and select the **right arrow** button to move the table to the Selected slush bucket.

13. Select **Save**.

14. On the Incident form, scroll down to the related list, select the **Incident Tasks** tab, and verify that the Incident Task has been created with inputs you entered.

C. Build the UI action to call the SubFlow

1. On the Incident form, select the **Additional actions menu** button.
2. Select **Configure -> UI Actions**.
3. Select **New**.



4. Create a new UI Action.

Name: **Create Incident Management Task**
Table: **Incident [incident]**
Active: **Selected (checked)**
Form Link: **Selected (checked)**
Show Insert: **Not Selected (unchecked)**
Show Update: **Selected (checked)**
Comments: **Create Incident Management Task for Review**
Condition: **current.active;**

5. Examine the pseudo-code for the script:

- Use a GlideRecord object for the input of Reference type as the value
- Create Name-value pairs that define Subflow inputs.
- Use the sn_fd namespace to access the Subflow API.
- Use startAsync(String scopeName.subflowName, Map inputs) to start the Subflow.
- Store the Sys ID of the flow execution as contextId

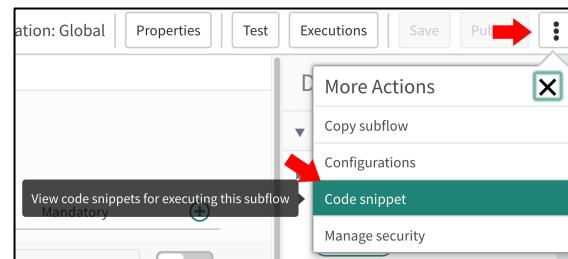
6. Select the **Flow Designer** tab.

7. Navigate to the **triggerbyui** subflow.

8. Select the **More Actions(:)** icon.

9. Select **Code Snippet**.

10. Use the **Server Script** that is generated in the **UI Action Script** field.



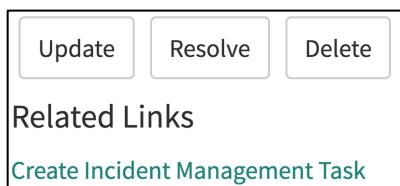
11. Make the following changes to the script:

```
(function() {  
  
    try {  
        var incRec = new GlideRecord('incident');  
        incRec.get(current.sys_id);  
  
        var inputs = {};  
        inputs['number'] = incRec; // GlideRecord of table: incident  
        inputs['short_description'] = current.short_description; // String  
        inputs['description'] = current.description; // String  
  
        // Start Asynchronously: Uncomment to run in background. Code snippet will not  
        // have access to outputs.  
        // sn_fd.FlowAPI.startSubflow('global.triggerbyui', inputs);  
  
        // Execute Synchronously: Run in foreground. Code snippet has access to outputs.  
        var outputs = sn_fd.FlowAPI.executeSubflow('global.triggerbyui', inputs);  
  
        // Current subflow has no outputs defined.  
    } catch (ex) {  
        var message = ex.getMessage();  
        gs.error(message);  
    }  
  
})());
```

12. Select **Submit**.

D. Test the UI Action

1. Navigate to the incident you used in your Subflow testing.
2. Select the UI Action **Create Incident Management Task** Related Link.



3. Verify the Incident Task has been created in the Related Lists.

Note: The Incident Task may not be visible immediately. Refresh the incident if does not show up.

4. Was the Incident Task created? If not, debug and retest.

Lab Completion

Congratulations on completing the lab! You have successfully called a Subflow from a script.

Flow Designer Scripting: Topics

now.

Flow Designer Overview

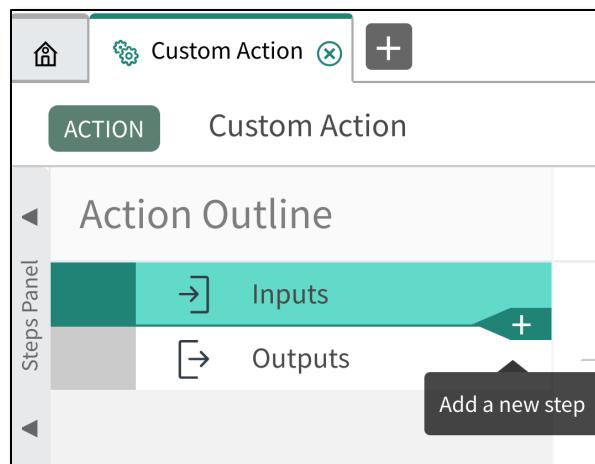
Script-Triggered Subflow

Action Designer Scripting Step

Action Designer Script Step: Custom Actions

now.

- Custom Actions should be used when there is a need for functionality the ServiceNow-Provided Actions do not provide.
- Design a Custom Action for reusability.
- Within the Steps Panel, the Action Outline contains the parts of an Action:
 - Inputs
 - Action Steps
 - Outputs



The last section in this Flow Designer module is creating an Action. It follows in the same style as Flow and Subflow. This is a top-down down approach that lowers the barrier to entry for process owners and low code admins to create Actions. These roles will use the set of core action steps to automate Now Platform processes. Also, subject matter experts, scripters, and developers can use the core action steps to quickly incorporate tedious tasks into a Custom Action. This leaves more time to focus on the other aspects of creating Flows.

Reusability

Reusability has been stressed in this module. You may recall from the Script Include module, script includes are consider a write once use many times component. A key difference between the reuse of Script Includes and Actions is the accessibility. Script Includes need to be called from code. Flow Designer interface makes code reuse easier by making the Action easily discoverable. Think about how other departments within your organization can utilize the same Actions you create.



IMPORTANT

The action_designer role is required to build Custom Actions.

Action Designer Script Step: Inputs and Outputs

now.

- Inputs and Outputs are data variables used in your action.
- Inputs store the information passed to the action in the Data Panel and can be used in the Action Step.
- Outputs return data back to the flow to be used by the flow or other actions.

Select an Input or Output to define the data variables

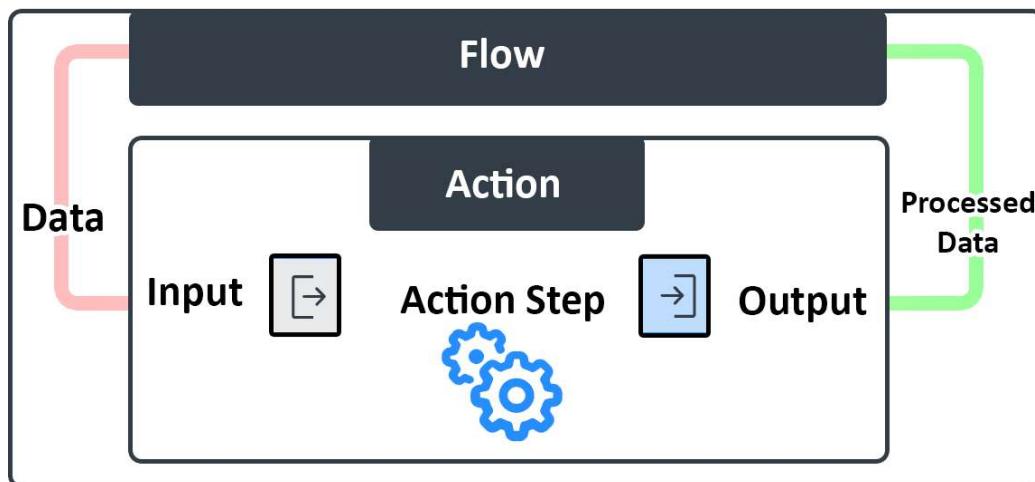
Action Input/Outputs can have multiple variable to support the Action Steps.

Input variables support a multitude of types.

The Data Panel stores the data in pills used through the Action.



Just like the subflow you created in the previous lab, inputs and outputs in an Action function the same way. The Input of an Action allows data, of various types, to pass to the Action Steps. After processing the data, Information is relayed back to the flow. This returning data may be used in additional subflows or actions.



Action Designer Script Step: Core Action Steps

- Steps are singular reusable operations that make up an action
- Action steps are added to actions from the Action Designer design environment.
- Creating an action step requires knowledge of application tables, fields, and business logic

Each Core Action Step will require information and data from the Data Panel to complete the step

Just as Flow designer has a number of core actions, several core action steps have been provided to help with building an action.

| ServiceNow Data | |
|---|---|
| Create Task Create a task that you can specify to wait for completion. | Look Up Records Return the count and Records that meets the search criterion. Records can be used in flow iterations. |
| Wait For Condition Pause the flow to wait for a state change on a record. | Create Record Create or return a record on a given table. Fails if business rules or data policies prevent the update. |
| Ask for Approval Request for an approval from users, groups, and manual approvers on a given record using rule sets. | Update Record Update a record on a given table. Fails if business rules or data policies prevent the update. |
| Delete Record Deletes a record. Fails if no record is found or business rules or data policies prevent the delete. | Look Up Record Look up a record that meets the search criterion. |

| Utilities | |
|--|-------------------------|
| Script Executes a custom Javascript. <small>Note: Requires a subscription for integration. Read More</small> | Email Send an email. |
| Notification Trigger a notification. | Log Log a message. |

| Integrations | |
|---|--|
| To enable Action Steps for integrations, the IntegrationHub plugin is required. Read More | |

Action Designer Script Step: Script Action Step

now.

- Script Action Step can execute functionality not in a core action step.
- Before Scripting a solution, verify that the core Actions Steps and a combination of core Action Steps can not create the desired result.
- The Javascript written should be reusable and ready for others to use.

The screenshot shows the 'Select Step to add' dialog in ServiceNow. At the top, it says 'Select Step to add'. Below that, there's a section titled 'ServiceNow Data' containing 'Create Task', 'Look Up Records', and 'Wait For Condition'. Under 'Utilities', there's a box labeled 'Script' which is highlighted with a red border. This box contains the text: 'Executes a custom Javascript.' and 'Note: Requires a subscription for integration. [Read More](#)'. Below the 'Script' box are 'Notification' and 'Email'. Under 'Integrations', there's a box for 'IntegrationHub' with the note: 'To enable Action Steps for integrations, the IntegrationHub plugin is required. [Read More](#)'.

Previous modules have stressed the importance that scripts should only be created when a feature or functionality is not available in a baseline instance. Action Designers should follow this same advise when a Script Step is created.

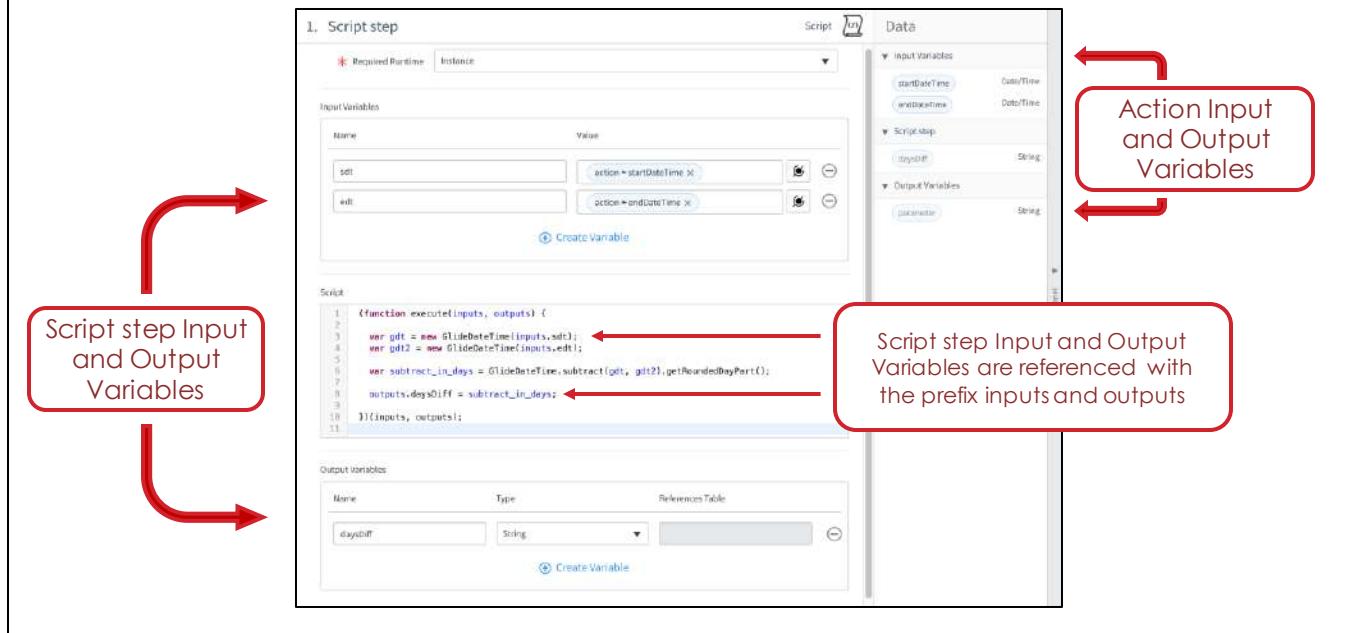


IMPORTANT

The Script step provides the ability to create integrations with third party systems, but this functionality requires the IntegrationHub plugin to be activated. If you are creating a script for your instance only, the plugin is not required.

Action Designer Script Step: Code

now.



The Javascript steps include separate Input and Outputs Variables. A Script step Input Variable will need to be created, named, and an Action Input Variables data pill added to the Value field to make the Action Input Variable data accessible to the script. Also, a Script step Output Variables can be created and named to return any results back to the action. The Script Step Input and Output Variable can be referenced in the script with the inputs and outputs prefix.

Module Labs

now.

- **Lab 11.3**

- **Time:** 20-25m
- **Flow Designer:** Add a Script to a Flow
 - Build a custom action that uses a script step.
 - Build a flow that uses a custom action



Add a Script to a Flow

Lab
11.03

⌚20-25m

Lab Summary

You will achieve the following:

- Build a custom action that uses a script step
- Build a flow that includes the custom action

Note: Visit the knowledge base for the “Module 11 - Student Resource”. The variables are available to copy and paste. The “Report” variable will get you started, but will need to be updated with the correct variables.

Scenario

The Priority 1-Critical has been resolved. Since the critical incidents are reviewed by management, a request has been made for an After Action Report to be added to the record’s work notes. Data should be used from the incident to make a brief summary.

A. Build the Custom Action

1. Navigate to the **Flow Designer -> Designer** in the Filter Navigator.
2. On the right side of the screen, Select the **+ New** button.
3. From the drop-down, select the **New Action** option.
4. In the Action Properties window, add the following information:

Name: **Lab 11.3 Custom Action Script**
Accessible From: **All Application Scopes**
Application: **Global**
Description: **Create an After Action Report**

5. Select **Submit**.
6. In the Action Input, select the **+ Create Input**.

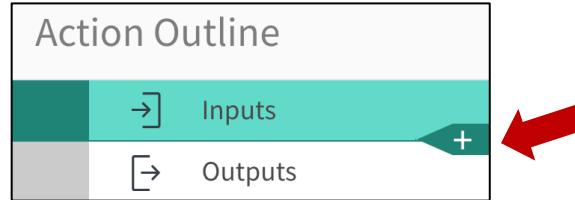
7. Select **variable** to change the Label, the **drop down** to change the Type, and the **+ Create Input** for new variables to add the following inputs to the Action Input section.

| Label | Type |
|--------------------|-----------|
| Assigned To | String |
| Assignment Group | String |
| Created on | Date/Time |
| Created By | String |
| Resolved | Date/Time |
| Resolved By | String |
| Number | String |
| Caller | String |
| Configuration Item | String |
| Resolution Code | String |
| Resolution Notes | String |
| Short Description | String |

TIP FROM THE FIELD:
This lab has several inputs and longer text portions than other labs. Use the Student Resources file to copy and paste text, so you can focus on scripting.

| Label | Type | Mandatory | Data |
|--------------------|-----------|-------------------------------------|---|
| Assigned To | String | <input checked="" type="checkbox"/> | <input type="button" value="Assigned To"/> String <input type="button" value="Assignment Group"/> String |
| Assignment Group | String | <input checked="" type="checkbox"/> | <input type="button" value="Created on"/> Date/Time <input type="button" value="Created By"/> String |
| Created on | Date/Time | <input checked="" type="checkbox"/> | <input type="button" value="Resolved"/> Date/Time <input type="button" value="Resolved By"/> String |
| Created By | String | <input checked="" type="checkbox"/> | <input type="button" value="Number"/> String <input type="button" value="Caller"/> String |
| Resolved | Date/Time | <input checked="" type="checkbox"/> | <input type="button" value="Configuration Item"/> String <input type="button" value="Resolution Code"/> String |
| Resolved By | String | <input checked="" type="checkbox"/> | <input type="button" value="Resolution Notes"/> String <input type="button" value="Short Description"/> String |
| Number | String | <input checked="" type="checkbox"/> | |
| Caller | String | <input checked="" type="checkbox"/> | |
| Configuration Item | String | <input checked="" type="checkbox"/> | |
| Resolution Code | String | <input checked="" type="checkbox"/> | |
| Resolution Notes | String | <input checked="" type="checkbox"/> | |
| Short Description | String | <input checked="" type="checkbox"/> | |

- Under the Action Outline, Select the + symbol between the Inputs and Outputs.



- On the Select Step to add window, locate the Utilities section and select the **Script Step**.
- Under the 1. Script step section, select the + **Create Variable**.
- Add the names below and drag the data pills to the values to create Script step Input Variables.

| Name | Value |
|-------------------|-----------------------------|
| assigned_to | action ->Assigned To |
| assignment_group | action ->Assignment Group |
| created_on | action ->Created on |
| created_by | action ->Created By |
| resolved_at | action ->Resolved |
| resolved_by | action ->Resolved By |
| number | action ->Number |
| caller_id | action ->Caller |
| cmdb_ci | action ->Configuration Item |
| closed_code | action ->Resolution Code |
| close_notes | action ->Resolution Notes |
| short_description | action ->Short Description |

Note: Do not use System Variable names for Input Variables. Names like sys_created_by or sys_created_on disappear when saved. No warning message is given.

- Under the Output Variables, add the Name **Payload** and the Type **String**.

- Examine the pseudo-code for the script you will write:

- Create variables for the Create and Resolved Date Time Inputs
- Calculate the difference between Create and Resolved in days
- Create a report variable that will hold the After Action Report
 - Include the calculated number of days in the report

14. Write the script in script field:

```
(function execute(inputs, outputs) {  
  
    var startDate = new GlideDateTime(inputs.created_on);  
    var endDate = new GlideDateTime(inputs.resolved_at);  
  
    var calcDiff = GlideDateTime.subtract(startDate, endDate).getRoundedDayPart();  
  
    var report = "After Action Report\n" +  
        "Subject: After Action Report, " + inputs.assigned_to + " - " + inputs.assignment_group + "\n" +  
        inputs.created_on + " - " + inputs.resolved_at + "\n\n" +  
        inputs.number + " with '" + inputs.short_description + "' was entered into ServiceNow by " +  
        inputs.created_by + " for " + inputs.caller_id + ". " +  
        "The Configuration Item affected was " + inputs.cmdb_ci + ". " +  
        "Resolved by " + inputs.resolved_by + " with a Resolution code of " + inputs.closed_code + ". " +  
        "Resolution notes are as follows: " + inputs.close_notes + "\n\n" +  
        "Total time: " + calcDiff + " Days";  
  
    outputs.Payload = report;  
  
})(inputs, outputs);
```

15. Under the Action Outline, select the **Outputs**.

16. Select **+ Create Output** to create an Action Output.

Label: **After Action Report**

Value: **Step-> Script step -> Payload**

(Drag the Payload Pill from the Data Panel)

17. Select **Save**.

18. Select **Publish**.

B. Create the Flow

1. In the tab section, select the **Plus (+)** icon.

2. From the drop-down, select the **New Flow** option.

Name: **Priority 1-Critical Incident Closed**

Description: **Create After Action Report**

3. After the flow is saved successfully, select the **+ Symbol** or **Click to add a Trigger**.

4. Create an **Updated** trigger with these conditions:

Table: **Incident[incident]**

Condition: **State - is - Closed** and

Priority - is - 1-Critical and

Active - is - false

Run Trigger: **Once**

5. Select **Done**.
6. In the Actions section, select the **+ symbol** or **Click to add an Action, Flow Logic, or Subflow**.
7. From the buttons Action, Flow Logic, and Subflow, select the **Action** button.

8. With the **Global** category highlighted, select the **Lab 11.3 Custom Action Script** Action.
9. Select the **Data Pill Picker** () or **Data Panel** to dot-walk to the data for each Action Input.

Note: To dot-walk to data in the Data Panel, look for the right arrow()next to the Data Pill. Select the right arrow and data from that table is shown below the pill.

To dot-walk to data in the Data Pill Picker, select the location of the data. In this case, it will be the **Trigger - Record Updated**. Select a right arrow() of referenced data. When you have drilled down to the table, use the search bar to narrow the fields shown.

TIP FROM THE FIELD:
The Data Pill Picker can be faster than the mouse to dot-walk to the data you need. Use the arrow keys and type the field name for navigation.

10. Update the Action Inputs

| | | |
|--------------------|--|--|
| Assigned To | Trigger->Incident Record->Assigned to->Name X | |
| Assignment Group | Trigger->Incident Record->Assignment group->Name X | |
| Created on | Trigger->Incident Record->Created X | |
| Created By | Trigger->Incident Record->Created by X | |
| Resolved | Trigger->Incident Record->Resolved X | |
| Resolved By | Trigger->Incident Record->Resolved by->Name X | |
| Number | Trigger->Incident Record->Number X | |
| Caller | Trigger->Incident Record->Caller->Name X | |
| Configuration Item | Trigger->Incident Record->Configuration item->Name X | |
| Resolution Code | Trigger->Incident Record->Resolution code X | |
| Resolution Notes | Trigger->Incident Record->Resolution notes X | |
| Short Description | Trigger->Incident Record->Short description X | |

11. Select **Done**.

12. Select the **+** symbol or **Click to add an Action, Flow Logic, or Subflow**.

13. From the buttons Action, Flow Logic, and Subflow, select the **Action** button.

14. With the **ServiceNow Core** category highlighted, select the **Update Record** Action.

15. Select and drag the **Incident Record** from the Data Panel to the **Record** field.
16. Select the **Fields** dropdown and choose **Work notes**.
17. Select and drag the Lab 11.3 Custom Action Script **After Action Report** data pill to the Fields value field.

The screenshot shows the 'Update Incident Record' flow in the ServiceNow Flow Designer. The flow consists of a single step: 'Update Record'. There are four input fields:

- Record:** Trigger->Incident Record
- Table:** Incident [incident]
- Fields:** Work notes
- Fields:** 1->After Action Report (highlighted with a green border)

18. Select **Done**.

19. Select **Save**.

C. Test Your Work

1. Select the **browser tab** with the regular ServiceNow interface.

2. Create an Incident record.

Caller: **Abel Tuter**
 Configuration Item: **3D Pinball**
 Short Description: **3D Pinball is Broken**
 Impact: **1-High**
 Urgency: **1-High**
 Assignment Group: **Incident Management**
 Assigned To: **Incident Manager**
 Resolution Code: **Solved (Work Around)**
 Resolution Notes: **Inserted a quarter**
 Resolved: **Pick at least two days in the future.**
 Resolved By: **Beth Anglin**

3. Select **Save**.

4. Return to **Flow Designer**.

5. On the flow **Priority 1-Critical Incident Closed**, select **Test**.
6. Use the number from the Incident created in the **Record** field.
7. Select **Run Test**.
8. After the flow process completes, open the Execution Details tab with the **Flow has been executed. To view the flow, click here** link.
9. Was “work_notes=After Action Report Subject: After Action Report, Incident Manager – Incident Management...” shown on the **Update Record -> RUNTIME VALUE** column? If not Debug and retest.
10. Return to the **Incident** created. Was After Action Report in the Work notes activities?

 System Administrator Work notes • 2018-12-12 11:35:22

After Action Report
Subject: After Action Report, Incident Manager – Incident Management
2018-12-12 19:25:09 – 2018-12-14 19:24:39

INC0010036 with '3D Pinball is Broken' was entered into ServiceNow by admin for Abel Tuter. The Configuration Item affected was 3D Pinball. Resolved by Beth Anglin with a Resolution code of Solved (Work Around). Resolution notes are as follows: Inserted a quarter

Total time: 2 Days

Lab Completion

Good job! You have successfully created a flow and tested the execution of that flow

Good Practices

- Determine if Flow Designer or Workflow is the better solution for your situation
- Before creating an action, subflow, or script, determine if a flow can be completed with core functionality.
- If you need to create a custom action or subflow, incorporate reusability.
- Use all the features of Flow Execution Details to debug flows.

Module Recap: Flow Designer Scripting

now.

| Core Concepts | Real World Cases |
|--|--|
| Flow Designer can be used for Kingston and later releases | • Why would you use these capabilities? |
| The natural language and low code approach to flow creation makes it easy for many roles | • When would you use these capabilities? |
| Actions and subflows provide reusable business logic across multiple flows | • How often would you use these capabilities? |
| Action and subflows easy discovery helps with the flow creation process | |
| Evaluate the core actions before scripting a solution. | |

Discuss: Why, when, and how often would you use the capabilities shown in this module.

Congratulations

now.

Course Complete

You have completed the **Scripting in ServiceNow Fundamentals** course!



www.servicenow.com/services/training-and-certification.html



4810 Eastgate Mall, San Diego, CA 92121, USA • (858) 720-0477 • (858) 720-0479 • www.servicenow.com

©2019 ServiceNow, Inc. All rights reserved.

ServiceNow believes information in this publication is accurate as of its publication date. This publication could include technical inaccuracies or typographical errors. The information is subject to change without notice. Changes are periodically added to the information herein; these changes will be incorporated in new additions of the publication. ServiceNow may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time. Reproduction of this publication without prior written permission is forbidden. The information in this publication is provided "as is". ServiceNow makes no representations or warranties of any kind, with respect to the information in this publication, and specifically disclaims implied warranties of merchantability or fitness for a particular purpose. ServiceNow is a trademark of ServiceNow, Inc. All other brands, products, service names, trademarks or registered trademarks are used to identify the products or services of their respective owners.