

Plug-ins Guide



Copyright © 2005, 2018, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

If this document is in public or private pre-General Availability status:

This documentation is in pre-General Availability status and is intended for demonstration and preliminary use only. It may not be specific to the hardware on which you are using the software. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to this documentation and will not be responsible for any loss, costs, or damages incurred due to the use of this documentation.

If this document is in private pre-General Availability status:

The information contained in this document is for informational sharing purposes only and should be considered in your capacity as a customer advisory board member or pursuant to your pre-General Availability trial agreement only. It is not a commitment to deliver any material, code, or functionality,

and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle Master Agreement, Oracle License and Services Agreement, Oracle PartnerNetwork Agreement, Oracle distribution agreement, or other license agreement which has been executed by you and Oracle and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Sample Code

Oracle may provide sample code in SuiteAnswers, the Help Center, User Guides, or elsewhere through help links. All such sample code is provided "as is" and "as available", for use only with an authorized NetSuite Service account, and is made available as a SuiteCloud Technology subject to the SuiteCloud Terms of Service at www.netsuite.com/tos.

Oracle may modify or remove sample code at any time without notice.

No Excessive Use of the Service

As the Service is a multi-tenant service offering on shared databases, Customer may not use the Service in excess of limits or thresholds that Oracle considers commercially reasonable for the Service. If Oracle reasonably concludes that a Customer's use is excessive and/or will cause immediate or ongoing performance issues for one or more of Oracle's other customers, Oracle may slow down or throttle Customer's excess use until such time that Customer's use stays within reasonable limits. If Customer's particular usage pattern requires a higher limit or threshold, then the Customer should procure a subscription to the Service that accommodates a higher limit and/or threshold that more effectively aligns with the Customer's actual usage pattern.

Beta Features

Oracle may make available to Customer certain features that are labeled "beta" that are not yet generally available. To use such features, Customer acknowledges and agrees that such beta features are subject to the terms and conditions accepted by Customer upon activation of the feature, or in the absence of such terms, subject to the limitations for the feature described in the User Guide and as follows: The beta feature is a prototype or beta version only and is not error or bug free and Customer agrees that it will use the beta feature carefully and will not use it in any way which might result in any loss, corruption or unauthorized access of or to its or any third party's property or information. Customer must promptly report to Oracle any defects, errors or other problems in beta features to support@netsuite.com or other designated contact for the specific beta feature. Oracle cannot guarantee the continued availability of such beta features and may substantially modify or cease providing such beta features without entitling Customer to any refund, credit, or other compensation. Oracle makes no representations or warranties regarding functionality or use of beta features and Oracle shall have no liability for any lost data, incomplete data, re-run time, inaccurate input, work delay, lost profits or adverse effect on the performance of the Service resulting from the use of beta features. Oracle's standard service levels, warranties and related commitments regarding the Service shall not apply to beta features and they may not be fully supported by Oracle's customer support. These limitations and exclusions shall apply until the date that Oracle at its sole option makes a beta feature generally available to its customers and partners as part of the Service without a "beta" label.

Table of Contents

Custom Plug-in Overview	1
Custom Plug-in Creation	3
Custom Plug-in Development	4
Creating a Custom Plug-in Interface	4
Creating a Custom Plug-in Default Implementation	5
Adding the Default Implementation to NetSuite	6
Instantiating a Custom Plug-in Script in SuiteScript 2.0	7
Instantiating a Custom Plug-in Script in SuiteScript 1.0	9
Adding a Script that Instantiates a Custom Plug-in to NetSuite	12
Bundling a Custom Plug-in	12
Custom Plug-in Implementation	13
Creating a Custom Plug-in Alternate Implementation	13
Adding the Alternate Implementation to NetSuite	14
Managing Custom Plug-in Implementations	15
Core Plug-in Overview	16
Core Plug-in Creation	17

Custom Plug-in Overview

A custom plug-in is customizable functionality that is defined by an interface. When an interface is defined, third party solution providers can develop custom plug-in implementations and bundle them as part of a SuiteApp. After the SuiteApp is installed within an account, a solution implementer can define one or more alternate implementations. These implementations allow the solution implementer to customize the custom plug-in's logic to suit specific business needs.



Important: The object-oriented interface is central to the plug-in model. To be more exact, a plug-in is an interface. Plug-ins do not act as APIs. In other words, a plug-in does not expose a class's functions or objects. It allows a third party to override the logic defined within its default implementation.

A custom plug-in interface defines the function names, their parameters, and return types. Interfaces can be called within any third party server-side SuiteScript, with the exception of Mass Update SuiteScripts.



Note: Functions referenced in a custom plug-in interface can only be called within the custom plug-in implementation by the solution provider.

The solution provider defines a custom plug-in's default logic in a default implementation. If applicable, the solution provider may define one or more alternate implementations.



Note: Alternate implementations must be associated with a plug-in implementation type, a standard record type in NetSuite. When a solution provider releases a custom plug-in with alternate implementations, these alternate implementations cannot be edited by solution implementers.

When the custom plug-in's implementations are installed or defined within the user's account, the end-user's NetSuite administrator activates the implementation or implementations available to each account, and is then able to select which implementation to use. Whether a custom plug-in can use a single or multiple implementations at one time is dependent upon the design of the custom plug-in. When an implementation is active, function calls made within the custom plug-in script execute that implementation's logic.

Sample Use Case

Consider a SuiteApp that includes logic for calculating asset depreciation. The solution provider turns this functionality into a custom plug-in. The solution implementer then overrides the default functionality with one or more alternate implementations based on the specific accounting principles and business requirements of the end-user.

In an alternative scenario, the solution provider releases the custom plug-in with multiple implementations, by-passing the solution implementer role. The end-user's NetSuite administrator chooses (activates) which one to run based on the end-user's needs.

What is a Custom Plug-in Type?

A custom plug-in type is a record type within NetSuite used to encapsulate a custom plug-in's implementations and any supporting library files. The name of a custom plug-in type is used to distinguish its implementations from the implementations of any other custom plug-in installed on an account.

The custom plug-in type is used in the following ways:

Solution Providers

- After a solution provider defines an interface's default implementation, they must create a custom plug-in type record for it.
- After a solution provider defines an alternate implementation, they must specify which custom plug-in type the alternate implementation belongs to. They do this by creating a new plug-in implementation record. The new plug-in implementation record is a child record to the custom plug-in type record created for the default implementation.


Solution Implementers

- After a solution implementer defines an alternate implementation, they must specify which custom plug-in type the alternate implementation belongs to. They do this by creating a new plug-in implementation record. The new plug-in implementation record is a child record to the custom plug-in type record created for the default implementation.


Administrators

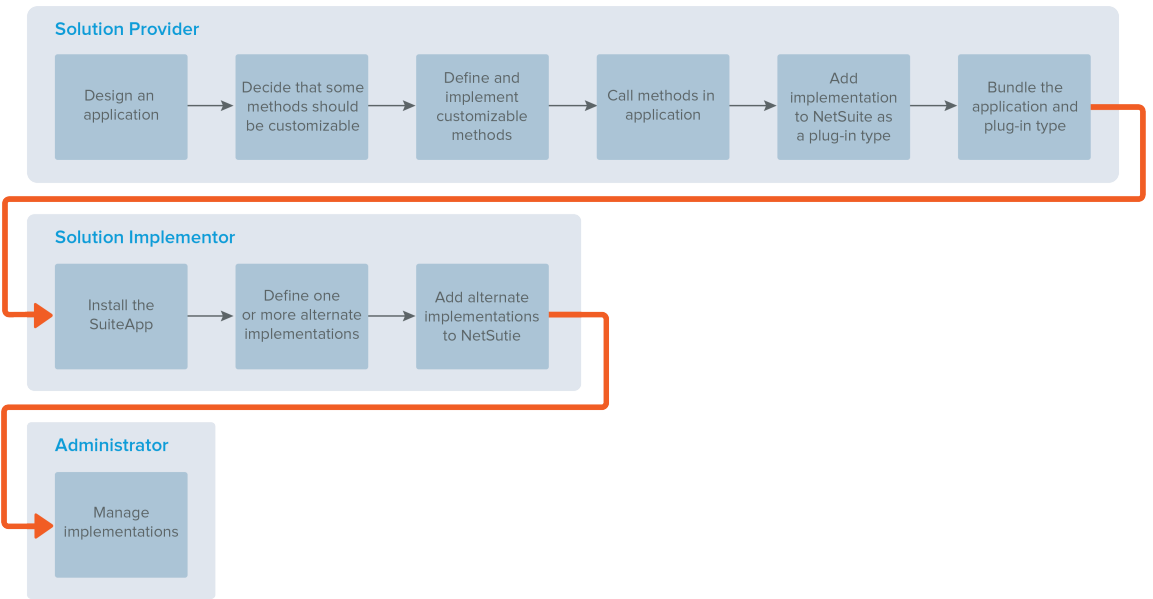
- The Management Implementations page is used by administrators to manage all plug-ins installed on an account. Administrators use the custom plug-in type name to distinguish a plug-in's implementations from the implementations of other custom plug-ins installed on an account. For more information, see the help topic [Managing Plug-ins](#).

Custom Plug-in Creation

**Important:** To create your own custom plug-ins and implementation scripts, you need some JavaScript coding experience and an understanding of SuiteScript. NetSuite recommends writing new custom plug-in scripts in SuiteScript 2.0. To get started with SuiteScript, see the help topics [SuiteScript 1.0](#) and [SuiteScript 2.0](#).

The following sequence illustrates the basic process for developing, implementing, and managing custom plug-ins. Note that your process flow may vary from the following diagram. See [Suggested Topics by Role](#) for a list of help topics organized by role.

**Note:** This diagram is organized by role, not title. Roles do not always equate to titles. For example, the solution implementor and administrator roles may be assigned to the same individual or group.



Suggested Topics by Role

Solution Provider	Solution Implementor	Administrator
<div>Custom Plug-in Development</div> <ul style="list-style-type: none">Creating a Custom Plug-in InterfaceCreating a Custom Plug-in Default ImplementationAdding the Default Implementation to NetSuiteIf you are creating alternative implementations:<ul style="list-style-type: none">Creating a Custom Plug-in Alternate ImplementationAdding the Alternate Implementation to NetSuiteInstantiating a Custom Plug-in Script in SuiteScript 2.0Adding a Script that Instantiates a Custom Plug-in to NetSuite	<div>Custom Plug-in Implementation</div> <ul style="list-style-type: none">Creating a Custom Plug-in Alternate ImplementationAdding the Alternate Implementation to NetSuite	<div>Managing Custom Plug-in Implementations</div> <div>Viewing Plug-in Implementation System Notes</div>

Solution Provider	Solution Implementor	Administrator
■ Bundling a Custom Plug-in		

Custom Plug-in Development

Important: To create your own custom plug-ins and implementation scripts, you need some JavaScript coding experience and an understanding of SuiteScript. NetSuite recommends writing new custom plug-in scripts in SuiteScript 2.0. To get started with SuiteScript, see the help topics [SuiteScript 1.0](#) and [SuiteScript 2.0](#).

Important: You cannot use the SuiteScript Debugger to ad hoc debug a script that uses the N/plugin module. You must use deployed debugging. To use deployed debugging, you must complete the steps described in [Adding a Script that Instantiates a Custom Plug-in to NetSuite](#). For additional information on ad hoc and deployed debugging, see the help topic [Using the SuiteScript Debugger](#).

As a solution provider, the following sections must be reviewed in the following order.

- [Creating a Custom Plug-in Interface](#)
- [Creating a Custom Plug-in Default Implementation](#)
- [Adding the Default Implementation to NetSuite](#)
- If you are creating alternative implementations:
 - [Creating a Custom Plug-in Alternate Implementation](#)
 - [Adding the Alternate Implementation to NetSuite](#)
- [Instantiating a Custom Plug-in Script in SuiteScript 2.0](#)
- [Adding a Script that Instantiates a Custom Plug-in to NetSuite](#)
- [Bundling a Custom Plug-in](#)

Creating a Custom Plug-in Interface

The interface is central to the custom plug-in model. A custom plug-in's interface defines functions that are executed within the custom plug-in script. A custom plug-in script can be any type of server-side SuiteScript other than a Mass Update script. Note that client-side SuiteScripts cannot act as custom plug-in scripts.

Important: Functions defined in a custom plug-in's interface are only ever called within the custom plug-in scripts's code.

The functions in an interface are not fully defined. Each function includes a signature (the function name and parameters) and a return type, but no body.

An implementation fully defines each of the interface's functions. In other words, an implementation contains the logic executed by the interface's functions. You must define a default implementation of the interface. If needed, you can also define one or more alternate implementations of the interface.



Important: Each implementation must keep the signature and return type defined in the interface.

Creating a Custom Plug-in Default Implementation



Important: To create your own custom plug-ins and implementation scripts, you need some JavaScript coding experience and an understanding of SuiteScript. NetSuite recommends writing new custom plug-in scripts in SuiteScript 2.0. To get started with SuiteScript, see the help topics [SuiteScript 1.0](#) and [SuiteScript 2.0](#).

An implementation fully defines an interface's functions. In other words, an implementation contains the logic executed by an interface's functions. When developing a plug-in, you must define a default implementation to define the custom plug-in script's default logic. A default implementation is written as an independent JavaScript file.



Note: Each function defined within the interface must be fully defined within the default implementation.

SuiteScript 2.0 Custom Plug-in Default Implementation Example

The following is a SuiteScript 2.0 example of a **default implementation**:

```
/**
 * @NApiVersion 2.0
 * @NScriptType plugintypeimpl
 */
define(function() {
    return {
        doTheMagic: function(inputObj) {
            var operand1 = parseFloat(inputObj.operand1);
            var operand2 = parseFloat(inputObj.operand2);
            if (!isNaN(operand1) && !isNaN(operand2)) {
                return operand1 + operand2;
            }
        },
        otherMethod: function() {
            return 'sample';
        }
    }
});
```

SuiteScript 1.0 Custom Plug-in Default Implementation Example

The following is a SuiteScript 1.0 example of a **default implementation**:

```
function doTheMagic(inputObj) {
    var operand1 = parseFloat(inputObj.operand1);
    var operand2 = parseFloat(inputObj.operand2);
    if (!isNaN(operand1) && !isNaN(operand2)) {
        return operand1 + operand2;
    }
}
```

```
function otherMethod() {
    return 'sample';
}
```

This sample default implementation fully defines the `doTheMagic` function. When called within the custom plug-in script, this function accepts two numbers, `operand1` and `operand2`, and adds them together to return a result.

Adding the Default Implementation to NetSuite

To add a default implementation to NetSuite, you create a Custom Plug-in Type. A Custom Plug-in Type is a record type within NetSuite that encapsulates a custom plug-in's default implementation and any supporting library files. Attach the default implementation's JavaScript file to the Custom Plug-in Type record to save the record. See [Creating a Custom Plug-in Default Implementation](#) before performing the steps below.



Important: You must have Server SuiteScript enabled and SuiteScript permission to create a custom plug-in type record.



Note: Alternate implementations are also encapsulated with a record type in NetSuite, the New Plug-in Implementation record. The New Plug-in Implementation record is a child record to the Custom Plug-in Type record.

To create a custom plug-in type:

1. Go to Customization > Plug-ins > Custom Plug-in Types > New.
2. Select the script file that contains your default implementation, and then click **Create Custom Plug-in Type**.
3. On the custom plug-in type record, enter the following:
 - **Name:** Provide a user-friendly name for the custom plug-in type. This name is seen by solution implementors creating alternate implementations of the plug-in type. It is also seen by administrators enabling/disabling implementations of this type.
 - **ID:** Provide an internal ID for the custom plug-in type. If you do not provide an ID, NetSuite provides one for you after you click **Save**.
 - **Class Name (SuiteScript 1.0 scripts only):** Provide a class name to represent the custom plug-in. Format the class name in Pascal case (PascalCase). You, the solution developer, use the class name to instantiate the implementation you want to use in your custom plug-in script. It is very important that you enter a descriptive class name (for example, `DemoPlugInType`). See [Instantiating a Custom Plug-in Implementation](#) for additional information.



Note: The term “class name” is a misnomer. When you create a new instance of an implementation in your custom plug-in script, you are actually instantiating a delegate, not an object.


- **Deployment Model:** Specify how many custom plug-in type implementations an administrator can activate at one time.



Important: The Deployment Model setting you choose affects how you write your custom plug-in script. See [Instantiating a Custom Plug-in Implementation](#) for additional information.

The Deployment Model field provides the following options:


- **Allow Multiple:** indicates multiple implementations of the interface can be activated at the same time within an account.
- **Allow Single:** indicates only one implementation of the interface can be activated at any time within an account.

 **Note:** The Deployment Model field does not define how many implementations a custom plug-in can have. If this field is set to **Allow Single**, the custom plug-in can still have an unlimited number of alternate implementations. However, only one implementation can be activated at any time.

- **Status:** Set to either **Testing** or **Release**. Be sure to set the status to **Released** before performing the steps in [Bundling a Custom Plug-in](#).
- **Log Level:** Set to the appropriate logging level you want for the custom plug-in script.
- **Description:** If you choose, provide a brief description of what the custom plug-in does.
- **Owner:** This field defaults to the name of the logged in user.
- **Inactive:** Set whether you want the custom plug-in type to be active or inactive.
- On the **Methods** tab, add the names of the functions defined in the interface. Only enter the function name. Do not enter parentheses or parameters.
- On the **Scripts** tab, add the following:
 - **Default Implementation (SuiteScript 1.0 scripts only):** Browse to the default implementation's JavaScript file. The lists only shows JavaScript files uploaded to the SuiteScript folder in the NetSuite file cabinet. You can, however, also attach your .js file from a local directory or by URL.
 - **Documentation:** Add documentation written for your default implementation here. It is **strongly recommended** that you write an interface definition describing the functions defined in your interface. Solution implementors need this information to create alternate implementations.
 - **Libraries (SuiteScript 1.0 scripts only):** Add library files that support your alternate implementation here.
- On the **Unhandled Errors** tab: Define which individual(s) will be notified if script errors occur. Note that notification is based on the errors that occur in your (the solution provider's) account. You are not going to be notified of errors that may occur in the accounts of those who install your custom plug-in.

4. Click **Save**.

Instantiating a Custom Plug-in Script in SuiteScript 2.0

 **Important:** To create your own custom plug-ins and implementation scripts, you need some JavaScript coding experience and an understanding of SuiteScript. NetSuite recommends writing new custom plug-in scripts in SuiteScript 2.0. To get started with SuiteScript, see the help topics [SuiteScript 1.0](#) and [SuiteScript 2.0](#).

To instantiate a custom plug-in script implementation in your code, use the `plugin.loadImplementation` function. For more information, see the help topic [plugin.loadImplementation\(options\)](#).

There are two ways to invoke a specific implementation:

- Using SuiteScript, pass the script implementation ID to the `implementation` parameter of the `plugin.loadImplementation` function.

- Using the **Manage Plug-ins** page, instantiate the implementation that is currently active in the end-user's account.



Important: To use the Script Debugger on a script that instantiates a specific implementation, you must add the script that instantiates a Custom Plug-in to NetSuite. For more information, see [Adding a Script that Instantiates a Custom Plug-in to NetSuite](#).

Instantiating a Specific Implementation using SuiteScript

Instantiate a specific implementation by passing the script implementation ID to the `implementation` parameter of the `plugin.loadImplementation` function. The ID for the default implementation is `default`. Use the `plugin.findImplementations` function to get a list of all the active script implementation IDs.

The following example shows a Suitelet script that performs these tasks:

- Use the `plugin.findImplementations` function to count the total number of active implementations and get a list of all the implementation IDs for a custom plug-in script type ID named `customscript_magic_plugin`.
- Traverse each implementation, instantiating them with the `plugin.loadImplementation` function and running `doTheMagic`, a function that is defined in each implementation.

```
/**
 * @NApiVersion 2.0
 * @NScriptType suitelet
 */
define(['N/plugin'], function(plugin) {
    return {
        onRequest: function(options) {
            var impls = plugin.findImplementations({
                type: 'customscript_magic_plugin'
            });
            for (i = 0; i < impls.length; i++) {
                var pl = plugin.loadImplementation({
                    type: 'customscript_magic_plugin',
                    implementation: impls[i]
                });
                options.response.write('impl = ' + impls[i] + ', result = ' + pl.doTheMagic({
                    operand1: 10,
                    operand2: 20
                }) + '\n');
            }
        }
    };
});
```

For more information about the `plugin.findImplementations` function, see the help topic [plugin.findImplementations\(options\)](#).

Instantiating a Specific Implementation using the Manage Plug-ins Page

To instantiate a specific implementation using the UI, refrain from specifying the `implementation` parameter in your `plugin.loadImplementation` call. When this parameter is not specified, NetSuite runs the active plug-in implementation that is specified by the **Manage Plug-ins** page.

The following example shows a Suitelet script that runs `doTheMagic`, a custom plug-in function that is defined in every implementation of the `customscript_magic_plugin` type. The `implementation` parameter is not specified.

```
/**
 * @NApiVersion 2.0
 * @NScriptType suitelet
 */
define(['N/plugin'], function(plugin) {
    return {
        onRequest: function(options) {
            var pl = plugin.loadImplementation({
                type: 'customscript_magic_plugin'
            });
            options.response.write('impl not specified, result = ' + pl.doTheMagic({
                operand1: 10,
                operand2: 20
            }) + '\n');
        }
    }
});
```

To specify which script implementation to run:

1. Go to Customization > Plug-ins > Manage Plug-ins.
2. Locate your custom plug-in name, and then select the desired active plug-in from the list of implementations.
3. Click Save.

Instantiating a Custom Plug-in Script in SuiteScript 1.0




Important: To create your own custom plug-ins and implementation scripts, you need some JavaScript coding experience and an understanding of SuiteScript. NetSuite recommends writing new custom plug-in scripts in SuiteScript 2.0. To get started with SuiteScript, see the help topics [SuiteScript 1.0](#) and [SuiteScript 2.0](#).

- [Instantiating a Custom Plug-in Implementation](#)
- [Calling a Default Implementation's Functions](#)
- [Calling an Alternate Implementation's Functions](#)
- [Calling Multiple Implementations' Functions](#)
- [Discovering Active Implementations](#)


Instantiating a Custom Plug-in Implementation

When you create a Custom Plug-in Type record in [Adding the Default Implementation to NetSuite](#), you enter a **Class Name** for the custom plug-in. Before you can call an implementation's functions, you must use the class name to create a new instance of the implementation you want to use.

 **Note:** The term “class name” is a misnomer. When you create a new instance of an implementation, you are actually instantiating a delegate, not an object.

Instantiate the implementation with the `new` keyword and the **Class Name** from the custom plug-in type record. The constructor has one optional parameter, `implementationId`. The argument you pass in determines which implementation you instantiate. See [Instantiating a Specific Implementation](#) and [Instantiating an Implementation Based on the Deployment Model](#) for the available arguments.

```
var a = new <Class Name>(implementationId);
```

 **Important:** Active implementations can only be instantiated after the plug-in is installed by an end-user and the plug-in script is executing.

There are two techniques you can use to instantiate an implementation:

- You can directly specify the implementation your code is instantiating.
- You can instantiate based on the **Deployment Model** setting on the Custom Plug-in Type record. In other words, you can instantiate the implementations that are currently active in the end-user's account.

Instantiating a Specific Implementation

Pass one of the following arguments for `implementationId` to instantiate a specific implementation.

- `default` – Pass this string argument to create an instance of the default implementation. Be sure to enclose the argument in single quotes. See [Calling a Default Implementation's Functions](#) for an example.
- The ID entered on the New Plug-in Implementation record – Pass this string argument to create an instance of a specific alternate implementation. Be sure to enclose the argument in single quotes. See [Calling an Alternate Implementation's Functions](#) for an example.

Instantiating an Implementation Based on the Deployment Model

Pass one of the following arguments for `implementationId` to instantiate the implementations that are currently active in the end-user's account. The argument you use depends on the value set for the **Deployment Model** field. See [Adding the Default Implementation to NetSuite](#) for additional information.

- No argument – If you do not pass an argument, the active implementation is instantiated. Use this option if the **Deployment Model** field is set to **Allow Single**.
- An array, returned by `getImplementations`, that contains all active implementations – Use this option if the **Deployment Model** field is set to **Allow Multiple**. You must use the `getImplementations` method to instantiate the implementations. See [Discovering Active Implementations](#) for additional information and an example.

Calling a Default Implementation's Functions

```
function useCustomPlugInType()
{
    var a = new DemoPlugInType('default');
    a.doTheMagic({
        operand1: 10,
```

```

        operand2: 20
    });
    a.otherMethod();
}

```

Calling an Alternate Implementation's Functions

```

function useCustomPlugInType()
{
    var b = new DemoPlugInType('customscriptimplementation1');
    b.doTheMagic();
    b.otherMethod();
}

```

Calling Multiple Implementations' Functions

In this example, you, the solution provider, create a default implementation and an alternate implementation for the plug-in. The alternate implementation is called by passing the ID entered on the New Plug-in Implementation record as an argument.

Note that the Deployment Model on the custom plug-in type record must be set to Allow Multiple.

```

function useCustomPlugInType()
{
    var a = new DemoPlugInType('default');
    a.doTheMagic({
        operand1: 10,
        operand2: 20
    });
    a.otherMethod();

    var b = new DemoPlugInType('customscriptimplementation1');
    b.doTheMagic();
    b.otherMethod();
}

```

Discovering Active Implementations

After your custom plug-in is installed on an end-user's account, the custom plug-in script has no direct knowledge of the active implementations. If you set the **Deployment Model** on the Custom Plug-in Type record to **Allow Single**, this is not an issue. See [Instantiating an Implementation Based on the Deployment Model](#) for additional information.

If **Deployment Model** field on the Custom Plug-in Type record is set to **Allow Multiple** and you do not specify which implementation to instantiate, you must use the `getImplementations` method to instantiate the implementations. The `getImplementations` method returns an array of active implementations.

```

function discoverImplementations()
{
    var x = DemoPlugInType.getImplementations();
}

```

```

for (var i = 0; i < x.length; i++)
{
    var t = new DemoPlugInType(x[i]);
    t.doTheMagic();
    t.otherMethod();
}
}

```

Adding a Script that Instantiates a Custom Plug-in to NetSuite

Create a script record for each script that instantiates your custom plug-in script. For more information, see the procedure described in [Steps for Creating a Script Record](#).

On the script record, you list the Custom Plug-in Type that represents your implementations.

To reference a custom plug-in type on a script record:

1. Click the **Scripts** tab. Then click the **Custom Plug-In Types** subtab.
2. In the Custom Plug-In Type field, select the Custom Plug-in Type that represents your plug-in's implementations.

Bundling a Custom Plug-in

Custom plug-in implementations are bundled as part of a SuiteApp. For installation instructions, see the applicable SuiteApp documentation. For more information about creating SuiteApps, see the help topic [SuiteBundler Overview](#).

To bundle a custom plug-in:

1. Go to Customization > SuiteBundler > Create Bundle.
2. Provide a name for your SuiteApp (and any other optional details).
3. On the second step of the Bundle Builder, note that you do not need to use the **Documentation** field to reference documentation written for the custom plug-in type.

The documentation that goes with your custom plug-in is already attached to the Custom Plug-in Type record, shown in [Adding the Default Implementation to NetSuite](#). Your documentation automatically gets bundled with the Custom Plug-in Type record.

4. In the next step of the Bundle Builder, click the **Plug-ins** folder.
 5. Next, click the **Custom Plug-in Types** folder.
 6. Under Choose Objects, select the name of the custom plug-in you want to include in your SuiteApp. The custom plug-in includes the default implementation.
- If you (as a solution provider) have developed alternate implementations of your custom plug-in, and you want to include these in the SuiteApp, select the **Plug-ins > Custom Plug-in Type Implementations** folder in the Bundler Builder. All of your implementations will appear in the Choose Objects column.

7. After bundling all other SuiteApp objects, click the **Next** button on the bottom of the screen.
8. In the Set Preferences step of the Bundler Builder, it is recommended that you click **Lock on Install** for all objects associated with your custom plug-in type.

Custom Plug-in Implementation



Important: To create your own custom plug-ins and implementation scripts, you need some JavaScript coding experience and an understanding of SuiteScript. NetSuite recommends writing new custom plug-in scripts in SuiteScript 2.0. To get started with SuiteScript, see the help topics [SuiteScript 1.0](#) and [SuiteScript 2.0](#).

Custom plug-in implementations are bundled as part of a SuiteApp. For installation instructions, see the applicable SuiteApp documentation. For more information about creating SuiteApps, see the help topic [SuiteBundler Overview](#).

As a solution implementer, the following sections must be reviewed in the following order.

- [Creating a Custom Plug-in Alternate Implementation](#)
- [Adding the Alternate Implementation to NetSuite](#)

Creating a Custom Plug-in Alternate Implementation

Review the custom plug-in type's default implementation and any documentation provided by the solution provider.

To download a copy of the default implementation:

1. Go to Customization > Plug-ins > Custom Plug-in Type.
2. Click **View** next to the custom plug-in type you want to work with.
3. On the **Methods** tab of this page, notice the names of the methods exposed in the default implementation. In your alternate implementation, you are required to use the same method names.
4. On the **Scripts** tab, click the **download** link next to the default implementation file. Review this file to understand the default logic for each method.
5. Click the **download** link next to the documentation file, if documentation has been provided by the solution provider. The documentation may provide additional information that explains what the methods are used for and what they return.

You can create a JavaScript file and re-implement the functions defined in the default implementation by creating an alternate implementation. The alternate implementation overrides the custom plug-in script's default logic. You cannot change the function signatures and return types defined in the default implementation. You can only override the logic within the function bodies. Each function that is defined in the default implementation must be fully defined in the alternative implementation.

SuiteScript 2.0 Custom Plug-in Alternate Implementation Example

The following is a SuiteScript 2.0 example of a **alternate implementation** to the SuiteScript 2.0 default implementation example shown in [Creating a Custom Plug-in Default Implementation](#):

```
/**
 * @NApiVersion 2.0
 * @NScriptType plugintypeimpl
 */
define(function() {
```

```

return {
  doTheMagic: function(inputObj) {
    return 1234;
  },
  otherMethod: function() {
    return 'sample';
  }
};

```

SuiteScript 1.0 Custom Plug-in Alternate Implementation Example

The following is a SuiteScript 1.0 example of a **alternate implementation** to the SuiteScript 1.0 default implementation example shown in [Creating a Custom Plug-in Default Implementation](#):

```

function doTheMagic(inputObj) {
  return 1234;
}
function otherMethod() {
  return 'sample';
}

```

This sample default implementation fully defines the `doTheMagic` function. When called within the a script, this function always returns a result of 1234. This is different from the default implementation, which accepts two numbers as parameters, adds them together, then returns the result.

Adding the Alternate Implementation to NetSuite

Alternate implementations are encapsulated by the New Plug-in Implementation record type. The New Plug-in Implementation record is a child record to the Custom Plug-in Type record. See [Creating a Custom Plug-in Alternate Implementation](#) before performing the steps in this procedure.

To create a New Plug-in Implementation record:

1. Go to Customization > Plug-ins > Plug-in Implementations > New.
2. Select the script file that contains your alternate implementation, and then click **Create Plug-in Implementation**.
3. On the New Plug-In Implementation record, enter the following:
 - **Name:** Provide a user-friendly name for your implementation. This name will appear on the Manage Plug-In Implementations page that administrators use to activate/deactivate the implementations in their account. See [Managing Custom Plug-in Implementations](#) for details.
 - **ID:** Provide an internal ID for the implementation. If you do not provide an ID, NetSuite provides one for you when you click **Save**.
 - **Status:** Set to either **Testing** or **Release**. Set the status to **Released** when you are ready to have the implementation accessible in a production environment.
 - **Log Level:** Set to the appropriate logging level you want for the script.
 - **Description:** If you choose, provide a brief description of what the alternate implementation includes.

- **Owner:** This field defaults to the name of the logged in user.
- **Inactive:** Set whether you want the custom plug-in type to be active or inactive.
- On the **Scripts** tab, enter the following:
 - **Implementation (SuiteScript 1.0 scripts only):** Select the script file that includes your alternate implementation of the custom plug-in type.
 - **Libraries (SuiteScript 1.0 scripts only):** If you have a library file that contains utility functions for your alternate implementation, add the library file(s) here.
- On the **Unhandled Errors** tab, define which individual(s) are notified if script errors occur. By default, the **Notify Script Owner** check box is selected.

4. Click **Save**.

You can access a list of all your implementations by going to Customization > Plug-ins > Manage Plug-ins.

Managing Custom Plug-in Implementations

As a NetSuite administrator, you may install a SuiteApp that includes a custom plug-in. If you do nothing, the logic within the custom plug-in's default implementation runs automatically in the context of the SuiteApp.

However, if the developers in your company choose to create alternate implementations of the custom plug-in, you use the Manage Plug-Ins page to enable one or more of their implementations. To access this page, go to Customization > Plug-ins > Manage Plug-ins.



Important: You must have Server SuiteScript enabled in your account to access to the Manage Plug-In Implementations page.

Custom plug-in types that allow multiple implementations to be enabled and deployed have a checkbox next to each alternate implementation.

Custom plug-in types that limit implementation deployments to one have a dropdown list. In this field you select only one implementation to enable and deploy. Note that there may be several implementations to choose from, but only one can be selected.

After enabling your implementations, click **Save**.



Important: The solution provider that created the custom plug-in type defines whether single or multiple deployments are supported. As the administrator, you select which implementation/deployment you want enabled in your NetSuite account.

Core Plug-in Overview

A plug-in is functionality, defined by an interface, that can be customized. After the plug-in is installed, a third party can override the plug-in's default logic with logic that suits its specific needs. The third party does this by defining alternate implementations of the interface



Important: The object-oriented interface is central to the plug-in model. To be more exact, a plug-in is an interface. Plug-ins do not act as APIs. In other words, a plug-in does not expose a class's functions or objects. It merely allows a third party to override the logic defined within its default implementation.

NetSuite develops core plug-ins and releases them, typically as part of a major release, to partners or customers. A core plug-in's interface defines functions that are executed within the core NetSuite code.



Note: Functions defined in a core plug-in's interface are only ever called within the core NetSuite code by core NetSuite developers.

NetSuite releases each core plug-in with a default implementation. Note that the logic defined in the default implementation may or may not be available to end users. If applicable, NetSuite may also release a core plug-in with one or more alternate implementations.

After a plug-in is installed within an account, the solution implementor can define one or more alternate implementations. These alternate implementations allow the solution implementor to customize the core plug-in's logic to suit specific needs. To accommodate this process, NetSuite provides an interface definition that describes the name, parameters, and return type of each function defined in the core plug-in's interface.



Note: Alternate implementations can only be edited by their owners. When NetSuite releases a core plug-in with alternate implementations, these alternate implementations cannot be edited by third parties.

When a plug-in's implementations are ready to be utilized, the end-user's NetSuite administrator activates the implementation or implementations available to each account. Whether a plug-in can use a single or multiple implementations at one time is dependent upon the design of the plug-in. When an implementation is active, function calls made within the core NetSuite code execute that implementation's logic.

Core Plug-in Creation

- [Available Core Plug-ins](#)
- [Creating a New Plug-in Implementation](#)
- [Debugging a Core Plug-in Implementation](#)

A plug-in is functionality defined through an interface. After a plug-in has been installed, a third party can replace the plug-in's default logic with logic that suits its specific needs. For more information about managing plug-ins, see the help topic [Managing Plug-ins](#)

Core plug-ins are developed and released by NetSuite. The logic defined in a core plug-in executes within the core NetSuite code. See [Core Plug-in Overview](#) for additional information.

Each core plug-in is released with documentation written specifically for that plug-in. Refer to your specific core plug-in's help for details needed to install, use and customize the plug-in. The [Available Core Plug-ins](#) table links to the help for each generally available core plug-in.

Available Core Plug-ins

Name	Description
	Adds custom general ledger postings on scriptable transactions.
Email Capture Plug-in	Sets up automated inbound email processing to trigger SuiteScript.

Creating a New Plug-in Implementation


The following is a generic procedure to create a new plug-in implementation in NetSuite. For details on how to create a new implementation for your specific core plug-in, see the applicable help topic listed under [Available Core Plug-ins](#).

To create a plug-in implementation, you create the plug-in implementation script file. Then, you upload the script file and other utility files as required to create the plug-in implementation.

To create a new plug-in implementation:

1. Review the interface description for your core plug-in.
2. Create a JavaScript file for the implementation. Use this file to define the logic for the methods listed in the interface description.
3. In the UI, click Customization > Plug-ins > Plug-in Implementations > New.
4. In the **Script File** field, open the script file or add a new file.
5. Click **Create Plug-in Implementation**.
6. On the Select Plug-in Type page, click the link for the plug-in type that you want to implement.
7. On the New Plug-in Implementation page, enter the following information.

Option	Description
Name	User-friendly name for the implementation. The plug-in implementation appears in the following locations: <ul style="list-style-type: none"> ■ Manage Plug-ins page. Page used by administrators to enable/disable the plug-in implementation in their account.

Option	Description
	<ul style="list-style-type: none"> Bundle Builder. Select this name in the Bundle Builder to distribute the plug-in implementation to other accounts.
ID	Internal ID for the implementation for use in scripting. If you do not provide an ID, NetSuite provides one for you when you click Save .
Status	Current status for the implementation. Choose Testing to have the implementation accessible to the owner of the implementation. Choose Released to have the implementation accessible to all accounts in a production environment.
Log Level	Logging level you want for the script. Select Debug , Audit , Error , or Emergency . These messages appear on the Execution Log subtab for the plug-in implementation.
Execute As Role	<p>Role that the script runs as. The Execute As Role field provides role-based granularity in terms of the permissions and restrictions of the executing script. The Current Role value indicates that the script executes with the permissions of the currently logged-in NetSuite user.</p> <div>  Note: You can create the custom role during testing to test the plug-in implementation with the proper role. The role requires the SuiteScript permission. You can then bundle the custom role to distribute it with the plug-in implementation. See Test the Plug-in Implementation and Bundle the Plug-in Implementation. </div>
Description	Optional description of the implementation. The description appears for the implementation on the Plug-In Implementations page.
Owner	User account that owns the implementation. Default is the name of the logged in user.
Inactive	Indicates the plug-in implementation does not run in the account. Inactivate a plug-in implementation, for example, to temporarily disable it for testing purposes.



Important: You cannot execute an implementation as Administrator. You must choose from one of the custom roles listed in the **Execute As Role** field. Note that System Administrator is a custom role.

8. On the **Scripts** subtab, in the **Implementation** list, change the JavaScript file that contains the implementation of the plug-in, if required.
9. On the **Scripts** subtab, in the **Library Script File** list, select any utility script files or supporting library files that are required by the plug-in script file..
10. On the **Unhandled Errors** subtab, specify who to notify if script errors occur.
 - To notify the user that is logged in and running the script, check the **Notify Current User** box.
 - To notify the script owner, check the **Notify Script Owner** box. The **Notify Script Owner** box is checked by default.
 - To notify all administrators, check the **Notify All Admins** box.
 - To notify a group about the error, select the group to notify. Only existing groups that were set up in are available.
 - Enter individual email addresses in the **Notify Emails** field. Separate multiple email addresses with a semi-colon.
11. Click **Save**.
12. Click **Configure** (not available for all plug-in types) and enter configuration settings as applicable. For example, see the help topic [Configure the Custom GL Lines Plug-in Implementation](#). Click **Save**.
13. Go to .

14. To enable the plug-in implementation, check the box and click **Save**.
15. Test the plug-in implementation.

You can access the list of implementations by going to Customization > Plug-ins > Plug-in Implementations.

Debugging a Core Plug-in Implementation

You can use the SuiteScript Debugger to test your core plug-in implementations.

Core plug-in implementations are debugged in the same way deployed scripts are debugged. To test your implementation in the Debugger, you must first create a Plug-in Implementation record and set the status to Testing. See [Creating a New Plug-in Implementation](#) and your specific core plug-in's help for additional information.

When you are ready to test your implementation, see the help topic [Deployed Debugging](#) for instructions.