

Scripting In ServiceNow

Module 1. CLIENT-SIDE SCRIPTING

Content:

1.1 Introduction to Client-Side Scripting

1.2 Client Scripts

1.3 The GlideForm (g_form) Class

1.4 The GlideUser (g_user) Class

1.5 UI Policy

1.6 UI Policy Action

1.7 Client Script vs UI Policy

1.8 Module Recap

In this module you will learn to:

- Describe the purpose of a client-side script and give examples of what client-side scripts can do
- Create and test Client Scripts
- Create and test UI Policy scripts
- Use the *GlideForm* and *GlideUser* APIs in scripts
- Determine whether to use UI Policy scripts or Client Scripts

1.1 Introduction to Client-Side Scripting:

This learning module is about client-side scripting. Client-side scripts execute within a user's browser and are used to manage forms and form fields. Examples of things client-side scripts can do include:

- Place the cursor in a form field on form load
- Generate alerts, confirmations, and messages
- Populate a form field in response to another field's value
- Highlight a form field
- Validate form data
- Modify choice list options

In this module you will learn to write, test and debug two types of client-side scripts:

- Client Scripts
- UI Policy Scripts

1.2 Client Scripts (sys_script_client):

Client scripts allow the system to run JavaScript on the client (web browser) when client-based events occur, such as when a form loads, after form submission, or when a field changes value.

1.2.1 There are several different types of client scripts in ServiceNow, including:

1. **OnLoad scripts:** These scripts are executed when a form or page is loaded in the web browser. OnLoad scripts can be used to set default values for fields, modify the behavior of UI elements, or perform other tasks to prepare the form or page for user interaction. An example of an onLoad client script in ServiceNow might look like this:

```
function onLoad() {  
  
    // Set the default value for the "Priority" field  
  
    g_form.setValue('priority', '3 - Low');  
  
  
  
    // Hide the "Assignment Group" field if the "Assigned to" field is empty  
  
    if (g_form.getValue('assigned_to') == "") {  
  
        g_form.setDisplay('assignment_group', false);  
  
    }  
  
}
```

In this example, the onLoad script is executed when the form is loaded in the web browser. The script sets the default value for the "Priority" field to "3 - Low", and then checks the value of the "Assigned to" field. If the "Assigned to" field is empty, the script hides the "Assignment Group" field.

2. OnChange scripts:

These scripts are executed when the value of a form field is changed by the user. OnChange scripts can be used to perform validation on user input, update other fields based on the new value, or take other actions in response to the change. An example of an onChange client script in ServiceNow might look like this:

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

    if (isLoading || newValue == "") {
        return;
    }

    // If the user changes the value of the "Category" field,
    // set the default value for the "Subcategory" field

    if (g_form.getValue('id') == 'category') {
        if (newValue == 'Hardware')
        {
            g_form.setValue('subcategory', 'Laptop');
        }

        else if (newValue == 'Software')
        {
```

```

        g_form.setValue('subcategory', 'Operating System');
    }
}
}

```

In this example, the onChange script is executed whenever the user changes the value of a form field. The script checks the ID of the field that was changed, and if it is the "Category" field, it sets the default value for the "Subcategory" field based on the new value of the "Category" field. For example, if the user selects "Hardware" as the category, the script sets the default value for the subcategory to "Laptop".

3. **OnSubmit scripts:** These scripts are executed when a user submits a form by clicking the Save button or performing another action that triggers a submission. OnSubmit scripts can be used to perform final validation on the form data, gather additional information, or perform other tasks before the form data is saved to the ServiceNow database. An example of an onSubmit client script in ServiceNow might look like this:

```

function onSubmit() {

    // Perform final validation on the form data

    if (g_form.getValue('category') == '') {

        g_form.showFieldMsg('category', 'Category is required', 'error');

        return false;

    }

    // Gather additional information before the form is submitted

    g_form.setValue('caller_id', g_user.userID);

    g_form.setValue('opened_at', g_form.getCurrentDateTime());

    // Return true to indicate that the form should be submitted

    return true;

}

```

In this example, the `onSubmit` script is executed when the user clicks the Save button to submit the form. The script first performs final validation on the form data, making sure that the "Category" field has a value. If the "Category" field is empty, the script displays an error message and returns false to prevent the form from being submitted. If the "Category" field has a value, the script gathers additional information (the caller ID and the time the form was submitted) and sets these values in the form. Finally, the script returns true to indicate that the form should be submitted to the ServiceNow database.

4. **OnCellEdit scripts:** These scripts are executed when a user edits the value of a cell in a list or table. OnCellEdit scripts can be used to validate the new value, update other cells based on the new value, or take other actions in response to the edit. An example of an onCellEdit client script in ServiceNow might look like this:

```
function onCellEdit(sysIDs, table, oldValues, newValue, callback) {  
  
    var saveAndClose = true;  
  
    var isAdmin = g_user.hasRole('admin');  
  
    if ((!isAdmin && newValue == 2) || (!isAdmin && newValue == 3)){  
        alert('Not allowed to set this state');  
        saveAndClose = false;  
    }  
  
    callback(saveAndClose);  
}
```

In this example we are restricting the updating of state field from lissst view if a person is not an admin.

1.2.2 Configuring the Client Script

As with any script, the configuration tells the script when to execute. The Client Script configuration options are:

Name: Name of Client Script. Use a standard naming scheme to identify custom scripts.

Table: Table to which the script applies.

UI Type: Select whether the script executes for Desktop and Tablet or Mobile/Service Portal or All.

Type: Select when the script runs: onChange, onLoad, or onSubmit.

Field Name: Used only if the script responds to a field value change (onChange); name of the field to which the script applies.

Active: Controls whether the script is enabled. Inactive scripts do not execute.

Inherited: If selected, this script applies to the specified table and all tables that inherit from it. For example, a client script on the Task table will also apply to the Change, Incident, Problem, and all other tables which extend Task.

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

The **Field name** field is available for onChange Client Scripts. The *View* field is available when the *Global* option is not selected.

1.2.3 Benefits of Client Script :

There are several benefits to using client scripts in ServiceNow, including:

Customization: Client scripts allow you to customize the behavior of forms and other UI elements in the ServiceNow platform, making it easier to tailor the platform to your specific needs and requirements.

User experience: Client scripts can be used to enhance the user experience on the ServiceNow platform, by providing additional functionality and making it easier for users to interact with forms and other UI elements.

Efficiency: Client scripts can help to automate and streamline common tasks and processes, making it faster and easier for users to complete tasks and access the information they need.

Flexibility: Client scripts provide a versatile and powerful way to customize the ServiceNow platform and can be used in a wide range of scenarios and contexts to meet the specific needs of your organization.

1.2.4 The Script Field

When the type value is set, a script template is automatically inserted into the *Script* field.

onLoad Script Template

The *onLoad* script template:

The image shows a screenshot of the ServiceNow 'Script' field editor. At the top, there is a toolbar with various icons for editing and saving. Below the toolbar, the script content is displayed in a text area. The script is a function named 'onLoad' that takes no arguments. It includes a comment instructing the user to add their script logic and begin with a comment. The script is formatted with line numbers 1 through 4.

```
1 function onLoad() {  
2     //Type appropriate comment here, and begin script below  
3  
4 }
```

The *onLoad* function has no arguments passed to it. As indicated by the comment, add your script logic in the *onLoad* function. It is a best practice to document your code so include comments to explain what the script does. Your future self will thank you for the clearly documented script.

This example script generates an alert when a user requests a form load for a record. The user cannot interact with a form until *onLoad* Client Scripts complete execution.



```
1 function onLoad() {  
2     //Type appropriate comment here, and begin script below  
3     alert("Thank you for loading the form. When you close this alert  
4     you will be able to interact with the form.");  
5 }
```

onSubmit Script Template


The *onSubmit* script template:



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

The *onSubmit* function has no arguments passed to it. As indicated by the comment, add your script logic in the *onSubmit* function.

This example script generates an alert when a user saves a *NeedIt* record. The record is not submitted to the server until the *onSubmit* Client Scripts complete execution and return true.



```
1 function onSubmit() {  
2     //Generate an alert to thank the user for submitting a record  
3     alert("Thank you for submitting a record.");  
4 }
```

The *onChange* Script template:

The *onChange* function is automatically passed five parameters by ServiceNow. Although you do not need to do anything to pass the parameters, you can use them in your script.

- **control**: field the Client Script is configured for.
- **oldValue**: value of the field when the form loaded and prior to the change.
- **newValue**: value of the field after the change.
- **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. When forms load, all the field values on the form change as the record is loaded into the form.
- **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.

When a user selects a record to load, the form and form layout are rendered first and then the fields are populated with values from the database. From the technical perspective, all field values are changed, from nothing to the record's values, when a form loads. The *if* statement in the template assumes that *onChange* Client scripts should not execute their script logic when a form loads. The *onChange* Client Script also stops execution if the *newValue* for a field is no value. Depending on your use case, you can modify or even remove the *if* statement. For example:

```
//Stop script execution if the field value change was caused by a Template

if(isLoading || newValue === '' || isTemplate) {

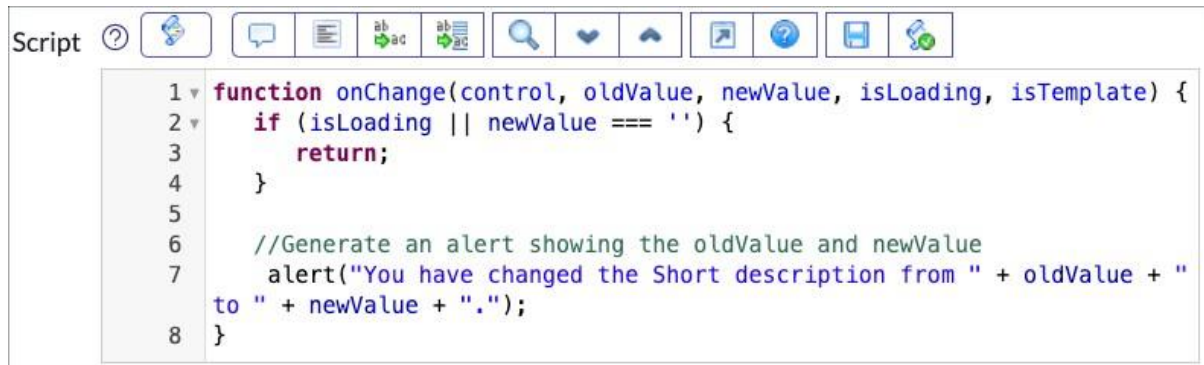
    return;

}
```

For this example, the *onChange* Client Script executes because of changes to the *Short description* field value:

Type	onChange ▼
Field name	Short description ▼

When a user changes the value of the *Short description* field on the form the *onChange* script logic executes. The example generates an alert stating that the value of the *Short description* field has changed from the value the field had when the form loaded to the new value on the form.

A screenshot of a script editor window. The title bar says "Script". The editor contains a JavaScript function named `onChange` with parameters `control`, `oldValue`, `newValue`, `isLoading`, and `isTemplate`. The function has three lines of code: a conditional return, a comment, and an alert call. The code is as follows:

```
1 function onChange(control, oldValue, newValue, isLoading, isTemplate) {  
2   if (isLoading || newValue === '') {  
3     return;  
4   }  
5  
6   //Generate an alert showing the oldValue and newValue  
7   alert("You have changed the Short description from " + oldValue + "  
8   to " + newValue + ".");  
9 }
```

The value of the *Short description* field has changed *only* on the form. Changes are not sent to the database until a user saves, updates, or submits the record.

It is important to know that the value of *oldValue* is set *when the form loads*. Regardless of how many times the value of the *Short description* field changes, *oldValue* remains the same until the form is re-loaded from the database.

1. Form loads:
 - a. *oldValue* = hello
 - b. *newValue* has no value
2. User changes the value in the *Short description* field to bye:
 - a. *oldValue* = hello
 - b. *newValue* = bye
3. User changes the value in the *Short description* field to greetings:
 - a. *oldValue* = hello
 - b. *newValue* = greetings
4. User saves the form and reloads the form page:
 - a. *oldValue* = greetings
 - b. *newValue* has no value

1. 3 The GlideForm (g_form) Class

The *GlideForm* client-side API provides methods for managing form and form fields including methods to:

- Retrieve a field value on a form
- Hide a field
- Make a field read-only
- Write a message on a form or a field
- Add fields to a choice list
- Remove fields from a choice list

The *GlideForm* methods are accessed through the global *g_form* object that is only available in client-side scripts. To use methods from the *GlideForm* class use the syntax:

```
g_form.<method name>
```

For example, the GlideForm API has a method called `getValue()`. The `getValue` method gets the value of a field on the form (not the value in the database). The example script gets the value of the Short description field from the form and displays the Short description field value in an alert.

```
alert(g_form.getValue('short_description'));
```

In addition to the `getValue()` method, other commonly used GlideForm methods include:

`addOption()`

`clearOptions()`

`addInfoMessage()`

`addErrorMessage()`

`showFieldMsg()`

`clearMessages()`

`getSections()`

`getSectionName()`

1.4 The GlideUser (g_user) Class

The GlideUser API provides methods and non-method properties for finding information about the currently logged in user and their roles. The typical use cases are personalizing feedback to the user and inspecting user roles. Note that client-side validation in any web application is easily bypassed.

The GlideUser API has properties and methods to:

1. Retrieve the user's:

First name

Full name

Last name

User ID

User name

2 . Determine if a user has a particular role assigned

The GlideUser methods and non-method properties are accessed through the global g_user object which is only available in client-side scripts. To use methods and properties from the GlideUser class use the syntax:

```
g_user.<method or property name>
```

For example, the GlideUser API has a property called userName. The userName property's value is the currently logged in user's user name. The example script shows the difference between the firstName, lastName, userName, and userID property values.

```
alert("g_user.firstName = " + g_user.firstName  
+ ", \n g_user.lastName = " + g_user.lastName
```

```
+ ", \n g_user.userName = " + g_user.userName  
+ ", \n g_user.userID = " + g_user.userID);
```

The alert generated by the script is:

```
g_user.firstName = System  
g_user.lastName = Administator  
g_user.userName = admin  
g_user.userID = System_ Administator
```

The `g_user.userID` property contains the record's `sys_id`. Every record has a 32-character unique `sys_id`.

Although you could concatenate the output of `g_user.firstName` with `g_user.lastName`, the convenience method, `g_user.getFullName()` concatenates the two.

The `GlideUser` API also has methods for determining if a user has a specific role. For example:

```
g_user.hasRole('client_script_admin');
```

The example script checks to see if the currently logged in user has the capability to create and edit Client Scripts (`client_script_admin` role). Note that the script returns true not only when the currently logged in user has the role assigned but also if the currently logged in user has the admin role. The admin user has all roles implicitly assigned. To test whether the currently logged in user has the role explicitly assigned, use the `hasRoleExactly()` method:

```
g_user.hasRoleExactly('client_script_admin');
```

1.5 UI Policies

Like Client Scripts, UI Policies are client-side logic that governs form and form field behavior. Unlike Client Scripts, UI Policies do not always require scripting.

1.5.1 Creating UI Policies

The procedure for adding files to an application in Studio is the same regardless of file type:

1. Click the **Create Application File** link.
2. Choose the file type, in this case, **UI Policy**.
3. Configure the new file.

1.5.1 UI Policy Configuration

UI Policies have two views: *Default* and *Advanced*. The fields in the UI Policy configuration are different depending on which View is selected. The *Advanced* view displays all the configuration fields. The *Default* view displays a subset of the fields.

Table: Form (table) to which the UI Policy applies.

Application: Identifies the scope of the UI Policy.

Active: Controls whether or not the UI Policy is enabled.

Short description: A short explanation of what the UI Policy does.

Order: If multiple UI Policies exist for the same table, use the *Order* field to set the order of evaluation of the UI Policy Conditions.

Condition: The condition(s) that must be met to trigger the UI Policy logic.

Global: If *Global* is selected the script applies to all views for the table. 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 that are part of the selected form view. If the *View* field is blank the script applies to the *Default* view.

On load: When selected, the UI Policy condition field is evaluated when a form loads in addition to when field values change. When not selected, the UI Policy Condition is evaluated only when field values change.

Reverse if false: Take the opposite action when the *Condition* field evaluates too *false*.

Inherit: When selected, executes the script for forms whose table is extended from the UI Policy's table.

DEVELOPER TIP: Enter a descriptive value in the short description field because UI Policies do not have a Name field. When debugging, identify UI Policies by the Short description field value.

If the *Condition* field does not have a value, the condition returns true, and the UI Policy logic will execute every time there is a change to a field value on the form.

The *Order* field sets the order of evaluation of UI Policy conditions for UI Policies for the same table. The order of evaluation is from the lowest number to the highest number (ascending order). By convention, *Order* field values are in round values of one hundred: 100, 200, 300 etc. This is not required.

DEVELOPER TIP: Avoid ordering UI Policies as 1, 2, 3, etc. Leave a gap between Order field values to make it possible to insert a new UI Policy into the existing line-up without re-ordering the existing UI Policies.

1.6 UI Policy Actions

UI Policy Actions are client-side logic in a UI Policy used to set three field attributes:

- *Mandatory*
- *Visible*
- *Read only*

Although you can use scripts to set these attributes using the *GlideForm (g_form)* API, UI Policy Actions do NOT require scripting to set the field attributes.

1.6.1 Creating UI Policy Actions

1. In Studio, create a UI Policy or open an existing UI Policy for editing.
2. Scroll to the *UI Policy Actions* related list. If creating a new UI Policy, the UI Policy must be saved before the *UI Policy Actions* related list is visible.
3. Click the **New** button.
4. Configure the UI Policy Action.
 - a. Select a *Field* name.
 - b. Set the *Mandatory*, *Visible*, or *Read-only* field values.

True:Apply the attribute to the field.

False:Do not apply the attribute to the field.

Leave alone:The attribute does not apply to the field.

5. To clear any existing value from the field, select the **Clear the field value** option.
6. Click the **Submit** button.

When the UI Policy condition tests *true*, the UI Policy Actions are applied. In the example, the *State* field is *Read only*. The *Mandatory* and *Visible* attribute values are not changed by the UI Policy Action.

What happens when the UI Policy condition tests *false*? There are two possible outcomes:

- No action is taken
- The opposite action is taken

1.6.2 How does ServiceNow know which to do? The decision is made by the *Reverse if false* option in the UI Policy trigger.

- If *Reverse if false* is selected (default), the opposite action is taken in the UI Policy Actions. If a field is mandatory (true), the field will no longer be mandatory (false). That is to say, attributes which were true become false, and false become true. There are no changes to attributes which are set to Leave alone.
- If *Reverse if false* is not selected, no UI Policy Action logic is applied.

1.6.3 Creating UI Policy Related List Actions

Use UI Policy Related List Actions to show or hide related lists. The *Problem* form has multiple related lists:

- *Incidents*
- *Affected CIs*
- *Problem Tasks*
- *Change Requests*
- *Outages*
- *Attached Knowledge*

1. In Studio, create a UI Policy or open an existing UI Policy for editing.
2. Scroll to the **UI Policy Related List Actions** related list. If creating a new UI Policy, the UI Policy must be saved before the *UI Policy Related List Actions* related list is visible.
3. Click the **New** button.
4. Configure the UI Policy Related List Action.
 - a. Select a *List* name.
 - b. Set the *Visible* field value.

True:Apply the attribute to the related list.

False:Do not apply the attribute to the related list.

Leave alone:The attribute does not apply to the related list.

5. Click the **Submit** button.

The *Incidents* related list is hidden

1.6.4 UI Policy Scripts

UI Policy scripts use the client-side API to execute script logic based on whether the UI Policy condition tests true or false. Use UI Policy scripts to create complex conditional checks or to take actions other than setting field attributes (mandatory, read-only, or visible).

The scripting fields for UI Policies are only visible in the Advanced view. To enable the scripting fields, select the Run scripts option.

The Execute if true script executes when the UI Policy condition tests true.

The Execute if false script executes when the UI Policy condition tests false.

Developers can write scripts in one or both script fields depending on the application requirements.

Write script logic inside the onCondition function which is automatically inserted in the scripting fields. The onCondition() function is called by the UI Policy at runtime.

IMPORTANT: The Reverse if false option must be selected for the Execute if false script to run.

Although UI Policy Actions execute on all platforms, UI Policy Scripts execute on Desktop/tablet only in the default case. Use the Run scripts in UI type field to select the platforms for the UI Policy scripts.

1.7 Client Scripts vs. UI Policies

Client Scripts and UI Policies both execute client-side logic and use the same API. Both are used to manage forms and their fields. When developing an application, how can you decide which client-side script type to use? Use this table to determine which type is best suited to your application's needs:

Criteria	Client Script	UI Policy
Execute on form load	Yes	Yes
Execute on form save/submit/update	Yes	No
Execute on form field value change	Yes	Yes
Have access to field's old value	Yes	No
Execute after Client Scripts	No	Yes
Set field attributes with no scripting	No	Yes
Require control over order of execution	*Yes	Yes

*Although the Order field is not on the Client Script form baseline you can customize the form to add it.

UI Policies execute after Client Scripts. If there is conflicting logic between a Client Script and a UI Policy, the UI Policy logic applies.

1.8 Client-side Scripting Module Recap

Core Concepts:

- Client-side scripts execute script logic in web browsers
- Client-side scripts manage forms and form fields
- Client Scripts execute script logic when forms are:
 - Loaded
 - Changed
 - Submitted/Saved/Updated
- UI Policies have a condition as part of the trigger
- UI Policies can take different actions when conditions return true or false
- UI Policy Actions do not require scripting to set field attributes:
 - *Mandatory*
 - *Visible*
 - *Read only*

- UI Policy Related List Actions do not require scripting to show or hide related lists
- The *GlideForm* API provides methods for interacting with forms and form fields
- The *GlideUser* API provides methods and properties for accessing information about the currently logged in user and their roles

Module 2. Data Policy

Content:

2.1 Data Policy

2.2 Data Policy Rule

2.3 Using Data Policy as UI Policy

2.4 Data Policy vs UI Policy

2.5 Converting Data and UI Policy

In this module, you will learn to:

- Create Data Policies Rules to control mandatory and read-only states for a field
- Configure Data Policy conditions to control when to apply Data Policy Rules
- Determine when to use a Data Policy or a UI Policy
- Configure Data Policies to behave dynamically on a form with the Use as UI Policy on client configuration option
- Convert UI Policies to Data Policies and Data Policies to UI Policies

2.1 Data Policy

Data Policies enforce data consistency by setting mandatory and read-only field attributes. Unlike UI Policies, Data Policies execute server-side. UI Policy logic only applies to data entered in a form. Data Policy logic executes regardless of how a record changes. Developers cannot apply scripts to Data Policies.

Data policies are similar to UI policies, but UI policies only apply to data entered on a form through the standard browser. Data policies can apply rules to all data entered into the system, including data brought in through import sets or web services and data entered through the mobile UI.

For example, suppose that you are configuring a web service that allows users from outside the platform to update problems on the ServiceNow instance. Since these problems are not updated through the instance UI, they are not subject to the UI policies on the problem form. To ensure that the Close notes field is completed before a problem is marked Closed/Resolved, you can create a data policy that applies to server-side imports. Data that does not comply with this data policy produces an error. You can also apply the policy on the browser by selecting the Use as UI Policy on client check box in the data policy record.

Since UI policies can also manage the visibility of fields on a form, you may want to augment UI policies with data policies rather than replace them.

By default, data policies are applied to all GlideRecord operations including those used in Scripted REST APIs, and the REST Table API. You can opt out of applying the data policy to:

- Target records of SOAP web services
- Import sets
- Client-side UI policies

The admin role is required to edit data policie

2.1.1 Creating Data Policy

Procedure

1. Navigate to **Data Policies** by completing one of the following actions.
 - Navigate to **System Policy > Rules > Data Policies**.
 - From any form header, right-click the header bar and select **Configure > Data Policies**.

- In List v2, open any column context menu and select **Configure > Data Policies**.
 - In List v3, open the list title menu and select **Configure > Data Policies**.
2. Click **New**.
 3. Select any options for the data policy.
 4. Create the condition that must exist for the platform to apply this policy.

For example, your conditions might include **[Problem state] [is] [Closed/Resolved]**

5. Right-click the header and select **Save**.

The **Data Policy Rules** related list appears.

6. Click **New** in the related list and create the record that identifies the field and the policy to apply.

The Advanced View displays all of the trigger fields.

Table: Form (table) to which the Data Policy applies.

Application: Identifies the scope of the Data Policy.

Inherit: When selected, executes the Data Policy for tables that extend the Data Policy's table.

Apply to import sets: When selected, the Data Policy applies to data imported with a System Import Set or Web Service Import Set.

Reverse if false: Take the opposite action when the Condition field evaluates to false.

Apply to SOAP: If selected, the Data Policy applies to data imported with a SOAP web service.

Active: Controls whether the Data Policy is enabled.

Use as UI Policy on client: When selected, the Data Policy applies dynamically on a form. Selecting Use as UI Policy on client allows users to see mandatory fields on a record before submitting a form.

Short description: A brief explanation of what the Data Policy does.

Description: A detailed explanation of what the Data Policy does.

Conditions: The conditions that must be met to trigger the Data Policy logic.

DEVELOPER TIP: Enter a descriptive value in the Short description field because Data Policies do not have a Name field. When debugging, identify Data Policies by the Short description field value.

If the Conditions field does not have a value, the condition returns true and the UI Policy logic will execute every time there is a change to a field value on the form.

2.2 Data Policy Rules

Data Policy Rules are server-side logic in a Data Policy used to set two field attributes:

Mandatory

Read only

Data Policy Rules do not use scripting to set the field attributes.

2.2.1 Creating Data Policy Rules

In Studio, create a Data Policy or open an existing Data Policy for editing.

Scroll to the Data Policy Rules related list. If creating a Data Policy, the Data Policy must be saved before the Data Policy Rules related list is visible.

Click the New button.

Configure the Data Policy Rule.

Select a Field name.

Set the Read only or Mandatory field values.

True:

Make the field read-only or mandatory.

False:

Make the field editable or not mandatory.

Leave alone:

Make no change to the Read only or Mandatory field attribute.

Click the Submit button.

When the Data Policy Conditions evaluate to true, the Data Policy Rules are applied. In the example, the Data Policy Rule configures the Resolution code field to be Mandatory when the Data Policy conditions are met. The Data Policy is for Incident records where the State value is set to Closed or Resolved.

Configure the field and Read only and Mandatory field properties for the Data Policy Rule.

What happens when the Data Policy conditions test false? There are two possible outcomes:

No action is taken

The opposite action is taken

How does ServiceNow know which to do? The decision is made by the Reverse if false option in the Data Policy configuration.

If Reverse if false is selected (default), the opposite action is taken in the Data Policy Rules. If a Data Policy Rule makes a field mandatory (True), the field will no longer be mandatory (False). That is to say, attributes that were True become False, and False become True. There are no changes to attributes that are set to Leave alone.

If Reverse if false is not selected, no Data Policy Rule logic is applied when the conditions are not met.

2.3 Using Data Policy as UI Policy

In the previous exercise, the user needed to save the to see that the *Additional notes* field was mandatory. The *Use as UI Policy on client* option was not selected, so the change in the *Mandatory* field attribute Using Data Policy as UI Policy

In the previous exercise, the user needed to save the to see that the Additional notes field was mandatory. The Use as UI Policy on client option was not selected, so the change in the Mandatory field attribute was not reflected immediately in the UI.

When the Use as UI Policy on client option is not selected, the record needs to be submitted before the user knows the field is mandatory, and the record submission fails. In the example, the State is set to Resolved, but the form does not indicate that the Resolution code and Resolution notes fields are now mandatory.

When the Use as UI Policy on client is not selected, changing the State on an Incident to Resolved does not affect the form, even though the Resolution code field is now mandatory.

When the user saves the record, an error message indicates that the Resolution code and Resolution notes fields are mandatory and the fields are marked mandatory.

When the user saved the record, the Resolution code field is marked mandatory and error messages indicate that the mandatory field has not been populated.

When the Use as UI Policy on client option is selected on the Data Policy, the Resolution code and Resolution notes fields become mandatory on the form when the State is set to Resolved.

When the Use as UI Policy on client is selected, changing the State on an Incident to Resolved makes the Resolution code field mandatory in the UI immediately. was not reflected immediately in the UI.

When the *Use as UI Policy on client* option is not selected, the record needs to be submitted before the user knows the field is mandatory, and the record submission fails. In the example, the *State* is set to *Resolved*, but the form does not indicate that the *Resolution code* and *Resolution notes* fields are now mandatory.

When the user saves the record, an error message indicates that the *Resolution code* and *Resolution notes* fields are mandatory and the fields are marked mandatory.

When the *Use as UI Policy on client* option is selected on the Data Policy, the *Resolution code* and *Resolution notes* fields become mandatory on the form when the *State* is set to *Resolved*.

2.4 Data Policy vs. UI Policy

Both Data Policies and UI Policies enforce data consistency by setting field attributes based on conditions. Data Policies execute server-side logic. UI Policies execute client-side logic. When developing an application, which type of policy should a developer use? Use this table to determine which type of policy to use:

Criteria	Data Policy	UI Policy
Execute based on field values	Yes	Yes
Execute on form save/submit/update	Yes	No
Execute on form field value change	* Yes	Yes
Set field Mandatory attribute	Yes	Yes

Set field Read only attribute	Yes	Yes
Set field Visibility attribute	No	Yes
Set field attributes with no scripting	Yes	Yes
Execute scripts for advanced logic	No	Yes

* Data Policies can execute on a form value change if the Use as UI Policy on client configuration option is selected.

2.5 Converting Data and UI Policy

A Data Policy can be converted to a UI Policy and vice versa. Why would a developer need to convert?

The application has a Data Policy, but the policy logic needs to dynamically control field visibility on a form

The application has a Data Policy, but the policy logic requires a script

The application has a UI Policy, but the policy logic needs to apply when data is imported

The application has a UI Policy, but the policy logic needs to apply when data is updated by a Business Rule, flow, or other server-side application logic

Converting a Data Policy to a UI Policy

Data Policies can be converted to UI Policies. To convert a Data Policy to a UI Policy, click the Convert this to UI Policy Related Link on a Data Policy.

The Convert this to UI Policy Related Link.

The converted UI Policy record opens. The Data Policy no longer exists.

Converting a UI Policy to a Data Policy

UI Policies have configuration options not available to Data Policies, such as the ability to run scripts and the ability to hide or show fields with a UI Policy Action. UI Policies can be converted to Data Policies if:

The Run scripts option is false.

No UI Policy Actions set the Visibility for a field.

Use the Convert this to Data Policy Related Link to convert a UI Policy to a Data Policy. The Related Link is only visible when Run scripts is false and no UI Policy Actions configure field visibility.

The Convert this to Data Policy Related Link.

After clicking the Convert this to Data Policy Related Link, the converted Data Policy record opens. The UI Policy no longer exists.

Module 3: Server-Side Scripting

Content:

- 3.1 Business Rules
- 3.2 Dot Walking
- 3.3 Server-Side Scripting
- 3.4 Glide System
- 3.5 Glide Record
- 3.6 Glide Date Time
- 3.7 Script Include
- 3.8 On Demand Script Include
- 3.9 Extend Script Include
- 3.10 Extending GlideAjax
- 3.11 Utility Script Include
- 3.12 Other Server-Side Script Include
- 3.13 Module Recap

Introduction to Server-side Scripting

Scripts in ServiceNow fall into two categories:

- Client-side
- Server-side

This module is about server-side scripting. Server-side scripts execute on the ServiceNow server or database. Scripts in ServiceNow can do many, many things. Examples of things server-side scripts can do include:

- Update record fields when a database query runs
- Set field values on related records when a record is saved
- Manage failed log in attempts
- Determine if a user has a specific role
- Send email
- Generate and respond to events
- Compare two dates to determine which comes first chronologically
- Determine if today is a weekend or weekday
- Calculate the date when the next quarter starts
- Log messages
- Initiate integration and API calls to other systems
- Send REST messages and retrieve results

In this module you will learn to write, test and debug two types of server-side scripts:

- Business Rules
- Script Includes

3.1 Business Rules

Business Rules are server-side logic that execute when database records are queried, updated, inserted, or deleted. Business Rules respond to database interactions regardless of access method: for example, users interacting with records through forms or lists, web services, or data imports (configurable). Business Rules do *not* monitor forms or form fields but do execute their logic when forms interact with the database such as when a record is saved, updated, or submitted.

3.1.1 Business Rule Configuration

- **Name:** Name of the Business Rule.
- **Table:** Specifies the database table containing the records this logic will run against.
- **Application:** Name of the application the Business Rule is part of.
- **Active:** Enables/disables
- **Advanced:** Select to display all Business Rule configuration options.

When to run Section

- **When:** Select when the Business Rule logic executes relative to the database access.
- **Order:** Order of execution for Business Rules for the same table. Execute in ascending order. By convention, but not required, use *Order* values in round values of one hundred: 100, 200, 300, etc.
- **Insert:** Select to execute the Business Rule logic when new records are inserted into the database.
- **Update:** Select to execute the Business Rule logic when records are modified.
- **Delete:** Select to execute the Business Rule logic when records are deleted.
- **Query:** Select to execute the Business Rule logic when the database table is queried.
- **Filter Conditions:** Add a condition to the configuration such as *State* is *14*. The 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 run.

3.1.2. Controlling When Business Rules Run

The *When* option determines when, relative to database access, Business Rule logic executes:

- *before*
- *after*
- *async*
- *display*

IMPORTANT: Business Rules do NOT monitor forms. The forms shown in the graphics on this page represent a user interacting with the database by loading (reading) and saving (updating) records using a form.

Before:

Before Business Rules execute their logic before a database operation occurs. Use before Business Rules when field values on a record need to be modified before the database access occurs. Before Business Rules run before the database operation so no extra operations are required. For example, concatenate two fields values and write the concatenated values to the *Description* field.



After

After Business Rules execute their logic **immediately** after a database operation occurs and before the resulting form is rendered for the user. Use after Business Rules when no changes are needed to the record being accessed in the database. For example, use an

after-Business Rule when updates need to be made to a record related to the record accessed. If a record has child records use an after-Business Rules to propagate a change from the parent record to the children.



Async:

Like after Business Rules, async Business Rules execute their logic after a database operation occurs. Unlike after Business Rules, async Business Rules execute asynchronously. Async Business Rules execute on a different processing thread than before or after Business Rules. They are queued by a scheduler to be run as soon as possible. This allows the current transaction to complete without waiting for the Business Rules execution to finish and prevents freezing a user's screen. Use Async Business Rules when the logic can be executed in near real-time as opposed to real-time (after Business Rules). For example use async Business Rules to invoke web services through the REST API. Service level agreement (SLA) calculations are also typically done as async Business Rules.



To see async Business Rules queued up for execution, use the *All* menu in the main ServiceNow window (not Studio) to open **System Scheduler > Scheduled Jobs > Scheduled Jobs**. Look for Scheduled Job names starting with ASYNC. They go in and out of the queue very quickly and can be hard to catch on the schedule.

DEVELOPER TIP: Use async Business Rules instead of after Business Rules whenever possible to benefit from executing on the scheduler thread.

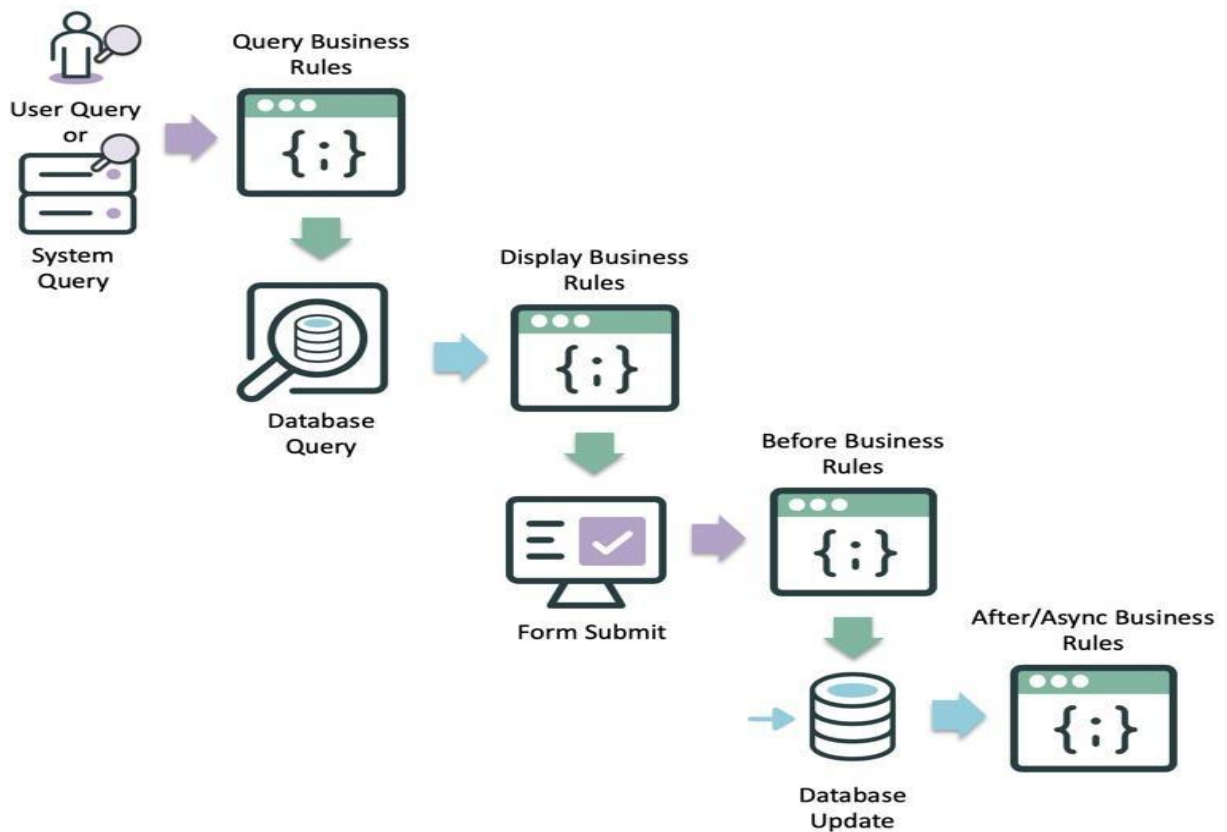
Display:

Display Business Rules execute their logic when a form loads and a record is loaded from the database. They must complete execution before control of the form is given to a user. The purpose of a display Business Rule is to populate an automatically instantiated object, *g_scratchpad*. The *g_scratchpad* object is passed from the display Business Rule to the client-side for use by client-side scripts. Recall that when scripting on the client-side, scripts only have access to fields and field values for fields on the form and *not* all of the fields from the database. Use the *g_scratchpad* object to pass data to the client-side without modifying the form. The *g_scratchpad* object has no default properties.



3.1.3 Business Rule Process Flow

A table can have multiple Business Rules of different *When* types. The order in which the Business Rules execute is:



3.1.4. Business Rule Actions

Business Rule Actions are a configurable way to:

- Set field values
- Add a message to a form
- Abort the Business Rule execution

The screenshot shows the 'Actions' tab in a configuration interface. The 'When to run' tab is selected, and the 'Actions' sub-tab is highlighted with a yellow box. Below the tabs, there are three options: 'Set field values', 'Add message', and 'Abort action'. The 'Set field values' option is expanded, showing a dropdown menu with '-- choose field --', a 'To' label, and a text input field with '-- value --'. The 'Add message' and 'Abort action' options have checkboxes next to them.

Set Field Values

The *Set field values* option allows setting field value without scripting. Values can be:

- Dynamically determined - *To (dynamic)*

- The same value as the value of another field - *Same as*
- Hard coded - *To*

The *dynamic* option is available only for *reference* fields.

Set field values	Requested for ▼	To (dynamic) ▼	Me ▼
	Description ▼	Same as ▼	as Short description ▼
	State ▼	To ▼	Awaiting Approval ▼
	-- choose field -- ▼	To	-- value --

In the example, the *Requested for* value is dynamically set to the currently logged in user as determined at runtime. The *Description* field has the same value as the *Short description* field. The *State* field is hard coded to the value *Awaiting Approval*.

Add Message

Use the *Add message* field to add a message to the top of a page. Although the message editor allows movies and images, only text renders on the pages. Use color, fonts, and highlighting effectively. The example text was chosen to demonstrate the types of effects which are available and should not be considered an example of effective styling.

Abort Action

The *Abort action* option stops execution of the Business Rule and aborts the database operation. When the *Abort action* option is selected, you can use the *Add Message* option

to print a message to the screen, but no other options are available. Use this option when the script logic determines the database operation should not be performed.

3.1.5. Business Rule Scripts

Business Rules scripts use the server-side APIs to take actions. Those actions could be, but are not limited to:

- Invoking web services
- Changing field values
- Modifying date formats
- Generating events
- Writing log messages

The *Advanced* option must be selected to write Business Rule scripts. The scripting fields are in the *Advanced* section.

Business Rule
New record

Name:

Application: ⓘ

Table:

Active: ☒

Advanced: ☒

When to run | Actions | **Advanced**

There are two fields for scripting in the Advanced section:

- *Condition*
- *Script*

current and previous

Business Rules often use the *current* and *previous* objects in their script logic.

The *current* object is automatically instantiated from the *GlideRecord* class. The *current* object's properties are all the fields for a record and all the *GlideRecord* methods. The property values are the *values as they exist in the runtime environment*.

The *previous* object is also automatically instantiated from the *GlideRecord* class. The *previous* object's properties are also all fields from a record and the *GlideRecord* methods. The property values are the values for the record fields *when they were loaded from the database and before any changes were made*. The *previous* object is not available for use in async Business Rules.

The syntax for using the *current* or *previous* object in a script is:

<object_name>.<field_property>

An example script using *current* and *previous*:

```
// If the current value of the description field is the same as when the
// record was loaded from the database, stop executing the script
if(current.description == previous.description){
    return;
}
```

```
}
```

Condition Field

Use the *Condition* field to write Javascript to specify when the Business Rule script should execute. Using the *Condition* field rather than writing condition logic directly in the *Script* field avoids loading unnecessary script logic. The Business Rule script logic only executes when the *Condition* field returns *true*. If the *Condition* field is empty, the field returns *true*.

There is a special consideration for async Business Rules and the *Condition* field. Because async Business Rules are separated in time from the database operation which launched the Business Rule, there is a possibility of changes to the record between when the condition was tested and when the async Business Rule runs. To re-evaluate async Business Rule *conditions* before running, set the system property, *glide.businessrule.async_condition_check*, to *true*. You can find information about setting system properties on the [ServiceNow docs site](#).

The Condition script is an expression which returns *true* or *false*. If the expression evaluates to *true*, the Business Rule runs. If the condition evaluates to *false*, the Business Rule does not run.

This is CORRECT syntax for a condition script:

```
current.short_description == "Hello world"
```

This is INCORRECT syntax for a condition script:

```
if(current.short_description == "Hello world"){}
```

Some example condition scripts:

The value of the *State* field changed from anything else to 6:

```
current.state.changesTo(6)
```

The *Short description* field has a value:

```
!current.short_description.nil()
```

The value of the *Short description* field is different than when the record was loaded:

```
current.short_description != previous.short_description
```

The examples use methods from the server-side API.

- The *changesTo()* method checks if a field value has changed from something else to a hardcoded value
- The *nil()* method checks if a field value is either *NULL* or the *empty string*

Notice that condition logic is a single JavaScript statement and does not require a semicolon at the end of the statement.

Script Field

The *Script* field is pre-populated with a template:

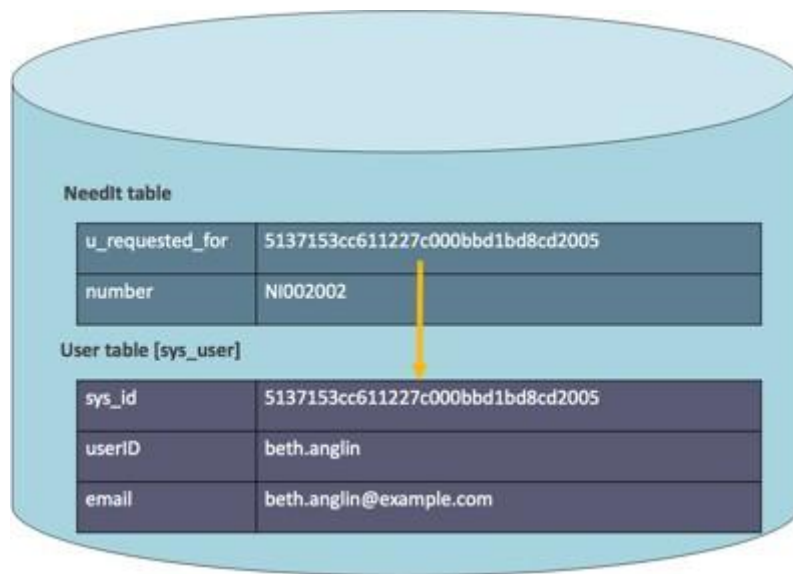


Developers write their code inside the *executeRule* function. The *current* and *previous* objects are automatically passed to the *executeRule* function.

Notice the template syntax. The function syntax is known in JavaScript as a self-invoking function or an Immediately Invoked Function Expression (IIFE). The function is immediately invoked after it is defined. ServiceNow manages the function and when it is invoked; developers do not explicitly call Business Rule scripts.

3.2 Dot-Walking

Dot-walking allows direct scripting access to fields and field values on related records. For example, the *NeedIt* table has a reference field called *Requested for*. The *Requested for* field references records from the *User [sys_user]* table. Reference fields contain the *sys_id* of the record from the related table.



When scripting, use dot-walking to retrieve or set field values on related records. The syntax is:

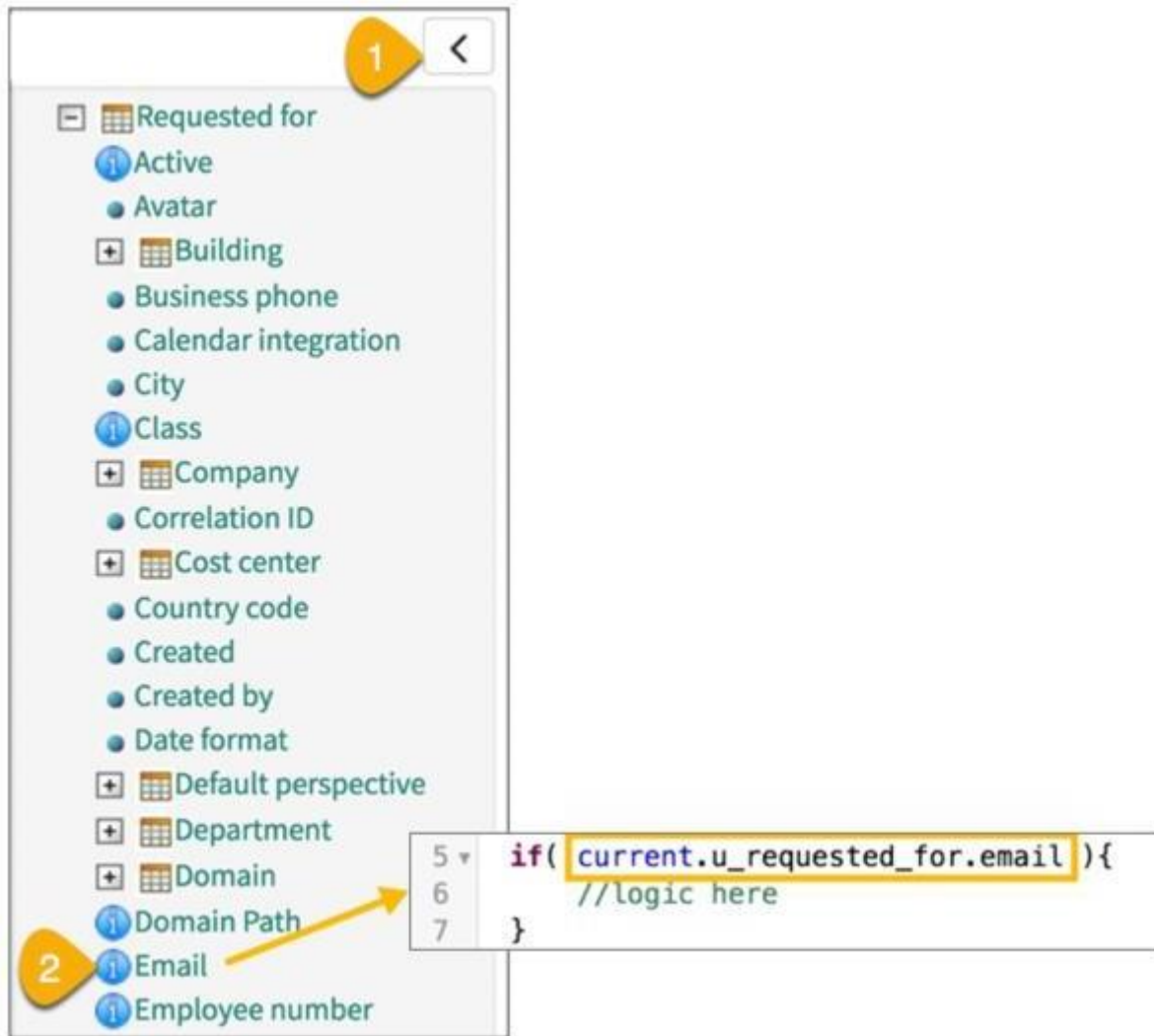
`<object>.<related_object>.<field_name>`

For example:

```
if(current.u_requested_for.email == "beth.anglin@example.com"){  
    //logic here  
}
```

The example script determines if the *NeedIt* record's *Requested for* person's email address is ***beth.anglin@example.com***.

To easily create dot-walking syntax, use the Script tree in the *Script* field:



1. Toggle the Script tree by clicking the **Script Tree** button.
2. Use the tree to navigate to the field of interest. Click the field name to create the dot-walking syntax in the script editor. The syntax starts from the *current* object.

Dot-walking syntax can be several levels deep. The example script finds the latitude for the company related to the user in the *Requested for* field.

```
current.u_requested_for.company.latitude
```

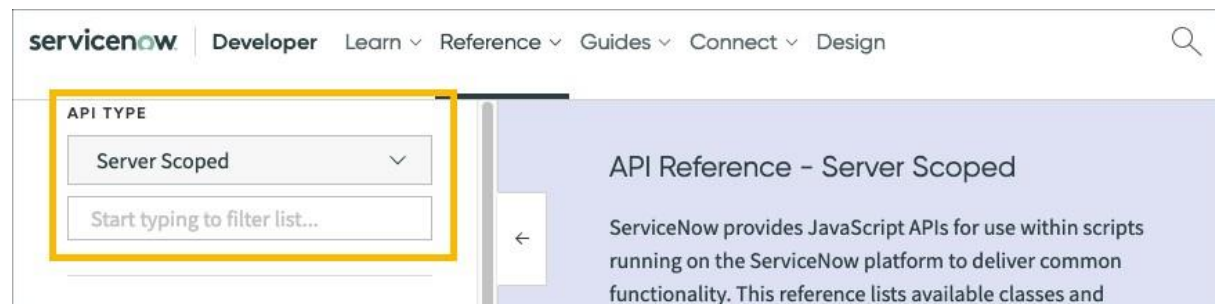
3.3 Server-side APIs

The complete documentation for the ServiceNow server-side APIs is available on the ServiceNow Developer Site. API documentation is release-specific. Use the Site Release Selector to set the release before viewing API documentation.



On the Developer Site, use the *Reference* menu to open API documentation. Use the *Server Scoped* API for scoped application scripts. Use the *Server Global* API for applications in the global scope. Some global APIs and methods do not have a scoped equivalent.

After opening API documentation, use the *API TYPE* field to switch APIs. Use the *Start typing to filter list...* field to look for a class or property within the API.



This module addresses some commonly used APIs:

- *GlideSystem*
- *GlideRecord*
- *GlideDateTime*

3.4 GlideSystem

Use the *GlideSystem* API to, for example:

- Find information about the currently logged in user
- Log messages (debug, error, warning, info)
- Add messages to pages
- Generate events
- Execute scheduled jobs
- And more...

See the *GlideSystem* [API](#) reference for a complete list of methods.

To use methods from the *GlideSystem* class, use the *gs* object:

`gs.<method>`

Examine the example script:

```
Script ? [Icons] >
1 (function executeRule(current, previous /*null when async*/ ) {
2
3     // Write an info level log message to log the Requested for's email address
4     gs.info("Requested for's email is: " + current.u_requested_for.email);
5
6     // Write an info message to the page showing the Requested for's employee number
7     gs.addInfoMessage("Requested for's employee number is " +
8     current.u_requested_for.employee_number);
9
10    // Write an info message to the page if the currently logged in user has the admin role
11    if (gs.hasRole("admin")) {
12        gs.addInfoMessage("Currently logged in user has the admin role.");
13    }
14 })(current, previous);
```

This sample script writes one message to the log and two messages to the screen:

App Log			
Created	▼	Search	
All > Created on Today			
<input type="checkbox"/>	🔍	Created ▼	Level Message
		-11-21 14:06:49	Information Requested for's email is: fred.luddy@example.com

**NeedIt
NI002002**

Requested for's employee number is 00001

Currently logged in user has the admin role.

3.5 GlideRecord

The *GlideRecord* class is the way to interact with the ServiceNow database from a script. See the *GlideRecord API* reference for a complete list of methods.

GlideRecord interactions start with a database query. The generalized strategy is:

1. Create a *GlideRecord* object for the table of interest.
2. Build the query condition(s).
3. Execute the query.
4. Apply script logic to the records returned in the *GlideRecord* object.

Here is what the generalized strategy looks like in pseudo-code:

```
/ 1. Create an object to store rows from a table
var myObj = new GlideRecord('table_name');

// 2. Build query
myObj.addQuery('field_name','operator','value');
myObj.addQuery('field_name','operator','value');

// 3. Execute query
myObj.query();

// 4. Process returned records
while(myObj.next()){
    //Logic you want to execute.
    //Use myObj.field_name to reference record fields
}
```

NOTE: The *GlideRecord* API discussed here is a server-side API. There is a client-side *GlideRecord* API for *global* applications. The client-side *GlideRecord* API cannot be used in scoped applications.

Build the Query Condition(s)

Use the *addQuery()* method to add query conditions. The *addQuery* operators are:

- **Numbers:** =, !=, >, >=, <, <=
- **Strings:** =, !=, STARTSWITH, ENDSWITH, CONTAINS, DOES NOT CONTAIN, IN, NOT IN, INSTANCEOF

The *addQuery()* method is typically passed three arguments: field name, operator, and value. In some scripts you will see only two arguments: field name and value. When the *addQuery()* method is used without an operator, the operation is assumed to be =.

When there are multiple queries, each additional clause is treated as an AND.

Queries with no query conditions return all records from a table.

If a malformed query executes in runtime, all records from the table are returned. For more strict query control you can enable the *glide.invalid_query.returns_no_rows* property which returns no records for invalid queries.

Iterating through Returned Records

There are several strategies for iterating through returned records.

The *next()* method and a *while* loop iterates through all returned records to process script logic:

```
// iterate through all records in the GlideRecord and set the Priority field value to 4 (low
priority).

// update the record in the database

while(myObj.next()){

    myObj.priority = 4;

    myObj.update();

}
```

The *next()* method and an *if* processes only the first record returned.

```
// iterate through all records in the GlideRecord and set the
Priority field value to 4 (low priority).

// update the record in the database

while(myObj.next()){

    myObj.priority = 4;

    myObj.update();

}
```

Use the *updateMultiple()* method to update all records in a *GlideRecord*. To ensure expected results with the *updateMultiple()* method, set field values with the *setValue()* method rather than direct assignment.

```
// When using updateMultiple(), use the setValue() method.

// Using myObj.priority = 4 may return unexpected results.

myObj.setValue('priority',4);

myObj.updateMultiple();
```

Counting Records in a GlideRecord

The *GlideRecord* API has a method for counting the number of records returned by a query: *getRowCount()*. Do not use the *getRowCount()* method on a production instance as there could be a negative performance impact on the database. To determine the number of rows returned by a query on a production instance, use *GlideAggregate*.

// If you need to know the row count for a query on a production instance do this

```
var count = new GlideAggregate('x_snc_needit_needit');

count.addAggregate('COUNT');

count.query();

var recs = 0;

if (count.next()){

    recs = count.getAggregate('COUNT');
```

```

}

gs.info("Returned number of rows = " + recs);

// Do not do this on a production instance.

var myObj = new GlideRecord('x_snc_needit_needit');

myObj.query();

gs.info("Returned record count = " + myObj.getRowCount());

```

Encoded Queries

As already discussed, if there are multiple conditions in query, the conditions are ANDed. To use ORs or create technically complex queries, use encoded queries. The code for using an encoded query looks like this:

```

var myObj = new GlideRecord("x_snc_needit_needit");

myObj.addEncodedQuery('<your_encoded_query>');

myObj.query();

while(myObj.next()){

    // Logic you want to execute for the GlideRecord records

}

```

The trick to making this work is to know the encoded query syntax. The syntax is not documented so let ServiceNow build the encoded query for you. In the main ServiceNow browser window, use the *All* menu to open the list for the table of interest. If there is no module to open the list, type *<table_name>.list* in the *Filter* field.

Use the Filter to build the query condition.

The screenshot shows the ServiceNow Filter interface for the 'Needits' table. The interface includes a search bar and a 'Run' button. Below the search bar, there are buttons for 'Run', 'Save...', 'AND', 'OR', 'Add Sort', and a star icon. The main area displays a list of conditions that must be met, separated by 'AND' and 'OR' operators. The conditions are:

- When needed: between Today and Last quarter
- Active: is true
- State: is Awaiting Approval
- or State: is Awaiting Feedback

Each condition is represented by a dropdown menu for the field, a dropdown for the operator, and a dropdown for the value. The 'AND' and 'OR' buttons are used to combine the conditions. The 'X' button is used to remove a condition.

Click the **Run** button to execute the query. Right-click the breadcrumbs and select the **Copy query** menu item. Where you click in the breadcrumbs matters. The copied query includes the condition you right-clicked on and all conditions to the left. To copy the entire query, right-click the condition farthest to the right.



Return to the script and paste the encoded query into the *addEncodedQuery()* method. Be sure to enclose the encoded query in "" or "".

```
var myObj = new GlideRecord("x_snc_needit_needit");

myObj.addEncodedQuery("u_when_neededBETWEENjavascript:gs.daysAgoStart(0)@javascript:gs.quartersAgoEnd(1)^active=true^state=14^ORstate=16");

myObj.query();

while(myObj.next()){

    // Insert logic you want to execute for the GlideRecord records

}
```


3.6 GlideDateTime

The scoped *GlideDateTime* class provides methods for performing operations on *GlideDateTime* objects. Use the *GlideDateTime* methods to perform date-time operations, such as instantiating a *GlideDateTime* object, performing date-time calculations, formatting a date-time, or converting between date-time formats. See the *GlideDateTime* [API](#) reference for a complete list of methods.

ServiceNow provides no default logic for managing dates in applications. The *NeedIt* application, for example, has a *When needed* field. There is no default logic preventing a user from setting the *When needed* date to a date in the past.

Applications that use dates may require script logic for the date fields. Examples include:

- Prevent users from selecting dates in the past
- Require start dates to be before end dates
- Disallow new requests to be submitted for today

When working with the *GlideDateTime* methods, pay attention to the date format and time zones. Some methods use GMT/UTC and some use local time zones. Some methods use the date in milliseconds and some do not.

3.7 Script Includes

Script Includes are reusable server-side script logic that define a function or class. Script Includes execute their script logic only when explicitly called by other scripts. There are different types of Script Includes:

- On demand/classless
- Extend an existing class
- Define a new class

Configuring a Script Include

Script Includes do not have many configuration options because they are called rather than triggered.

- **Name:** Name of Script Include.
- **API Name:** The internal name of the Script Include. Used to call the Script Include from out-of-scope applications.
- **Client callable:** Select this option if client-side scripts can call the Script Include using GlideAjax.
- **Application:** The application the Script Include is part of.
- **Caller Access:** When the *Scoped Application Restricted Caller Access* (com.glide.scope.access.restricted_caller) plugin is installed, allow scoped applications to restrict access to public tables and script includes.
 - **--None--:** No restrictions on access.
 - **Caller Restriction:** Do not allow access unless an admin approves access by the scope.

- **Caller Tracking:** Allow access but track which scopes access the Script Include.
- **Accessible from:** Choose *This application scope only* or *All application scopes*. Specifies the scope(s) that can use the Script Include.
- **Active:** Select 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.
- **Protection policy:** If set to *Read-only*, instances on which the application is installed from the ServiceNow Store can read but not edit the Script Include. If set to *Protected*, the Script Include is encrypted on instances on which the application is installed from the ServiceNow Store. Protection policies are never applied to the instance on which an application is developed.

3.8 On Demand Script Include

A Script Include that defines a single function is known as an on demand, or classless, Script Include. The function is callable from other server-side scripts. On demand Script Includes can never be used client-side even if the Client callable option is selected.

A script template is automatically inserted into the *Script* field. The template does not apply to on demand Script Includes. Delete the template and replace it with your function definition. The Script Include function is defined using standard JavaScript syntax. The example shows an on demand Script Include:

The screenshot shows the configuration page for a Script Include named 'sumTwoNums'. The 'Name' field is highlighted with a yellow box. An orange arrow points from this box to the 'Script' field, indicating that the function name must match the Script Include name. The 'API Name' is 'x_58872_needit.sumTwoNums'. The 'Application' is 'NeedIt'. 'Caller Access' is '-- None --'. 'Accessible from' is 'This application scope only'. 'Client callable' is unchecked. 'Active' is checked. The description states: 'This on demand, classless Script Include is passed two integers. The integers are added together and the sum is logged.' The 'Script' field contains the following JavaScript code:

```
1 function sumTwoNums(int1, int2) {  
2   var mySum = int1 + int2;  
3   gs.addInfoMessage("The sum of " + int1 + " + " + int2 + " = " + mySum);  
4 }
```


The Script Include name must *exactly match* the name of the function. In the example, both the Script Include and the function are named *sumTwoNums*.

The on demand Script Include is usable by any other server-side script allowed by the *Accessible from* option. In the example, the Script Include is callable only from *NeedIt* application server-side scripts. The example Business Rule script uses the *sumTwoNums* on demand Script Include:

The screenshot shows a Business Rule script using the *sumTwoNums* Script Include. The script is as follows:

```
1 (function executeRule(current, previous /*null when async*/) {  
2  
3   var myNum1 = 7;  
4   var myNum2 = 35;  
5  
6   sumTwoNums(myNum1, myNum2);  
7  
8 })(current, previous);
```

Although the *sumTwoNums* function is not defined in the Business Rule script, the function exists because it is defined in the Script Include.

 The sum of 7 + 35 = 42



On demand Script Includes are typically used when script logic needs to be reused. Examples include standardizing date formats, enabling/disabling logging, and validating email addresses.

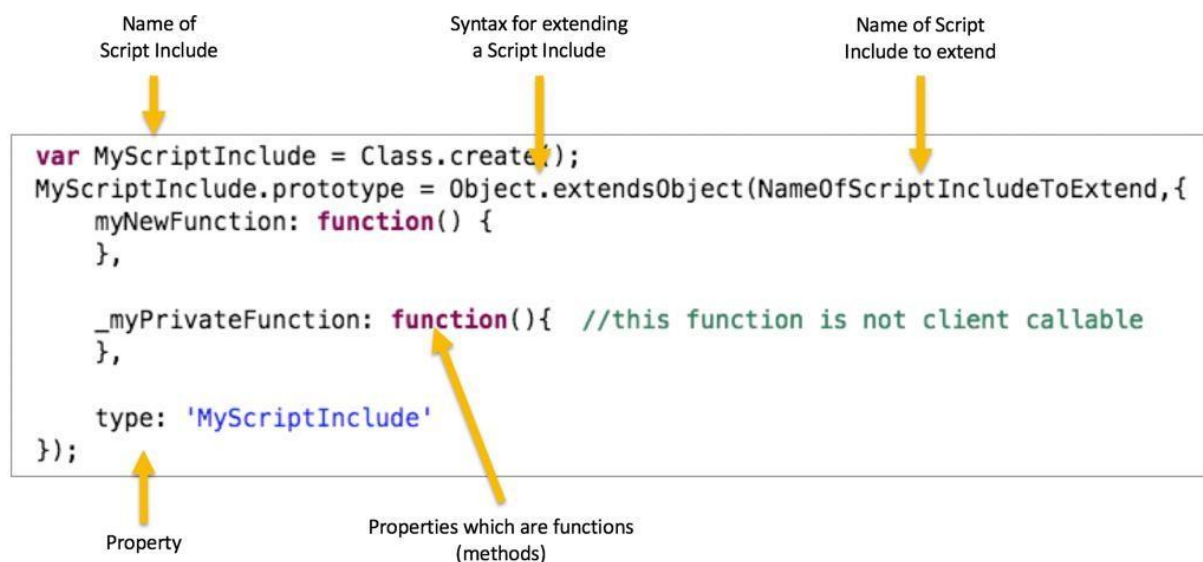
3.9 Extend a Script Include

Script Includes can extend existing Script Includes by adding new methods and non-method properties.

Although most ServiceNow classes are extensible, the most commonly extended classes are:

- **GlideAjax**: make AJAX calls from Client Scripts
- **LDAPUtils**: add managers to users, set group membership, debug LDAP
- **Catalog***: set of classes used by the Service Catalog for form processing and UI building

The generalized Script Include script syntax for extending a class is:



By convention, but not required, Script Include names start with an uppercase letter and are camel case thereafter. This type of capitalization is sometimes referred to as upper camel case. The Script Include name and the new class name must be an **exact match**.

If the class being extended is from another scope, prepend the class name with the scope. For example, if `NameOfClassYouAreExtending` is in the *global* scope, reference it as `global.NameOfClassYouAreExtending` in the scoped Script Include.

When creating a Script Include, a template is automatically inserted in the *Script* field:

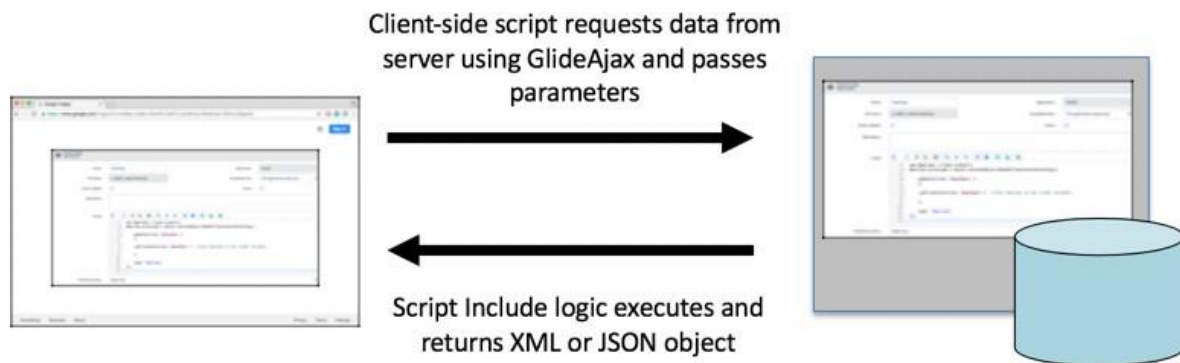
```
var NewClass = Class.create();
NewClass.prototype = {
  initialize: function() {
  },
  type: 'NewClass'
};
```

The Script Include template prototype must be modified when extending a Script Include.

Notice that the template includes an *initialize* function. When extending Script Includes, be cautious about overriding methods from the parent class such as the *initialize* function.

3.10 Extending GlideAjax

The *GlideAjax* class is used by client-side scripts to send data to and receive data from the ServiceNow server. The client-side script passes parameters to the Script Include. The Script Include returns data as XML or a *JSON* object. The client-side script parses data from the response and can use the data in subsequent script logic.



The *NeedIt* form has a reference field called *Requested for*. The *Requested for* field references a record on the *User [sys_user]* table. The *User* table has a column called *Email*, which stores a User's email address. Recall that client-side scripts have access only to data from the fields on a form. Although the *NeedIt* form has a field that references the *User* table, it does not have access to the *Requested for's* email address which is stored in the database. This example will extend the *GlideAjax* class. The new class will be passed a *sys_id* for the *User* table and will retrieve and pass back the user's email address.

GlideAjax is used by *g_form* methods like *getReference()*. The *getReference()* method is passed a *sys_id* and returns that record as a *GlideRecord*. The new class created in this demonstration is passed a *sys_id* and returns only an email address and not an entire record. Although the performance difference between returning an entire record and a single value may seem negligible, performance is based on all the calls occurring on a form and not a single call. If multiple scripts save time, the cumulative effect can be noticeable.

Script Include for Retrieving an Email Address

The Script Include must be client callable because it will be used by client-side scripts.

Script Include GetEmailAddress	
Name	GetEmailAddress
API Name	x_58872_needit.GetEmailAddress
Client callable	<input checked="" type="checkbox"/>
Application	Needit
Caller Access	-- None --
Accessible from	This application scope only
Active	<input checked="" type="checkbox"/>
Description	Script Include to return an email address. The calling client-side script passes a sys_id for a User table record.

The *AbstractAjaxProcessor* class is part of the *Global Scope*. The *GetEmailAddress* Script Include is in the *NeedIt* scope. To extend the *AbstractAjaxProcessor*, the class must be prepended by the scope: *global.AbstractAjaxProcessor*. The new class defines a method called *getEmail*.

```

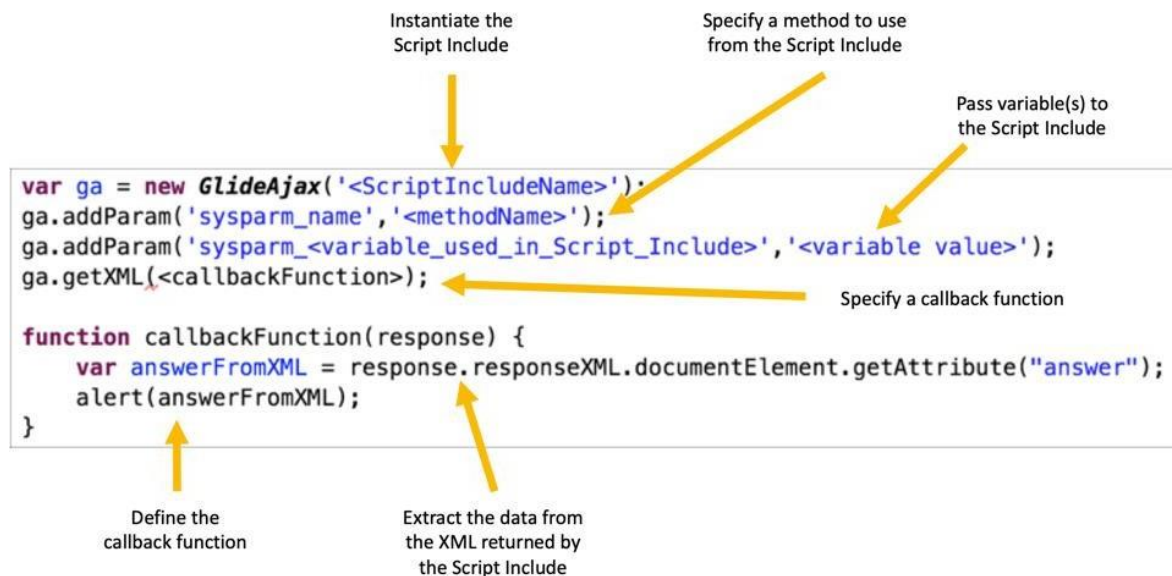
Script ? [Icons]
1  var GetEmailAddress = Class.create();
2  // Extend the global.AbstractAjaxProcessor class
3  GetEmailAddress.prototype =
   Object.extend(Object.prototype, {
4      // Define the getEmail function.
5      // Create a GlideRecord for the User table.
6      // Use the sysparm_userID passed from the client side to retrieve a
7      // record from the User table.
8      // Return the email address for the requested record
9      getEmail: function() {
10         var userRecord = new GlideRecord("sys_user");
11         userRecord.get(this.getParameter('sysparm_userID'));
12         return userRecord.email + '';
13     },
14     type: 'GetEmailAddress'
15 });

```

The syntax *this.getParameter('sysparm_<parameter_name>')* means to get the value of the parameter passed in from the client-side script. In this example, *sysparm_userID* contains the *sys_id* of a *User* table record.

Client-side Script for Using the GetEmailAddress Script Include

Any in-scope client-side script can use Script Includes which extend the *AbstractAjaxProcessor*. The generalized syntax is:

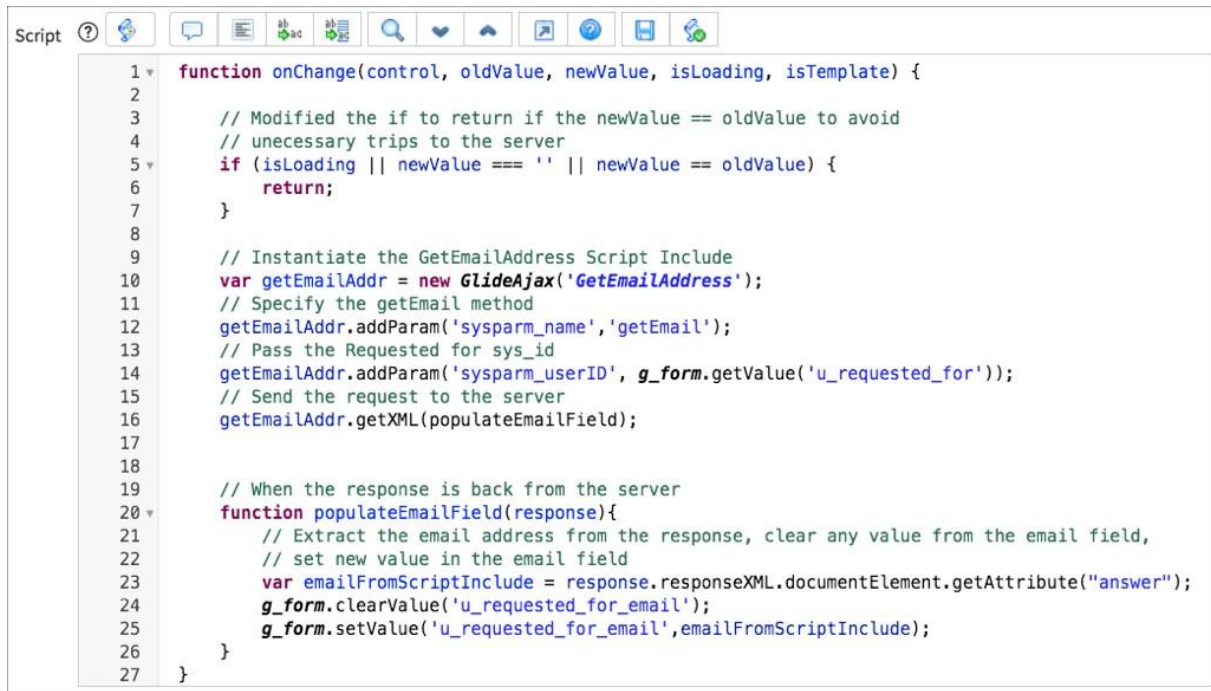


The first call to `addParam()` should be for the parameter `sysparm_name` and should pass as the value the name of the server-side method you want to call. Use `addParam` to pass user-defined parameters starting with `sysparm_` and their values. Users can create their own `sysparm_` variables except `sysparm_name`, `sysparm_type`, `sysparm_function`, and `sysparm_value`.

The `getXML` method sends the server a request to execute the method and parameters associated with this *GlideAjax* object. The server processes the request asynchronously and returns the results via the function specified as the callback function. This example, which uses a callback function, is asynchronous meaning that the user's screen is not frozen while the script waits for a response to come back from the server. Asynchronous *GlideAjax* is recommended over synchronous.

The callback function is passed the response back from the server. The script logic extracts the return value from the response. Subsequent script logic can use the extracted value.

The example shows the client-side logic for using the *GetEmailAddress* Script Include:



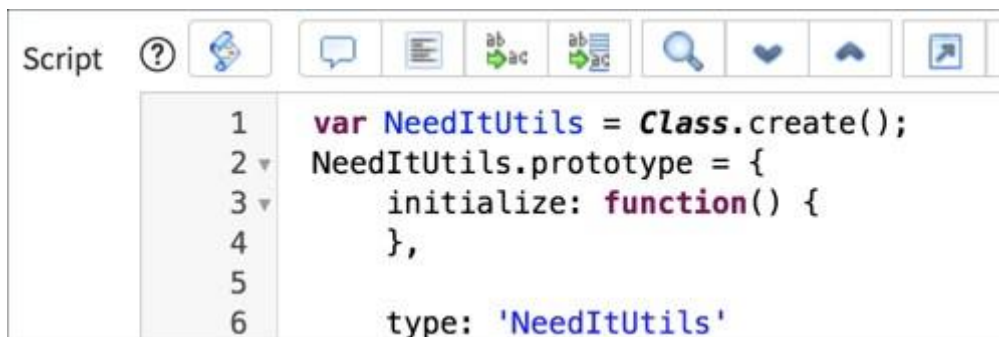
```
1  function onChange(control, oldValue, newValue, isLoading, isTemplate) {
2
3      // Modified the if to return if the newValue == oldValue to avoid
4      // unnecessary trips to the server
5      if (isLoading || newValue === '' || newValue == oldValue) {
6          return;
7      }
8
9      // Instantiate the GetEmailAddress Script Include
10     var getEmailAddr = new GlideAjax('GetEmailAddress');
11     // Specify the getEmail method
12     getEmailAddr.addParam('sysparm_name', 'getEmail');
13     // Pass the Requested for sys_id
14     getEmailAddr.addParam('sysparm_userID', g_form.getValue('u_requested_for'));
15     // Send the request to the server
16     getEmailAddr.getXML(populateEmailField);
17
18
19     // When the response is back from the server
20     function populateEmailField(response){
21         // Extract the email address from the response, clear any value from the email field,
22         // set new value in the email field
23         var emailFromScriptInclude = response.responseXML.documentElement.getAttribute("answer");
24         g_form.clearValue('u_requested_for_email');
25         g_form.setValue('u_requested_for_email', emailFromScriptInclude);
26     }
27 }
```

For information about returning multiple values or *JSON* objects, see the [Return multiple values from *GlideAjax*](#) article on the ServiceNow community site.

3.11 Utilities Script Include

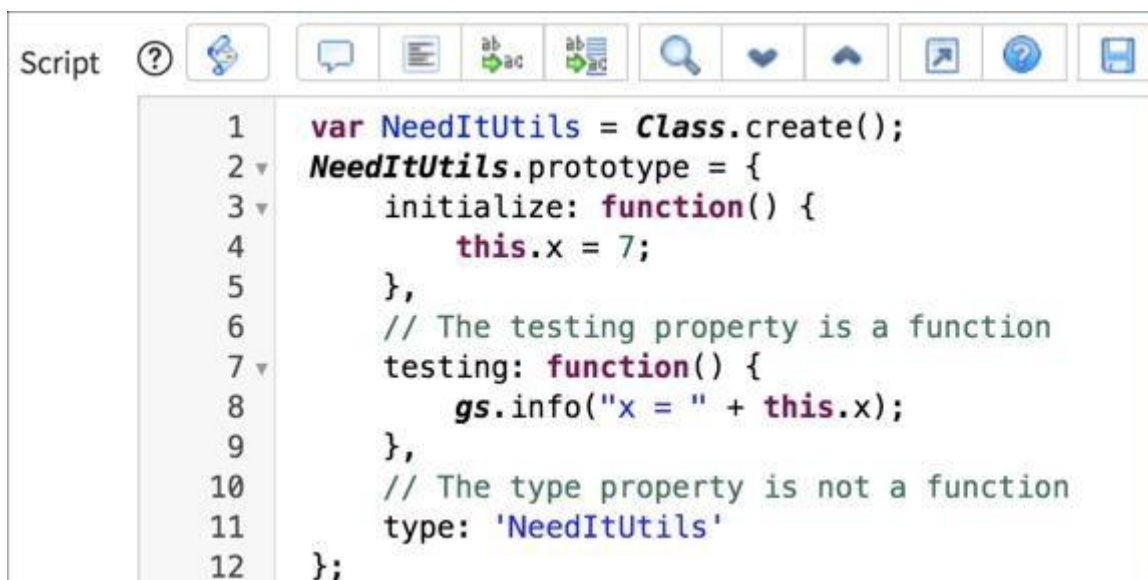
Although not required, many applications have one or more Script Includes to store the majority of the application's server-side logic. If there is a single Script Include for an application, it is often named *<App Name>Utils*. For example, *NeedItUtils*. If a Script Include becomes long and hard to manage, consider breaking it up into multiple Script Includes based on functionality or logical groupings.

Utilities Script Includes typically define a new class and therefore use the automatically inserted script template.










```
Script  ? [Icons]
1  var NeedItUtils = Class.create();
2  NeedItUtils.prototype = {
3    initialize: function() {
4    },
5
6    type: 'NeedItUtils'
```

The *initialize* function is automatically invoked when JavaScript objects are instantiated from the Script Include. Any variable defined as part of the *this* object in the *initialize* function is known to all other functions in the Script Include.















```
Script  ? [Icons]
1  var NeedItUtils = Class.create();
2  NeedItUtils.prototype = {
3    initialize: function() {
4      this.x = 7;
5    },
6    // The testing property is a function
7    testing: function() {
8      gs.info("x = " + this.x);
9    },
10   // The type property is not a function
11   type: 'NeedItUtils'
12 };
```

To use a Utils Script Include in other server-side scripts, instantiate the Script Include. Any script which instantiates the Script Include has access to the methods and non-method properties from the class.

Run this script ?            

```
1 // Instantiate the NeedItUtils Script Include
2 var niutil = new NeedItUtils();
3 //Invoke the testing method from the NeedItUtils Script Include
4 niutil.testing();
```

The value of x is written to the Application Log.

	App Log	New	Search	Created	▼	Search
		All > Created on Today				
		 Created ▼	 Level	 Message	 App Scope	 Source Script
		2020-11-24 17:38:45	Information	$x = 7$	NeedIt	Script Include: NeedItUtils

3.13 Other Server-side Script Types

In this module you have learned to write, test, and debug Business Rules and Script Includes. There are many other types of server-side scripts. The primary difference between the script types is what triggers the script logic execution.

The table shows some commonly used server-side script types.

Script Type	Executes on	Description	Often used to
Business Rule	Database access	Execute logic when records are queried, updated, inserted, or deleted.	Validate data or set fields on other records in response to fields on the current record.
Script Include	Must be explicitly called	A library of reusable functions.	Validate format, retrieve shared records, and work with application properties.
Script Action	Events	Respond to an event.	Send email notifications or write logging information.
Scheduled Script Execution (also known as a Scheduled Job)	Time	Script logic executed on a time-based schedule.	Create reports: send daily, weekly, monthly, quarterly, and annual information. Execute script logic only on weekdays or weekends. Can also be run on demand so sometimes

Script Type	Executes on	Description	Often used to
			used for testing.
UI Actions	Users	Add buttons, links, and context menu items to forms and list to allow users to perform application-specific operations.	Enable users to perform actions such as navigating to another page, modifying records, or allowing operations such as saving.
Scripts - Background	admin users only (some instances require the <i>security_admin</i> role)	Execute server-side code on demand from a selectable scope. Scripts - Background should be used with caution because badly written scripts can damage the database.	Test scripts.
Fix Scripts	Application installation or upgrade	Make changes that are necessary for the data integrity or product stability.	Create or modify groups or user authorizations.
Notification Email Script	Notification	Execute when emails are generated to add content to the email content or configuration.	Add a CC or BCC email address, or query the database and write information to

Script Type	Executes on	Description	Often used to
			the message body.
Scripted REST APIs	Request sent or received through web services	Defines a web service endpoint	Return value(s) or a <i>JSON</i> object based on a calculation or database lookup(s)
UI Page Processing Script	Users	Executes when a UI Page is submitted.	Validating data, setting values etc.
Transform Map Script	Data import	Modifies or copies data or data format when records are imported.	Standardize date formats, fill in missing data, standardize values, map incoming values to database values for choice lists, set default values.

You can practice using additional server-side script types in other courses and learning modules on the ServiceNow developer site.

3.13 Server-side Scripting Module Recap

Core Concepts:

- Server-side scripts execute on the ServiceNow server and have access to the database
- Business Rules are triggered by database operations: query, update, insert, and delete
- Server-side script APIs include:
 - *GlideRecord*
 - *GlideSystem*
 - *GlideDateTime*
- Business Rule script logic is executed relative to when the database operation occurs
 - before
 - after
 - async
 - display
- Debug Business Rules using:
 - Script Tracer - Determine which server-side scripts execute as part of a UI interaction
 - JavaScript Debugger - debug script logic
 - Debug Business Rule (Details) - debug condition script
 - Application log - view log messages
- Use the Script Tracer to find information about:
- State
- Script
- Transaction
- Use the JavaScript Debugger to debug synchronous server-side scripts
- See variable values
- Set breakpoints
- Set logpoints
- Use the Console to evaluate expressions in the runtime environment
- View call stack
- See transaction information
- Script Includes are reusable server-side logic
- On demand/classless Script Includes
 - Cannot be called from the client-side
 - Contain a single function
- Script Includes which extend a class
 - Inherit properties of extended class
 - Do not override inherited properties
 - Most commonly extended class is *GlideAjax*
- Script Includes which create a new class (does not extend an existing class)

- Many applications have a Utils Script Include
- Initialize function automatically invoked
- Developers must create all properties