ORACLE NETSUITE

SuiteScript 1.0

2023.1

June 7, 2023

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, timing, and pricing of any features or functionality described in this document may change and 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.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at https://www.oracle.com/corporate/accessibility.

Access to Oracle Support

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

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.

Send Us Your Feedback

We'd like to hear your feedback on this document.

Answering the following questions will help us improve our help content:

- Did you find the information you needed? If not, what was missing?
- Did you find any errors?
- Is the information clear?
- Are the examples correct?
- Do you need more examples?
- What did you like most about this document?

Click here to send us your comments. If possible, please provide a page number or section title to identify the content you're describing.

To report software issues, contact NetSuite Customer Support.

Table of Contents

SuiteScript 1.0 - The Basics

If you are new to SuiteScript 1.0, you should review these topics in order. You do not need to read every topic that is associated with each Overview, but you should at least read the Overviews to get a sense of how to use SuiteScript in your account.

1. SuiteScript Overview
2. SuiteScript 1.0 Script Creation, Deployment, and Logging
3. SuiteScript 1.0 API Overview
4. SuiteScript 1.0 Script Types Overview
5. Working with the SuiteScript Records Browser
6. Setting Up Your SuiteScript Environment

Throughout your SuiteScript development, you may refer to the SuiteBuilder Guide, a user guide that is available in the NetSuite Help Center. This guide provides detailed information about creating custom records, forms, fields, and sublists. In SuiteScript, much of what you will be doing is extending or getting/setting values on many of these custom elements. The SuiteBuilder Guide provides a basic understanding of how these elements are created and their relationship to one another. In the help center, see the help topic SuiteBuilder Overview to learn more about SuiteBuilder.

Important: If you are using SuiteScript 1.0 for your scripts, consider converting these scripts to SuiteScript 2.0. Use SuiteScript 2.0 to take advantage of new features, APIs, and functionality enhancements. For more information, see the help topic SuiteScript 2.x Advantages.

Additional Topics

After you understand the basic concepts behind SuiteScript 1.0 and how to run a script in NetSuite, see the following topics for details on how to maximize SuiteScript in your account. These topics do not need to be read in order. However, as you progress through SuiteScript development, you will refer to each section often:

- SuiteScript 1.0 Working with Records - Defines what a NetSuite record is as it pertains to SuiteScript. Also provides links to the Record APIs you will use when working with the entire record object.
- Working with Fields - Defines what a field is as it pertains to SuiteScript. Also provides links to the Field APIs you will use when working with fields on a record.
- Working with Subtabs and Sublists - Defines what a sublist is as it pertains to SuiteScript. Also provides links to the Sublist APIs you will use when working with sublists on a record.
- Setting Runtime Options - Provides additional information about the types of runtime options available on a Script Deployment record.
- Creating Script Parameters Overview- Defines the concept of "script parameters" as they pertain to SuiteScript. Also provides information about how to assign company, user, or portlet preference values to a script parameter.
- SuiteScript 1.0 Searching Overview - Explains how to search NetSuite using SuiteScript and the types of searches that are supported in scripting.
- UI Objects Overview - Explains the concept of UI objects and how these objects can be used in NetSuite to extend your application.
- SuiteScript Debugger- Describes how to use the SuiteScript Debugger to debug server-side SuiteScripts.

- SuiteScript Governance and Limits - Describes governance limits that are applied to each API and each SuiteScript type.

SuiteScript 1.0 Script Types

SuiteScript 1.0 Script Types Overview

Script types are organized primarily by where they run (on the client or on the server). They are also organized by the types of tasks you are trying to complete or the data you want to capture.

Use the SuiteScript API to create the following types of scripts.

- User Event Scripts: User Event scripts are triggered when users work with records and data changes in NetSuite as they create, open, update, or save records. User Event scripts are useful for customizing the workflow and association between your NetSuite entry forms. These scripts can also be used for doing additional processing before records are entered or for validating entries based on other data in the system.
- Suitelets: Suitelets enable the creation of dynamic web content. Suitelets can be used to implement custom front and backends. Through API support for scripting forms and lists, these Suitelets can also be used to build NetSuite-looking pages. NetSuite tasklinks can be created to launch a Suitelet. These tasklinks can be used to customize existing centers.
- RESTlets: RESTlets are server-side scripts that can be used to define custom, RESTful integrations to NetSuite. RESTlets follow the principles of the REST architectural style and use HTTP verbs, HTTP headers, HTTP status codes, URLs, and standard data formats. They operate in a request-response model, and an HTTP request to a system-generated URL invokes each RESTlet.
- Scheduled Scripts: Scheduled scripts are executed on-demand in real-time or by using a user-configurable schedule. Scheduled scripts are useful for batch processing of records.
- Client Scripts: Client scripts are executed on the client. These scripts can be attached to and run on individual forms, or they can be deployed globally and executed on entity and transaction record types. Global client scripts enable centralized management of scripts that can be applied to an entire record type.
- Portlet Scripts: Portlet scripts are used to create custom dashboard portlets. For example, you can use SuiteScript to create a portlet that is populated on-the-fly with company messages based on data within the system.
- Mass Update Scripts: Mass update scripts allows you to programmatically perform custom mass updates to update fields that are not available through general mass updates. You can also use action scripts to run complex calculations, as defined in your script, across many records.
- Workflow Action Scripts: Workflow action scripts allow you to create custom actions that are defined on a record in a workflow.
- Bundle Installation Scripts: Bundle installation scripts fire triggers that execute as part of bundle installation, update, or uninstall. Trigger execution can occur either before install, after install, before

update, after update, or before uninstall. These triggers automatically complete required setup, configuration, and data management tasks for the bundle.

> **Note:** SuiteBundler is still supported, but it will not be updated with any new features.
>
> To take advantage of new features for packaging and distributing customizations, you can use the Copy to Account and SuiteCloud Development (SDF) features instead of SuiteBundler.
>
> Copy to Account is an administrator tool that you can use to copy custom objects between your accounts. The tool can copy one object at a time, including dependencies and data. For more information, see the help topic Copy to Account Overview.
>
> SuiteCloud Development Framework is a development framework that you can use to create SuiteApps from an integrated development environment (IDE) on your local computer. For more information, see the help topic SuiteCloud Development Framework Overview.

Note that when you create a SuiteScript file, you will need to designate the type of script you want to write. You will do this by going to Setup > Customization > Scripts > New > [type], where **type** is one of the types shown below:

Select Type

TYPE	DESCRIPTION
Suitelet	Build interactive Web applications by scripting web requests
RESTlet	Build custom RESTful web services
User Event	Define business logic that is triggered when records are created, updated, viewed, or deleted
Scheduled	Schedule complex batch operations or queue them on-demand for execution
Client	Define business logic and perform client-side validation on your forms
Portlet	Publish scriptable portlets to your dashboards and centers
Mass Update	Perform an update to a record as part of a mass update
Workflow Action	Defines a custom action on a record that can be used as part of a workflow
Bundle Installation	Define scripts that run as part of bundle installation or update

To run a script in NetSuite, see the help topic SuiteScript 1.0 Script Creation, Deployment, and Logging.

Client Scripts

The following topics are covered in this section. If you are new to SuiteScript, these topics should be read in order.

- What is Client SuiteScript?
- Client Script Execution
- Client Event Types
- Form-level and Record-level Client Scripts
- Client Script Metering
- Role Restrictions in Client SuiteScript
- How Many Client Events Can I Execute on One Form?
- Error Handling and Debugging Client SuiteScript

What is Client SuiteScript?

Client scripts are SuiteScripts executed in the browser. They can run on most standard records, custom record types, and custom NetSuite pages (for example, Suitelets).

Note: To know which standard record types support client SuiteScript, see the help topic SuiteScript Supported Records in the NetSuite Help Center. If a record supports client scripts, an X will appear in the column called "Scriptable in Client SuiteScript".

Generally, client scripts are used to validate user-entered data and to auto-populate fields or sublists at various form events. Such events can include loading or initializing a form, changing a field, or saving a record. Another use case for client scripts is to source data from an external data source to a field. This is accomplished using the API nlapiRequestURL(url, postdata, headers, callback, httpMethod).

Client scripts are executed by pre-defined event "triggers." These triggering event types are discussed in the section Client Event Types. These events include:

- Initializing forms
- Entering or changing a value in a field (before and after it is entered)
- Entering or changing a value in a field that sources another field
- Selecting a line item on a sublist
- Adding a line item (before and after it is entered)
- Saving a form
- Searching for another record
- Loading, saving or deleting a record

After you have created your client script, you can attach your .js script file to the form you are customizing.

If you have created a client script that you want to execute across an entire record type (for example, all Customer records in the system), then you can deploy the script to the specified record type. Client scripts deployed globally affect the behavior of all the records they are deployed to, rather than a specific form on a single record.

You can use SuiteCloud Development Framework (SDF) to manage client scripts as part of file-based customization projects. For information about SDF, see the help topic SuiteCloud Development Framework Overview. You can use the Copy to Account feature to copy an individual client script to another of your accounts. Each client script page has a clickable Copy to Account option in the upper right corner. For information about Copy to Account, see the help topic Copy to Account Overview.

Client Script Execution

Client scripts are executed within a browser. Whether they are client scripts attached to individual forms, or client script deployed globally to an entire record type, all execution occurs in the browser.

Record-level (globally deployed) client scripts are executed **after** any existing form-based client scripts are run, and **before** any user event scripts. This means that record-level client scripts can run on both built-in and custom forms.

Note that there are some scripts that are considered to be client scripts, yet they make calls back to a NetSuite database. In this case you are working with records as "remote objects" on the client.

The following sample is a client script that has been attached to a Sales Order form. (For information about attaching client scripts to forms, see Step 3: Attach Script to Form.) When the user saves the Sales Order, the script gets the value of the **item** field on the Item sublist, then loads a specific Inventory Item record based on the value of the item in the Item sublist. The script then sets a value on the Inventory Item record and submits the record. Although there is backend activity being executed in this script, the script's initial execution is based on the saveRecord client event trigger (see Client Event Types), therefore it is still considered to be a client script.

```
// Client side script on sales order, on save
// Load the 1st item and mark it inactive
function onSave()
{
   var id = nlapiGetLineItemValue('item', 'item', 1);
   var record = nlapiLoadRecord('inventoryitem', id);
   record.setFieldValue('isinactive', 'T');
   nlapiSubmitRecord(record);

   return true;
}
```

Tip: You can set the order in which client scripts execute on the Scripted Records page. See the help topic The Scripted Records Page.

Client Event Types

In NetSuite, client scripts can be executed on 10 different client-side events. These client events can occur when a user loads a NetSuite form into the browser, or when a user selects a field or a field is updated. Field updates can occur when a user updates a field, or when a field is auto-updated through a sourcing relationship with another field. A client event can also occur when a user clicks the Submit or Save button on a NetSuite page.

The following table describes each client event type and the actions associated with the event.
Note that the functions that are called on each event type do not have to be written as pageInit() or fieldChanged(), and so on. However, when writing your client script, it is best practice to indicate the event type in the function name, for example: pageInit_alertSalesRep(), or validateField_department(), or saveRecordCustomer().

Client Event Type (and sample function name)	Parameters	Returns	Description
pageInit	*type* : the mode in which the record is being accessed. These modes can be set to: ■ create ■ copy ■ edit		This client event occurs when the page completes loading or when the form is reset. This function is automatically passed the **type** argument from the system. This is similar to an onLoad JavaScript client-side event. See PageInit Sample.
saveRecord		boolean	This client event occurs when the submit button is pressed but prior to the form being submitted. You should always return either **true** or **false** from a saveRecord event. A return value of **false** suppresses submission of the form. See Save Record Sample.
validateField	*type* : the sublist internal ID	boolean	This client event occurs whenever a field is about to be changed by the user or by a client side call.

Client Event Type (and sample function name)	Parameters	Returns	Description
	name : the field internal ID *linenum* : line number if this is a sublist. Line numbers start at 1, not 0.		Returning false from this function prevents the field's value from changing. This function is automatically passed up to three arguments by the system: **type, name**, **linenum**. This event type is similar to an onBlur JavaScript client-side event. In NetSuite, validateField events execute on fields added in beforeLoad user event scripts. Note: This event type does NOT apply to dropdown select or box fields. See ValidateField Sample.
fieldChanged	*type* : the sublist internal ID *name* : the field internal ID *linenum* : line number if this is a sublist. Line numbers start at 1, not 0.		This client event occurs whenever a field is changed by the user or by a client side call. This event can also occur directly through beforeLoad user event scripts. This client event does not occur when field information is changed or appears in the page URL. Use the pageInit event to handle URLs that may contain updated field values. This function is automatically passed up to three arguments by the system: **type, name**, **linenum**. This event type is similar to an onChange JavaScript client-side event. See Field Changed Sample.
postSourcing	*type* : the sublist internal ID *name* : the field internal ID		This client event occurs following a field change after all of the field's child field values are sourced from the server. Enables fieldChange style functionality to occur after all dependent field values have been set. This function is automatically passed up to two arguments from the system: **type, name.** See Post Sourcing Sample. **Note:** postSourcing functions will not execute when the Add Multiple button is used to add multiple line items.
lineInit	*type* : the sublist internal ID		This client event occurs when an existing line is selected. It can be thought of as the pageInit function for sublist line items (inlineeditor and editor sublists only). This function is automatically passed the **type** argument from the system.
validateLine	*type* : the sublist internal ID	boolean	This client event occurs prior to a line being added to a sublist (inlineeditor or editor sublists only). It can be thought of as the saveRecord equivalent for sublist line items (inlineeditor and editor). Returns false to reject the operation. References to fields should be done using nlapiGetCurrent*** functions.

Client Event Type (and sample function name)	Parameters	Returns	Description
			This function is automatically passed the **type** argument from the system.
recalc	*type* : the sublist internal ID		This event occurs after a sublist change, but only if the sublist change causes the total to change. This event is designed to be used for updating a global total, not for manipulating the current line item value. Do not call any getCurrentLineItem, setCurrentLineItem, or nlapiSelectLineItem functions for this event, they will not work. Instead use the validateLine event. Using the validateLine event, you can call the getCurrentLineItem, setCurrentLineItem, and selectLineItem functions. Notes: ■ recalc functions will not execute when the **Add Multiple** button is used to add multiple line items. ■ Scripts that execute on recalc events do not need to include a call to nlapiCommitLineItem(type) since recalculation occurs automatically. Calling nlapiCommitLineItem will calculate the line twice.
validateInsert	*type* : the sublist internal ID	boolean	The validateInsert event occurs when you insert a line into an edit sublist. For information about the edit sublist type, see Editor Sublists in the NetSuite Help Center. The UI equivalent of this event is when a user selects an existing line in a sublist and then clicks the Insert button. In SuiteScript, the equivalent action is calling nlobjRecord.insertLineItem(...). Note that returning false on a validateInsert blocks the insert.
validateDelete	*type* : the sublist internal ID	boolean	The validateDelete event occurs when you try to remove an existing line from an edit sublist. Returning false blocks the removal. For information about the edit sublist type, see Editor Sublists in the NetSuite Help Center.

*The ValidateField and FieldChanged scripts require a null line item for body fields.

Form-level and Record-level Client Scripts

Form-based client scripts run against specific fields and forms. Record-level client scripts, similar to user event scripts, are deployed globally and run against an entire record type. For example, record-level client scripts can be deployed to run against all Invoice records or all Customer records in the system.

Record-level client scripts run independent of any client scripts already attached to a specific form on the record. Record-level client scripts can also be used on forms and lists that have been generated through UI Objects during Suitelets development. Form-based client scripts cannot be used by Suitelets. Also note that fields that are hidden on a form are not accessible by client scripts.

Additionally, record-level clients scripts allow you to set audience definitions on the Script Deployment page. Defining an audience for the script deployment lets you specify which members of your company,

which departments, and which roles will be able to see the record-level customizations that have been implemented.

To deploy record-level client scripts into your account, you must follow the deployment model used for Suitelet, user event, scheduled, and portlet scripts. (See Running SuiteScript 1.0 in NetSuite Overview to learn how to run a record-level script in NetSuite.)

Form-level client scripts, however, require only that the client script is attached to the individual form it is running against. For details on attaching client scripts to forms, see Step 3: Attach Script to Form.

Note: You can deploy up to 10 record-level client script to any record types that are already supported in the existing form-based model. For information about records supported in client scripting, see the help topic SuiteScript Supported Records in the NetSuite Help Center.

Client Script Metering

Client scripts are metered or governed on a per-script basis. For example, if an account has one **form-level** client script attached to a form, and one **record-level** client script deployed to the record (which contains the form), **each** client script can total 1000 units. Units are not shared among the client scripts that are associated with a form or record.

Note: For information about script metering and the SuiteScript governance model, see the help topic SuiteScript Governance and Limits.

Role Restrictions in Client SuiteScript

Running a client script to access a record will result in an error if the role used does not have permission to view/edit that record. Client SuiteScript respects the role permissions specified in the user's NetSuite account.

Example

The following is a client script, which you can attach to a custom sales order form and set to execute on the field change client event.

```
function email(){
   var salesRep = nlapiGetFieldValue('salesrep');
   var salesRepEmail = nlapiLookupField('employee', salesRep, 'email');
   alert(salesRepEmail);
}
```

If you are logged in as an administrator, when you load the sales order with this form, and then select the Sales Rep field, you will receive the alert. However, if you log in using a non-administrator role (such as Sales Manager), or a role that does not have permission to view/edit Employee records, you will receive an error when you select the Sales Rep field.

To work around this issue, as the script developer you must consider the types of users who may be using your custom form and running the script. Consider which record types they do and do not have access to. If it is vital that all who run the script have access to the records in the script, you may need to redefine the permissions of the users (if your role is as an administrator). Or you may need to rewrite your script so that it references only those record types that all users have access to.

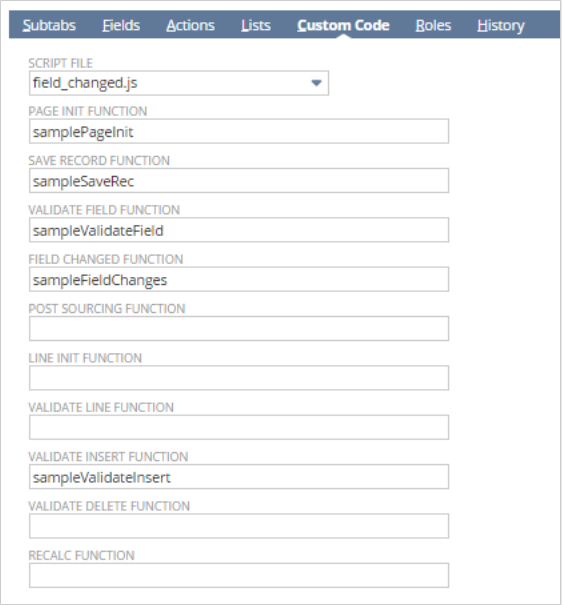
Another consideration is to write the script as a server-side user event script and set the “Execute As Admin” preference on the script's Script Deployment page. Note that in the sample script above, you would not be able to run the script as a user event script and throw an alert, as alerts are a function of client scripts only. However, you could rewrite the script so that it emails the user the sales rep's email address (instead of throwing an alert).

Note: For information about user event scripts, see User Event Scripts. For information about executing scripts as using an administrator role, see the help topic Executing Scripts Using a Specific Role.

How Many Client Events Can I Execute on One Form?

Client scripts have a limit of 10 client events per form. The following figure shows the Custom Code tab on a Custom Entry Form for a Customer record. If you choose, you can run a script that contains client event functions for all 10 available event types. This figure shows only five client event functions are specified within this script, but all 10 in one script file is supported.

For information about client event functions, see Client Event Types.



Note: For information about attaching a client script to a form and running the script in your account, see Step 3: Attach Script to Form.

Error Handling and Debugging Client SuiteScript

Important: You cannot debug form or record-level client scripts in the SuiteScript Debugger. To debug client scripts, you should use your browser's debugger. For instructions on working with either of these debuggers, please see the documentation provided with each product.

Regarding error handling in client SuiteScript, NetSuite catches and throws alerts for all unhandled SuiteScript errors that occur in both form- and record-level client scripts.

Note that alerts provide the scriptId of the Script record. This is information that will help NetSuite administrators locate the specific SuiteScript file that is throwing the error.

Also note that like other script types, the Script record page for **record-level** client scripts includes an Unhandled Errors subtab. NetSuite administrators can use this tab to define which individual(s) will be notified if script errors occur. For additional information about the Unhandled Errors subtab, see Steps for Creating a Script Record.

Additionally, the Script Deployment page for **record-level** client scripts includes an Execution Log subtab, on which all script errors are logged.

Client Remote Object Scripts

A client remote object script is a client-side SuiteScript that makes a call to the NetSuite server to create, load, copy, or transform a record object.

The following is an example of a client remote object script. On the saveRecord client event, the script executes the nlapiCreateRecord(...) function to create a new estimate record. This script creates the new Estimate, sets values on the new record, and then submits the record, all from within a script that is executed on the client.

Example

```
function onSave()
{
// access the NetSuite server to instantiate a new Estimate
var rec = nlapiCreateRecord('estimate');  rec.setFieldValue('entity','846');
rec.insertLineItem('item',1);
rec.setLineItemValue('item','item', 1, '30');
rec.setLineItemValue('item','quantity', 1, '500');

var id = nlapiSubmitRecord(rec, true);

return true;
}
```

Important: You cannot use client remote object scripts to access a remote object in dynamic mode. See SuiteScript 1.0 Client Scripting and Dynamic Mode for details.

Running a Client Script in NetSuite

To run a client script in NetSuite, you must:

1. Create a JavaScript file for your client script.
2. Load the file into NetSuite.
3. Attach your script file to a custom form (if you have written a form-level client script).
4. Create a Script record (if you have written a record-level client script).
5. Define all runtime options on the Script Deployment page (if you have written a record-level client script).

If you are new to SuiteScript and need information about each of these steps, see Running SuiteScript 1.0 in NetSuite Overview.

Client SuiteScript Samples

The following samples are covered in this section. They illustrate how client event functions are used to interact with the form.

- Writing Your First Client Script
- PageInit Sample
- Save Record Sample
- Post Sourcing Sample
- ValidateField Sample

- Field Changed Sample

Writing Your First Client Script

A great way to get started with client scripts is to deploy a script that has a function on every exposed event. Consider the following client script:

```
function myPageInit(type)
{
   alert ('myPageInit' );
   alert ('type=' + type);
}

function mySaveRecord()
{
   alert ('mySaveRecord' );
   return true;
}

function myValidateField(type, name, linenum)
{
   if (name === 'custentity_my_custom_field' )
   {
      alert ('myValidateField' );
      alert ('type=' + type);
      alert ('name=' + name );
      alert ('linenum=' + linenum);
     return true;
   }
}
function myFieldChanged(type, name, linenum)
{
   alert ('myFieldChanged' );
   alert ('type=' + type);
   alert ('name=' + name );
   alert ('linenum=' + linenum);
}

function myPostSourcing(type, name )
{
   alert ('myPostSourcing' );
   alert ('type=' + type);
   alert ('name=' + name );
}

function myLineInit(type)
{
   alert ('myLineInit' );
   alert ('type=' + type);
}

function myValidateLine(type)
{
   alert ('myValidateLine' );
   alert ('type=' + type);
   return true;
}
function myValidateInsert(type)
{
   alert ('myValidateInsert' );
   alert ('type=' + type);
   return true;

}
function myValidateDelete(type)
{
   alert ('myValidateDelete' );
   alert ('type=' + type);
   return true;
}
```

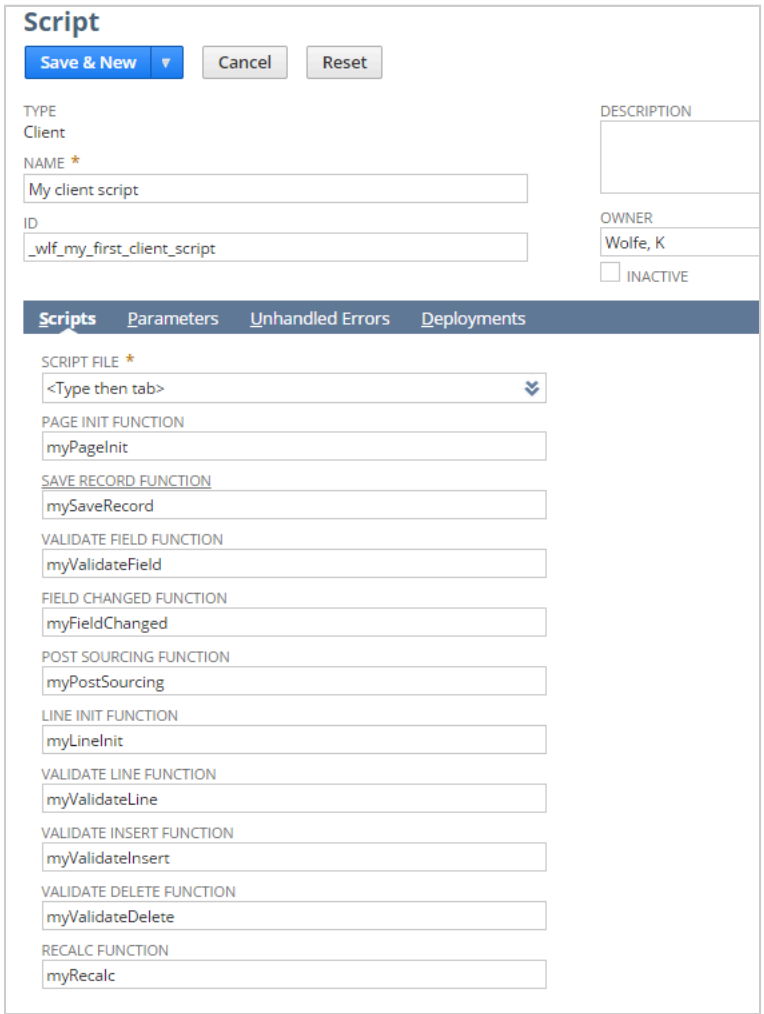
```
function myRecalc(type)
{
   alert ('myRecalc' );
   alert ('type=' + type);
}
```

This sample displays all the available arguments for every script-triggered event. Notice that some functions return a boolean whereas some do not. Also note that some functions have **linenum** as one of the arguments. Sublist functions do not have a **linenum** argument because the event is confined to the specific line that triggered it.

The function myValidateField has an additional **if** block to check whether the event was invoked by a custom field with the id custentity_my_custom_field . This ensures the logic is executed only under the correct circumstances.

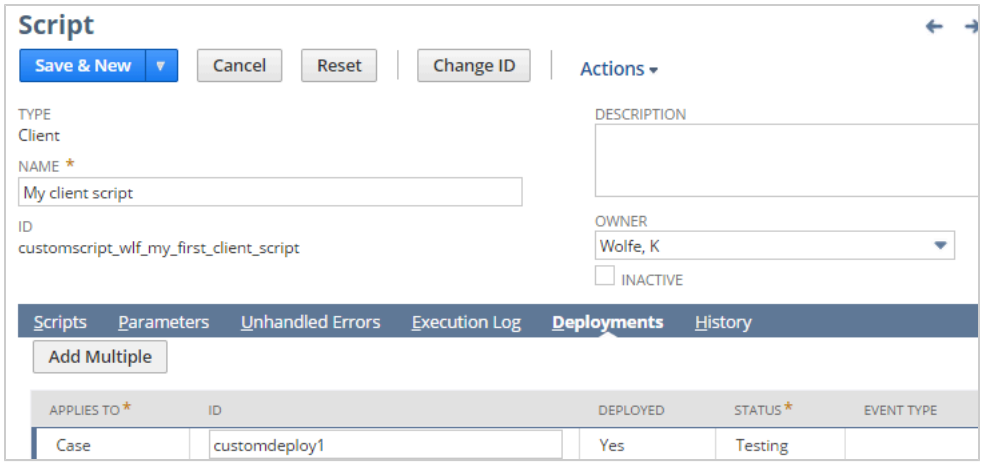
Note: It is important to check the argument values to branch execution logic. This improves performance and avoids logic executed indiscriminately.

To obtain a better understanding on when these client script events are triggered and what the arguments contain, upload the JavaScript file to the SuiteScript folder in the NetSuite File Cabinet, and deploy the script by specifying the functions in a script record. See the help topic Creating a Script Record.

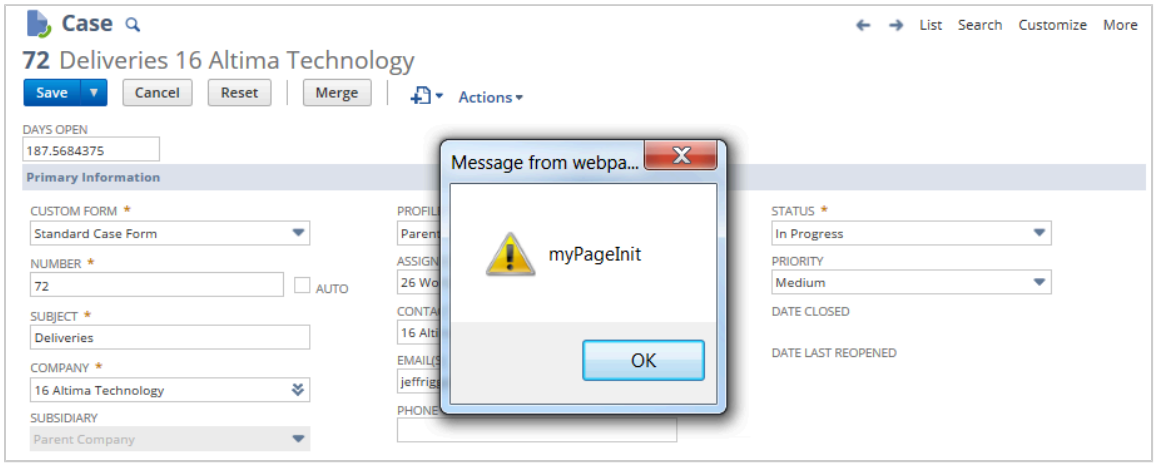


Note: When saved the Script record is saved, the system will automatically prefix the script ID with **customscript**. In the figure above, the final unique ID for this client script will be **customscript_wlf_my_client_script**. This custom script record identifier can be passed as a value to the **scriptId** parameter that is included in several SuiteScript APIs.

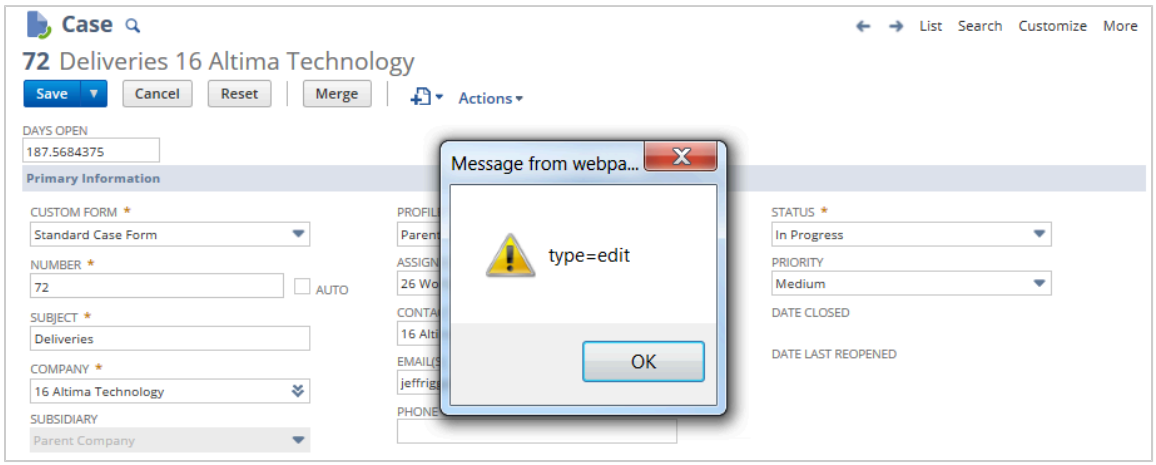
The previous screenshot demonstrates the definition of a record-level client script. This script is deployed globally to any records specified in the Deployments tab (see figure). In this case, this script will be deployed to all **Case** records in the system.



When a person opens any Case record, the following alerts are thrown right way on the pageInit (page load) trigger:



When you click **OK** to begin editing the record, the type=edit alert is thrown.



PageInit Sample

The PageInit function is called when the form is first loaded. Some of the functions that can be performed on PageInit include the following:

- Populate field defaults
- Disable or enable fields
- Change field availability or values depending on the data available for the record
- Add flags to set initial values of fields
- Provide alerts where the data being loaded is inconsistent or corrupted
- Retrieve user login information and change field availability or values accordingly
- Validate that fields required for your custom code (but not necessarily required for the form) exist

Examples

Set Default Field Values for a Field

```
function pageInit()
{
   // if fieldA is either NULL or equal to "valueA"
   if ((nlapiGetFieldValue('fieldA').length === 0) || (nlapiGetFieldText('fieldA') === "valueA"))
   {
      // then set fieldA to valueB
      nlapiSetFieldText('fieldA', nlapiGetFieldText('valueB'));
   }
}
```

Disable a Field

```
function pageInit()
{
   //On init, disable two optional Other fields: fieldA and fieldB.
   nlapiDisableField('custrecord_other_fieldA', true);
   nlapiDisableField('custrecord_other_fieldB', true);
}
```

Display User Profile Information

```
function pageInit()
{
   //On page init display the currently logged in User's profile information.
```

```
    // Set variables
    var userName = nlapiGetUser();         // entity id of the current user
    var userRole = nlapiGetRole();         // id of the current user's role
    var userDept = nlapiGetDepartment();   // id of the current user's department
    var userLoc = nlapiGetLocation();      // id of the current user's location

    //  Display information
    alert("Current User Information" + "\n\n" +
        "Name: " + userName + "\n" +
        "Role: " + userRole + "\n" +
        "Dept: " + userDept + "\n" +
        "Loc: " + userLoc
        );
}
```

Save Record Sample

The Save Record function is called when the user requests the form to be saved. This function returns false to reject the operation. Use the Save Record function to provide alerts to the user before committing the data. If it is necessary for the user to make changes before committing the data, return false — otherwise display the alert, return true and allow the user to commit the data.

You can also use the Save Record function to:

- Enable fields that were disabled with other functions
- Redirect the user to a specified URL

Examples

Requesting Additional Information

```
function saveRecord()
{
   // Verify that fieldA is populated. If not, block the save and warn with a popup.

   if (String(nlapiGetFieldValue('fieldA')).length === 0)
   {
      alert("Please provide a value for fieldA");
      return false;
   }
   alert("Are you sure you want to Save the record?");
   return true;
}
```

Redirect the User to Another Location

```
function saveRecord()
{
   window.open('https://<accountID>.app.netsuite.com/[url string]');void(0)
   return true;
}
```

Post Sourcing Sample

(Transaction Forms Only)

The Post Sourcing function is called when a field is modified that sources information from another field. Event handlers for this function behave similar to event handlers for the Change Field function except that the function is called only after all sourcing is completed — it waits for any cascaded field changes to complete before calling the user defined function. Therefore, the event handler is not triggered by field changes for a field that does not have any changes.

If there is at least one field sourced from a dropdown list (either a built-in sourcing or one created through customization) the post sourcing event is fired. Therefore, if you need to do something based on sourced values, you should do it in Post Sourcing rather than from Field Changed.

Example

On Sales order – Post sourcing

```
// Wait for all sourcing to complete from item field, get the rate field. If rate < 10, set it to 20.
// Execute this post sourcing function
function postSourcing(type, name)
{
 // Execute this code when all the fields from item are sourced on the sales order.

  if(type === 'item' && name === 'item')
  {
    // After all the fields from item are sourced
    var rate = nlapiGetCurrentLineItemValue('item', 'rate');
    var line = nlapiGetCurrentLineItemIndex(type);

    if(rate < 10)
    {
        nlapiSetCurrentLineItemValue('item', 'rate', 20);
    }
  }
}
```

ValidateField Sample

The ValidateField function is called whenever the user **changes** the value of a field. This function returns false to reject the value.

Note: This event type does not apply to dropdown fields or box fields.

Use the Validate Field function to validate field lengths, restrict field entries to a predefined format, restrict submitted values to a specified range, validate the submission against entries made in an associated field,

Examples

Validate Field Lengths

```
function ValidateField(type, name)
{
   // if fieldA is not at least 6 characters, fail validation
   if (name === 'fieldA')
   {
      var fieldALength = String(nlapiGetFieldValue('fieldA')).length;

      if (fieldALength < 6)
      {
         alert("FieldA must be at least 6 characters.");
         return false;
      }
   }
   // Always return true at this level, to continue validating other fields
   return true;
}
```

Validate Field is Uppercase

This sample uses a validate field function that ensure the value in the field with ID custrecord_mustbe_uppercase is always set to uppercase.

```
function validateFieldForceUppercase(type, name)
{
    if (name === 'custrecord_mustbe_uppercase')
    {
        //obtain the upper case value
        var upperCase = nlapiGetFieldValue('custrecord_mustbe_uppercase').toUpperCase();

        //make sure it hasn't been set
        if (upperCase !== nlapiGetFieldValue('custrecord_mustbe_uppercase'))
        {
            nlapiSetFieldValue('custrecord_mustbe_uppercase', upperCase, false);
        }

        return true ;
    }
}
```

Since this function is invoked every time there is an attempt to move the focus way from a field, the first **if** block ensures the uppercase logic is executed only for the correct field. Since using the API nlapiSetFieldValue would also trigger events, the second **if** block is used to ensure the code will not get into an infinite loop. The final `return true` statement ensures the focus can be successfully taken away from the field.

Field Changed Sample

The Field Changed function is called when a new value for a field is **accepted**. Use the Field Changed function to provide the user with additional information based on user input, disable or enable fields based on user input.

Examples

Requesting Additional Information

```
function FieldChanged(type, name)
{
   // Prompt for additional information,  based on values already selected.
   if ((name === 'fieldA') && (nlapiGetFieldText('fieldA') === "Other"))
   {
      alert("Please provide additional information about fieldA
      in the text field below.");
   }
}
```

User Event Scripts

The following topics are covered in this section. If you are new to user event scripts, these topics should be read in order:

- What Are User Event Scripts?
- User Event Script Execution
- Setting the User Event type Argument
- User Event Script Execution Types
- How Many User Events Can I Have on One Record?
- Running a User Event Script in NetSuite
- User Event Script Samples

What Are User Event Scripts?

User event scripts are executed on the NetSuite server. They are executed when users perform certain actions on records, such as create, load, update, copy, delete, or submit. Most standard NetSuite records and custom record types support user event scripts.

Important: User event scripts cannot be executed by other user event scripts or by workflows with a **Context** of **User Event Script**. In other words, you cannot chain user event scripts. You can execute a user event script from a call within a scheduled script, a portlet script, or a Suitelet.

With user event scripts you can do such things as:

- Implement custom validation on records
- Enforce user-defined data integrity and business rules
- Perform user-defined permission checking and record restrictions
- Implement real-time data synchronization
- Define custom workflows (redirection and follow-up actions)
- Implement custom form customizations

Note: To know which standard record types support user event scripts, see the help topic SuiteScript Supported Records in the NetSuite Help Center. If a record supports user event scripts, an X appears in the column called "Scriptable in Server SuiteScript".

You can use SuiteCloud Development Framework (SDF) to manage user event scripts as part of file-based customization projects. For information about SDF, see the help topic SuiteCloud Development Framework Overview. You can use the Copy to Account feature to copy an individual user event script to another of your accounts. Each user event script page has a clickable Copy to Account option in the upper right corner. For information about Copy to Account, see the help topic Copy to Account Overview.

Which User Event Types are Available in SuiteScript?

The user events types that are available in scripting are:

- Before Load – event occurs when a read operation on a record takes place. A Before Load event occurs when a user clicks Edit or View on an existing record, or clicks New to create a new record.
- Before Submit – event occurs when a record is submitted, but before the changes are committed to the database. A Before Submit event occurs when a user clicks Save or Submit on a record.
- After Submit – event occurs **after** the changes to the record are committed to the database. An After Submit event occurs after a user clicks Save or Submit on a record.

  See User Event Script Execution Types or specific details on these event types.

User Event Script Execution

Important: User event scripts cannot be executed by other user event scripts or by workflows with a **Context** of **User Event Script**. In other words, you cannot chain user event scripts. You can execute a user event script from a call within a scheduled script, a portlet script, or a Suitelet.

User event scripts are executed based on operation types defined as: **beforeLoad, beforeSubmit**, and **after submit**. See the following sections for details:

- User Event beforeLoad Operations

- User Event beforeSubmit and afterSubmit Operations

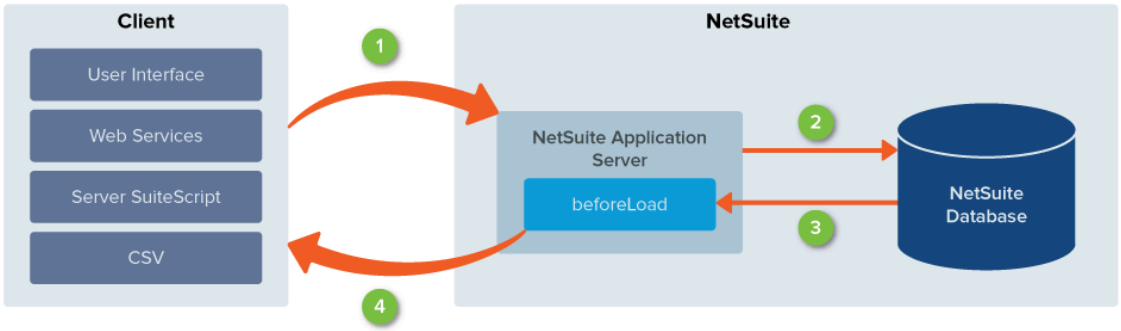
For information about the arguments each user event function takes, see Setting the User Event type Argument.

Tip: You can set the order in which user event scripts execute on the Scripted Records page. See the help topic The Scripted Records Page.

User Event beforeLoad Operations

The following steps and diagram provide an overview of what occurs during a beforeLoad operation:

1. The client sends a read operation request for record data. The client request can originate from the user interface, SOAP web services, server–side SuiteScript calls, CSV imports, or XML.
2. Upon receiving the request, the application server performs basic permission checks on the client.
3. The database loads the requested information into the application server for processing. This is where the **beforeLoad** operation occurs – before the requested data is returned to the client.
4. The client receives the now validated/processed **beforeLoad** data.



Note: Standard records cannot be sourced during a beforeLoad operation. Use the pageInit client script for this purpose. See Client Scripts.

User Event beforeSubmit and afterSubmit Operations

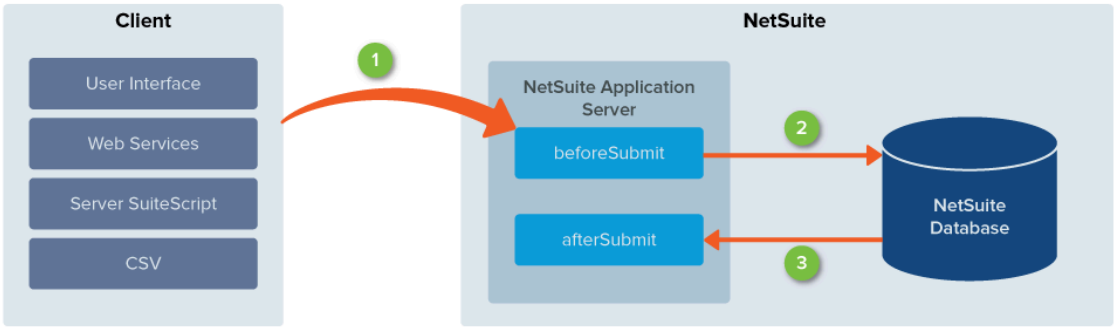
The following steps and diagram provide an overview of what occurs on submit (**beforeSubmit** and **afterSubmit**) operations:

1. The client performs a write operation by submitting data to the application server. The client request can originate from the user interface, SOAP web services, server–side SuiteScript calls, CSV imports, or XML. The application server:
   a. performs basic permission checks on the client
   b. processes the submitted data and performs specified validation checks during a **beforeSubmit** operation

   The submitted data has **NOT** yet been committed to the database.
2. After data has been validated, it is committed to the database.
3. If this (newly committed) data is then called by an **afterSubmit** operation, the data is taken from the database and is sent to the application server for additional processing. Examples of afterSubmit operations on data that are already committed to the database include, but are not limited to:

a. sending email notifications (regarding the data that was committed to the database)
b. creating child records (based on the data that was committed to the database)
c. assigning tasks to employees (based on data that was committed to the database)

Note: Asynchronous afterSubmit user events are only supported during webstore checkout.



Setting the User Event type Argument

For User Event scripts, you can associate a script execution context with the *type* argument of the script's function. For example, if you have a script associated with a **beforeLoad** operation for a specific record type, and you would like to cause an action only when the record is initially created, specify *create* as the script execution type.

Important: The *type* argument is an **auto-generated** argument passed by the system. You can NOT set this as a parameter for a specific deployment like other function arguments.

```
//Define the User Event function for a beforeLoad operation.
function beforeLoadSalesOrder( type )
{
   var newRecord = nlapiGetNewRecord();
   var cutoffRate = custscript_maximumdiscountlevel;
   var discountRate = newRecord.getFieldValue('discountrate');

 //Define the value of the type argument.
   if (type == ' create ' && discountRate != null && discountRate.length > 0
      && cutoffRate != null && cutoffRate.length > 0)
     {
        discountRate = Math.abs( parseFloat( discountRate ) );
        ...remainder of code...
```

Event type arguments vary depending on whether the event will occur on beforeLoad, beforeSubmit, or afterSubmit operations. The following table lists the script execution event types you can use with each operation.

Note: When you deploy user event scripts in NetSuite, you can also define a script execution event type using the Event Type list on the Script Deployment page. Be aware that the event type you choose from the list overrides the type(s) specified in the script. For details, see the help topic Setting Script Execution Event Type from the UI. For general information about defining other deployment parameters for User Event scripts, see Steps for Defining a Script Deployment.

All of the events have the common type argument which indicates the type of operation that invoked the event. This argument allows the script code to branch out to different logic depending on the operation

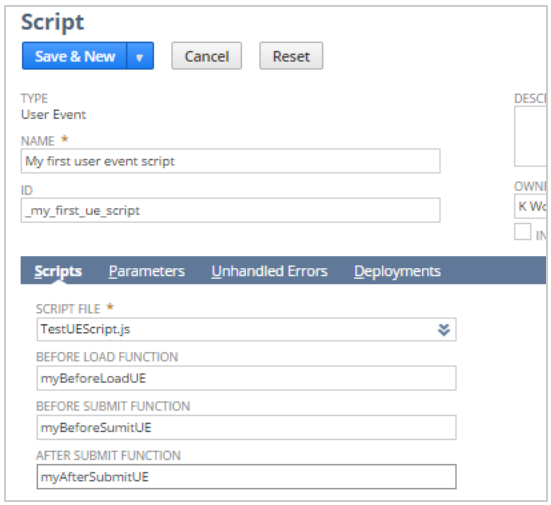
type. For example, a script with "deltree" logic that deletes a record and all of its child records should only be invoked when type equals to "delete". It is important that user event scripts check the value of the type argument to avoid indiscriminate execution.

The following sample demonstrates how to check the value of the type argument for each event. Event types include beforeLoad, beforeSubmit, afterSubmit. (See User Event Script Execution Types for more details.)

```
function myBeforeLoadUE(type)
{
  if (type == 'create')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is create');
  }

  if (type == 'view')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is view');
  }

  if (type == 'edit')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is edit');
  }
}

function myBeforeSubmitUE(type)
{
  if (type == 'create')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is create');
  }

  if (type == 'delete')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is delete');
  }

  if (type == 'edit')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is edit');
  }

  if (type == 'cancel')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is cancel');
  }
}

function myAfterSubmitUE(type)
{
  if (type == 'create')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is create');
  }

  if (type == 'delete')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is delete');
  }

  if (type == 'edit')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is edit');
  }

  if (type == 'approve')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is approve');
  }
}
```

Note: Logging done with `nlapiLogExecution` may be classified into 4 types: DEBUG, AUDIT, ERROR, and EMERGENCY. The source code should correctly set the logging type. Log type filtering may be set during runtime to give concise and useful logged information.

After uploading the source code file to the SuiteScript folder in the File Cabinet, a user event script record is defined by going to Customization > Scripts > New. The following shows the script record definition page.



User Event Script Execution Types

The following table lists all the execution context types that are supported in each user event type:

Operation Type	Execution Event Type	Notes
beforeLoad	*type* : the read operation type ■ create ■ edit ■ view ■ copy ■ print ■ email ■ quickview *form* : an nlobjForm object representing the current form *request* : an nlobjRequest object representing the GET request (Only available for browser requests.)	Event occurs whenever a read operation on a record occurs. These operations include navigating to a record in the UI, reading a record in SOAP web services, attaching a child custom record to its parent, detaching a child custom record from its parent, or calling nlapiLoadRecord. The function cannot be used to source standard records. Use the pageInit client script for this purpose. See Client Event Types. The user-defined function is executed prior to returning the record or page. The function is passed either the *type*, *form*, or *request* arguments by the system. **Notes:** ■ beforeLoad user events cannot be triggered when you load/access an online form. ■ Data cannot be manipulated for records that are loaded in beforeLoad scripts. If you attempt to update a record loaded in beforeLoad, the logic is ignored. ■ Data can be manipulated for records created in beforeLoad user events.

Operation Type	Execution Event Type	Notes
beforeSubmit	*type* : the write operation type ■ create ■ edit ■ delete ■ xedit - (see Inline Editing and SuiteScript) ■ approve - (only available for certain record types) ■ reject - (only available for certain record types) ■ cancel - (only available for certain record types) ■ pack- (only available for certain record types, for example Item Fulfillment records) ■ ship - (only available for certain record types, for example Item Fulfillment records) ■ markcomplete (specify this type for a beforeSubmit script to execute when users click the Mark Complete link on call and task records) ■ reassign (specify this type for a beforeSubmit script to execute when users click the Grab link on case records) ■ editforecast (specify this type for a beforeSubmit script to execute when users update opportunity and estimate records using the Forecast Editor)	Events on a beforeSubmit operation occur prior to any write operation on the record. Changes to the current record at this stage will be persisted during the write operation. The beforeSubmit operation is useful for validating the submitted record, performing any restriction and permission checks, and performing any last-minute changes to the current record. **Notes:** ■ The *approve, cancel*, and *reject* argument types are only available for record types such as sales orders, expense reports, timebills, purchase orders, and return authorizations. ■ Only beforeLoad and afterSubmit user event scripts will execute on the Message record type when a message is created by an inbound email case capture. Scripts set to execute on a beforeSubmit event will not execute. ■ User Event Scripts cannot override custom field permissions. For instance, if a user's role permissions and a custom field's permissions differ, beforeSubmit cannot update the custom field, even if the script is set to execute as Administrator. ■ Attaching a child custom record to its parent or detaching a child custom record from its parent will trigger an edit event. **Best practices:** To set a field on a record or make ANY changes to a record that is being submitted, do so on a **beforeSubmit** operation, NOT an afterSubmit operation. If you set a field on an afterSubmit, you will be duplicating a record whose data has already been committed to the database. **Important:** Do not attempt to load or submit the current record with nlapiLoadRecord or nlapiSubmitRecord on a beforeSubmit event. Doing so will result in the loss of data.
afterSubmit	*type* : the write operation type ■ create ■ edit ■ delete ■ xedit - (see Inline Editing and SuiteScript) ■ approve - (only available for certain record types) ■ cancel - (only available for certain record types) ■ reject - (only available for certain record types)	Events on an afterSubmit operation occur immediately following any write operation on a record. **Note:** Asynchronous afterSubmit user events are only supported during webstore checkout. The afterSubmit operation is useful for performing any actions that need to occur following a write operation on a record. Examples of these actions include email notification, browser redirect, creation of dependent records, and synchronization with an external system.

Operation Type	Execution Event Type	Notes
	▪ pack - (only available for certain record types, for example Item Fulfillment records) ▪ ship - (only available for certain record types, for example Item Fulfillment records) ▪ dropship - (for purchase orders with items specified as "drop ship") ▪ specialorder - (for purchase orders with items specified as "special order") ▪ orderitems - (for purchase orders with items specified as "order item") ▪ paybills - (use this type to trigger afterSubmit user events for Vendor Payments from the Pay Bill page. Note that no sublist line item information will be available . Users must do a lookup/search to access line item values.)	Attaching a child custom record to its parent or detaching a child custom record from its parent will trigger an edit event. **Note:** The *approve, cancel*, and *reject* argument types are only available for record types such as sales orders, expense reports, timebills, purchase orders, and return authorizations. **Best Practices:** Users should be doing post-processing of the current record on an afterSubmit. Use Case: 1. Load the record you want to make changes to by calling the nlapiLoadRecord API. Do NOT load the record object by using nlapiGetRecord, as this API returns the record in READ ONLY mode; therefore, changes made to the record cannot be accepted and an error is thrown. 2. After using nlapiLoadRecord to load the record, make the changes to the record, and submit the record. For example, if you have a sales order that you want tied to an estimate, load the sales order record, update it with any information, and submit it again.

How Many User Events Can I Have on One Record?

There is no limit to the number of user event scripts you can execute on a particular record type. For example, you could have 10 beforeLoad, 9 beforeSubmit, and 15 afterSubmit executing functions on a Customer record. However, assigning this many executable functions to one record type is **highly discouraged**, as this could negatively affect user experience with that record type. In other words, if you have 10 beforeLoad scripts that must complete their execution before a record loads into the browser for the user, this may significantly increase the time it takes for the record to load. Therefore, the user's experience working with the record will be negatively affected.

Developers who include scripts in their SuiteApps should also be aware of the number of user events scripts that might already be deployed to records types in the target account. For example, if 8 beforeSubmit user event scripts are deployed to the Sales Order record in the target account, and your SuiteApp includes another 7 beforeSubmit user event scripts on the Sales Order record type, this is 15 beforeSubmit scripts running every time a user clicks Save on the record. Although all of the scripts will run, the time it takes for the record to save may be significantly increased, again, negatively affecting user experience with the record.

If you must run multiple user event scripts on one record type, and are experiencing slow execution times, use the Application Performance Management (APM) SuiteApp to troubleshoot the issue. APM keeps 30 days worth of performance data for each scriptable record type. You can view the overall execution time of each record instance, as well as the execution time of each script. See the help topic Application Performance Management (APM) for additional information.

Running a User Event Script in NetSuite

To run a user event script in NetSuite, you must:

1. Create a JavaScript file for your user event script.
2. Load the file into NetSuite.
3. Create a Script record.
4. Define all runtime options on the Script Deployment page.

If you are new to SuiteScript and need information about each of these steps, see Running SuiteScript 1.0 in NetSuite Overview.

User Event Script Samples

The following samples are provided in this section:

- Generating a Record Log
- Creating Follow-up Phone Call Records for New Customers
- Enhancing NetSuite Forms with User Event Scripts

Generating a Record Log

This user event script creates an execution log entry containing the type, record type, and internalId of the current record.

```
function beforeLoad(type,form)
{
   var newId = nlapiGetRecordId();
   var newType = nlapiGetRecordType();
   nlapiLogExecution('DEBUG','<Before Load Script> type:'+type+', RecordType: '+newType+', Id:'+newId);
}
function beforeSubmit(type)
{
   var newId = nlapiGetRecordId();
   var newType = nlapiGetRecordType();
   nlapiLogExecution('DEBUG','<Before Submit Script> type:'+type+', RecordType: '+newType+', Id:'+newId);
}
function afterSubmit(type)
{
   var newId = nlapiGetRecordId();
   var newType = nlapiGetRecordType();
   nlapiLogExecution('DEBUG','<After Submit Script> type:'+type+', RecordType: '+newType+', Id:'+newId);
}
```

Creating Follow-up Phone Call Records for New Customers

A CRM use case that could be addressed with user event scripts is creating a follow-up phone call record for every newly created customer record. The solution is to deploy a script on the customer record's *afterSubmit* event that will create the phone call record.

In the above use case, *afterSubmit* is a better event to handle the logic than *beforeSubmit*. In the *beforeSubmit* event, the customer data has not yet been committed to the database. Therefore, putting the phone call logic in the *afterSubmit* event guarantees there will not be an orphan phone call record.

Note: During design time, developers should carefully consider in which event to implement their server logic.

```
function followUpCall_CustomerAfterSubmit(type)
{
   //Only executethe logic if a new customer is created
```

```
  if (type == 'create' )
  {
    //Obtain a handle to the newly created customer record
    var custRec = nlapiGetNewRecord();

    if (custRec.getFieldValue('salesrep') != null )
    {
      //Create a new blank instance of a Phone Call
      var call = nlapiCreateRecord("phonecall");

      //Setting the title field on the Phone Call record
      call.setFieldValue('title', 'Make follow-up call to new customer');

      //Setting the assigned field to the sales rep of the new customer
      call.setFieldValue('assigned', custRec.getFieldValue('salesrep'));

      //Use the library function to obtain a date object that represents tomorrow
      var today = new Date();
      var tomorrow = nlapiAddDays(today, 1);
      call.setFieldValue('startdate', nlapiDateToString(tomorrow));

      //Setting the phone field to the phone of the new customer
      call.setFieldValue('phone', custRec.getFieldValue('phone'));

      try
      {
        //committing the phone call record to the database
        var callId = nlapiSubmitRecord(call, true);
        nlapiLogExecution('DEBUG', 'call record created successfully', 'ID = ' + callId);
      }
      catch (e)
      {
        nlapiLogExecution('ERROR', e.getCode(), e.getDetails());
      }
    }
  }
}
```

Note: APIs such as `nlapiSubmitRecord` that access the database should be wrapped in try-catch blocks.

The phone call use case may be further enhanced by redirecting the user to the Phone Call page after it has been created. This redirect is accomplished by putting in redirect logic after the phone call record is submitted.

```
try
{
  //committing the phone call record to the database
  var callId = nlapiSubmitRecord(call, true);
  nlapiLogExecution('DEBUG', 'call record created successfully', 'ID = ' + callId);

  //Redirect the user to the newly created phone call
  nlapiSetRedirectURL('RECORD', 'phonecall', callId, false, null);
}
catch (e)
{
  nlapiLogExecution('ERROR', e.getCode(), e.getDetails());
}
```

User event scripts are not only triggered by user actions carried out through the browser, they are also triggered by other means as well (for example, CSV or SOAP web services).

Examples:

- Using CSV to import records triggers *before submit* and *after submit* events
- Using the SOAP web services "GET" operation to retrieve an existing record would trigger its *before load* event.

Sometimes these events invoke scripts not designed to be executed in that manner and create undesirable results. To prevent a script from getting executed by the wrong execution context, use the nlobjContext object as a filter.

For example, to ensure a *before load* user event script is executed only when a record is created using the browser interface, the script must check both the type argument and the execution context as (as shown below):

```
function myBeforeLoadUE(type)
{
  //obtain the context object
  var context = nlapiGetContext();
  if (type == 'create' && context.getExecutionContext == 'userinterface')
  {...}
}
```

Note that the API nlapiGetContext() is not exclusive to user event scripts. It can also be used in Client Scripts and Suitelets.

Note: The nlobjContext object provides metadata of the script's context. Use this information to help implement fine-grained control logic in SuiteScript.

Enhancing NetSuite Forms with User Event Scripts

Another common use of user event scripts is to dynamically customize or enhance entry forms and transactions forms. This approach gives NetSuite forms the ability to customize themselves in runtime – something that cannot be done with pre-configured, roles-based forms.

In NetSuite, entry forms and transaction forms are customized by administrators. The placement of UI elements (fields, tabs, sublists) on a form can be arranged, or be made inline or hidden depending on the business needs of the end users. Multiple forms can be created for a record type and assigned to specific roles. Typically this kind of customization is done during design time. Custom forms are confined to specific roles and do not allow for a lot of runtime customization. A user event script on a record's **beforeLoad** event can provide flexibility to runtime customization.

Note: For more specific information about NetSuite entry forms and transaction forms, see the help topic Custom Forms in the NetSuite Help Center.

The key to using user event scripts to customize a form during runtime is a second argument named `form` in the **beforeLoad** event. This optional argument is the reference to the entry/transaction form. Developers can use this to dynamically change existing UI elements, or add new ones. The UI elements are added using the UI Objects API.

A use case for this scripting capability could be the following:

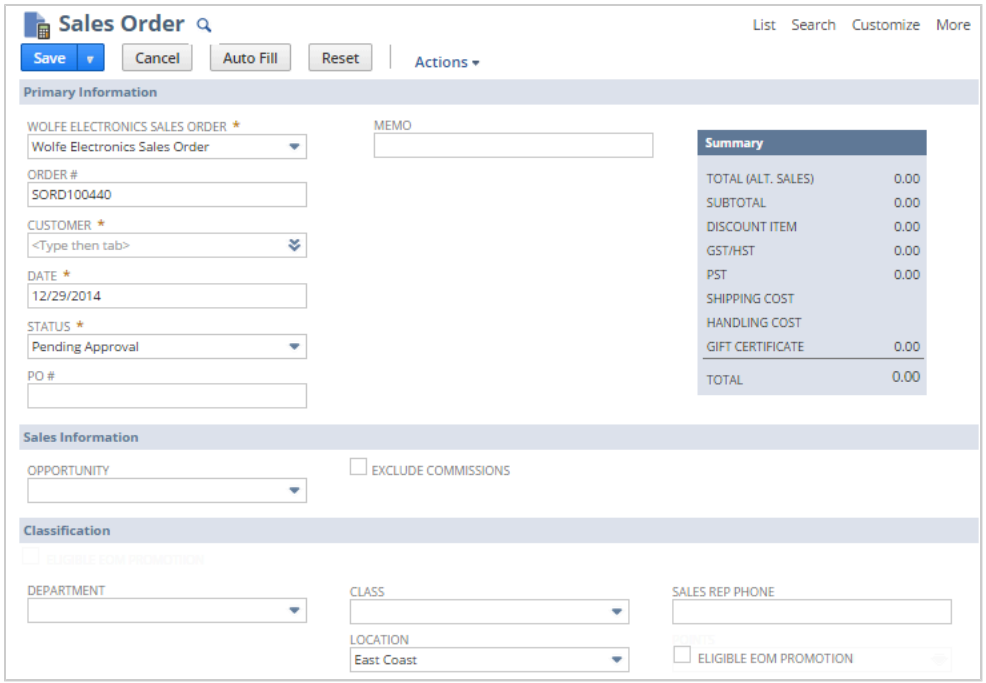
To improve month-end sales, a company introduces an end-of-month promotion that is only active for the last five days of the month. All sales order forms must have a custom field called “Eligible EOM Promotion” on the last five days of the month.

The following is a sample user event script that is meant to be deployed on the **beforeLoad** event of the sales order record.

```
/**************************************************
 *This function is a module to implement at end of
 * month(last 5 days of month) promotion for sales
 * orders. It is meant to be deployed on the before
 * load event of the sales order record.
 */
function customizeUI_SalesOrderBeforeLoad(type, form)
{
  var currentContext = nlapiGetContext();
```

```
10
11    //Execute the logic only when creating a sales order with the browser UI
12    if (type == 'create' && currentContext.getExecutionContext() == 'userinterface')
13    {
14      var fieldId = 'custpage_eom_promotion';
15      var fieldLabel = 'Eligible EOM promotion';
16      var today = new Date();
17      var month = today.getMonth();
18      var date = today.getDate();
19      nlapiLogExecution('DEBUG', 'month date', month + ' ' + date);
20
21      //February
22      if (month==1)
23      {
24        if (date==24 | date==25 | date==26 | date==27 | date==28 | date==29)
25        form.addField(fieldId, 'checkbox', fieldLabel);
26      }
27      //31-day months
28      else if (month==0 | month==2 | month ==4 | month==6 | month==7 | month==9 | month==11)
29      {
30        if ( date==27 | date==28 | date==29 | date==30 | date==31)
31        form.addField(fieldId, 'checkbox', fieldLabel);
32      }
33      //30-day months
34      else
35      {
36        if ( date==26 | date==27 | date==28 | date==29 | date==30)
37        form.addField(fieldId, 'checkbox', fieldLabel);
38      }
39    }
40  }
```

When the script is deployed, all sales order forms will display the **Eligible EOM Promotion** box during the last five days of the month.



Note that since these UI elements are created dynamically, they are superficial and do not have supporting server data models. There is a disconnect between the UI and server data, therefore the script-created fields' values **will not be saved**.

UI elements (such as the Eligible EOM promotion field) created with user event scripts and SuiteScript Objects are scriptable by client script APIs. A remedy to the disconnect problem is linking the script-

created field to a real field (with server data support) using a client script. The value of the real field, which might be made hidden or inline on the form definition, is driven by the value entered in the script-created field. The real fields are populated and the data is saved.

Suitelets

The following topics are covered in this section. If you are not familiar with Suitelets, these topics should be read in order.

- What Are Suitelets?
- Suitelet Script Execution
- Building Custom Workflows with Suitelets
- Building Suitelets with UI Objects
- Backend Suitelets
- Reserved Parameter Names in Suitelet URLs
- SuiteScript and Externally Available Suitelets
- Running a Suitelet in NetSuite
- Suitelets Samples

What Are Suitelets?

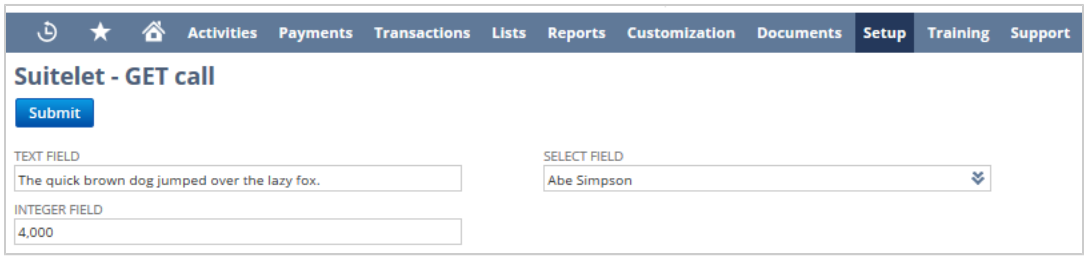
Suitelets are extensions of the SuiteScript API that give developers the ability to build custom NetSuite pages and backend logic. Suitelets are server-side scripts that operate in a request-response model. They are invoked by HTTP GET or POST requests to system generated URLs.

Note: Suitelets are not intended to work inside web stores. Use online forms to embed forms inside a web store.

Important: The governance limit for concurrent requests for Suitelets available without login is the same as the limit for RESTlets. For information about the account limits, see the help topic Web Services and RESTlet Concurrency Governance. For Suitelets that are authenticated through login, the number of concurrent requests is currently not governed.

Shown below are screenshots of a Suitelet with a few fields. The Suitelet is invoked by making a GET request from the browser. Notice that this Suitelet is built with SuiteScript UI Objects, which encapsulate scriptable interface components that have a NetSuite look-and-feel.

After a Suitelet has been deployed, developers can create NetSuite tasklinks to these scripts, which can then be used to customize existing NetSuite centers.



When the Submit button is clicked, the same Suitelet is invoked again with a HTTP POST event. The values entered in the previous screen are displayed in inline (read-only) mode.

Below is the source code for this Suitelet. It is executed on the server, which generates HTML and sends it to the browser.

```
function gettingStartedSuitelet(request, response)
{
  if (request.getMethod() == 'GET' )
  {
    //Create the form and add fields to it
    var form = nlapiCreateForm("Suitelet - GET call" );
    form.addField('custpage_field1', 'text', 'Text Field' ).setDefaultValue('This is a text field' );
    form.addField('custpage_field2', 'integer', 'Integer Field' ).setDefaultValue(10);
    form.addField('custpage_field3', 'select', 'Select Field', 'customer' );

    form.addSubmitButton('Submit' );

    response.writePage(form);
    }
  //POST call
  else
  {
    var form = nlapiCreateForm("Suitelet - POST call" );

    //create the fields on the form and populate them with values from the previous screen
    var resultField1 = form.addField('custpage_res1', 'text', 'Text Field value entered: ' );
    resultField1.setDefaultValue(request.getParameter('custpage_field1' ));
    resultField1.setDisplayType('inline' );

    var resultField2 = form.addField('custpage_res2', 'integer', 'Integer Field value entered: ' );
    resultField2.setDefaultValue(request.getParameter('custpage_field2' ));
    resultField2.setDisplayType('inline' );

    var resultField3 = form.addField('custpage_res3', 'select', 'Select Field value entered: ', 'customer' );
    resultField3.setDefaultValue(request.getParameter('custpage_field3' ));
    resultField3.setDisplayType('inline' );

    response.writePage(form);
  }
}
```

The entry point of the function has two required arguments: `request` and `response`. These arguments are instances of nlobjRequest and nlobjResponse, respectively.

Typically, invoking a Suitelet using the browser would make a HTTP GET call. The type of HTTP call is determined by the nlobjRequest.getMethod() API. The code creates an nlobjForm object and populates it with SuiteScript UI Objects. The populated form is sent to the `response` object using the nlobjResponse.writePage(pageobject) API.

When the user clicks the Submit button, an HTTP POST call is made. The code's `else` block obtains the values entered in the first page from the `request` object and populates them into another nlobjForm object and sends it to response.writePage(pageobject).

Note: As a best practice, set the content type of the response object using the setContentType(type, name, disposition) API. For more information about content types, see SuiteScript 1.0 Supported File Types.

You can use SuiteCloud Development Framework (SDF) to manage Suitelets as part of file-based customization projects. For information about SDF, see the help topic SuiteCloud Development

Framework Overview. You can use the Copy to Account feature to copy an individual Suitelet to another of your accounts. Each Suitelet page has a clickable Copy to Account option in the upper right corner. For information about Copy to Account, see the help topic Copy to Account Overview.

Translating Suitelet Labels

If your account has the Multi-Language feature enabled, you can translate Suitelet labels so that they match the language of the NetSuite user interface.

When you create a link for the Suitelet, there is a **Translation** column where you can add the translations for the languages you have selected on the **Languages** subtab of the General Preferences page. See the help topic Enabling the Entry of Translation Strings for a Specific Language for more information about how to specify the languages that you want to have available.

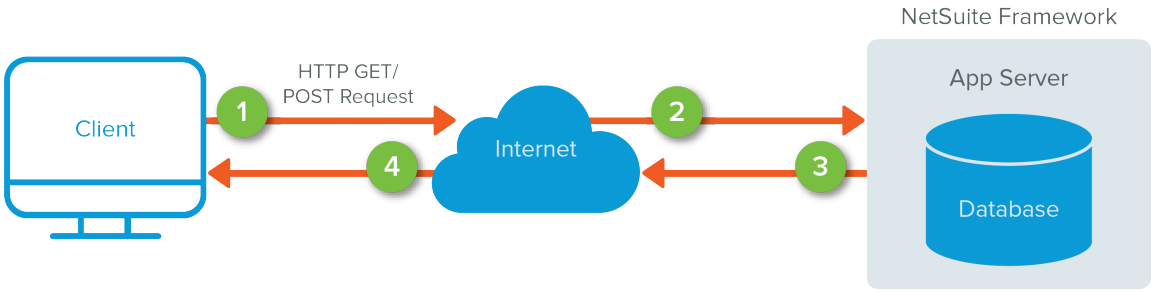
If you need more information to create a link for the Suitelet, see Running a Suitelet in NetSuite.

Suitelet Script Execution

The following steps and diagram provide an overview of the Suitelet execution process:

1. Client initiates an HTTP GET or POST request (typically from a browser) for a system-generated URL. A web request object (nlobjRequest) contains the data from the client's request.
2. The user's script is invoked, which gives the user access to the entire Server SuiteScript API as well as a web request and web response object.
3. NetSuite processes the user's script and returns a web response object (nlobjResponse) to the client. The response can be in following forms:
   - Free-form text
   - HTML
   - RSS
   - XML
   - A browser redirect to another page on the Internet

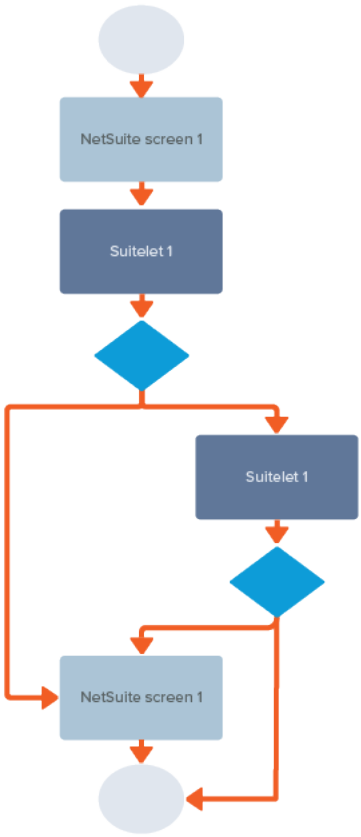
     Important: You can only redirect to external URLs from Suitelets that are accessed externally (in other words, the Suitelet has been designated as "Available Without Login" and is accessed from its external URL).
   - A browser redirect to an internal NetSuite page. The NetSuite page can be either a standard page or custom page that has been dynamically generated using UI Objects.
4. The data renders in the user's browser.



Building Custom Workflows with Suitelets

With user event scripts, Suitelets, and UI Objects, SuiteScript developers can create custom workflows by chaining together NetSuite pages (both standard and custom). These workflows may be complemented by custom backend logic.

The following diagram shows how a SuiteScript developer can potentially create a workflow that starts off with either a standard or custom NetSuite record, then redirects to a Suitelet, then redirects to either another Suitelet or standard/custom NetSuite record, all depending on the logic of the developer's application.



Building Suitelets with UI Objects

When building Suitelets, developers can use SuiteScript UI Objects to create custom pages that look like NetSuite pages. SuiteScript UI objects encapsulate the elements for building NetSuite-looking portlets, forms, fields, sublists, tabs, lists, and columns.

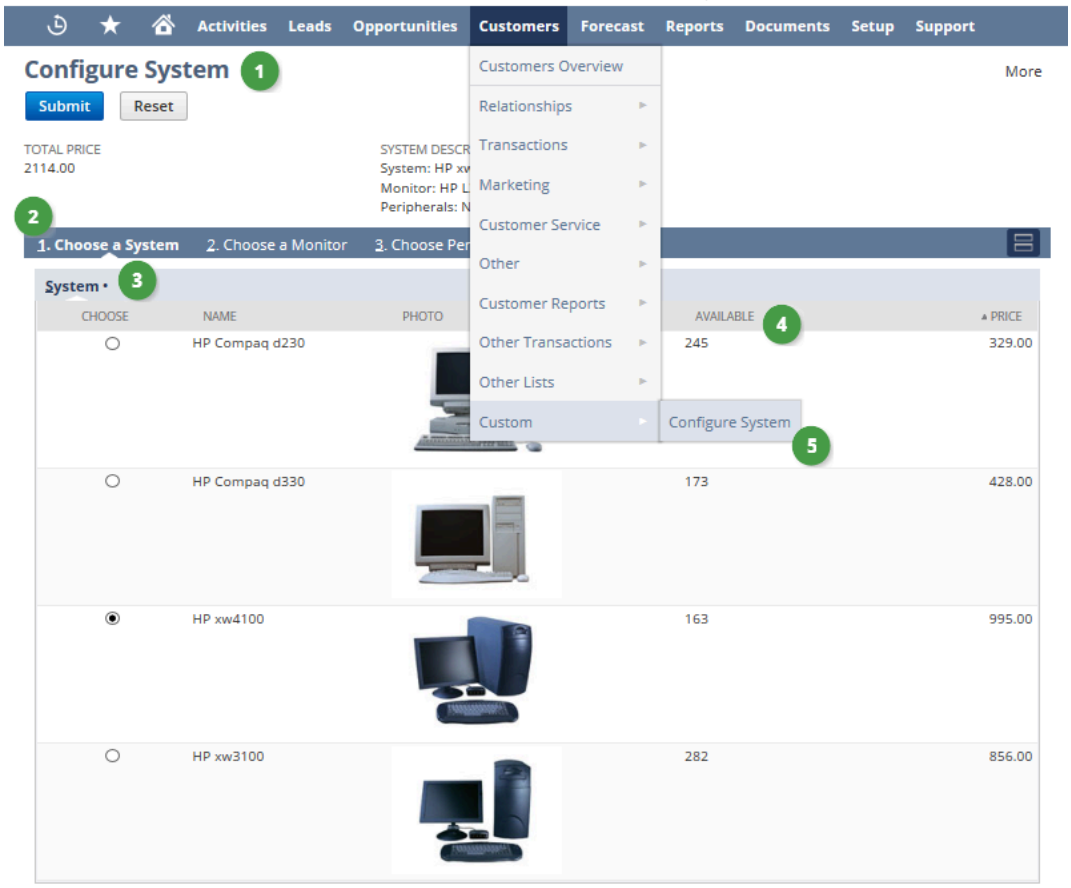
When developing a Suitelet with UI objects, you can also add custom fields with inline HTML.

Important: When adding UI elements to an **existing** NetSuite page, you must prefix the object name with **custpage**. This minimizes the occurrence of field/object name conflicts. For example, when adding a custom tab to an entry form, the name should follow a convention similar to **custpage** customtab or **custpage** mytab.

The figure below shows a custom interface that has been built with the following SuiteScript UI objects:

- nlobjForm
- nlobjTab
- nlobjSubList
- nlobjField

Note that a custom menu link was created to access the Suitelet. In this figure, the Configure System Suitelet can be accessed by going to Customers > Custom > Configure System. For information about creating menu links for Suitelets, see Running a Suitelet in NetSuite.



Item	Description
1	SuiteScript UI Object: FORM
2	SuiteScript UI Object: TAB
3	SuiteScript UI Object: SUBLIST
4	SuiteScript UI Object: FIELD
5	Custom Menu Link

Backend Suitelets

Suitelets give developers the ability to build custom NetSuite pages. However, developers can create Suitelets that do not generate any UI elements. These kinds of Suitelets are referred to as backend

Suitelets. Their sole purpose is to execute backend logic, which can then be parsed by other parts of a custom application.

Similar to a Suitelet that builds NetSuite pages, a backend Suitelet is invoked by making HTTP GET or POST calls to a NetSuite-generated Suitelet URL.

The following are good uses of backend Suitelets:

- Providing a service for backend logic to other SuiteScripts, or to other external hosts outside of NetSuite
- Offloading server logic from client scripts to a backend Suitelet shipped without source code to protect sensitive intellectual property

  A use case of a backend Suitelet is a service that provides customer information based on a phone number. The following is the code for a Suitelet that returns customer entity IDs (for records with matching phone numbers) separated by the | character.

```
/**********************************************
 *This functionsearches forcustomer records* thatmatch asupplied parametercustparam_phone
 *and returnthe resultsin astring separated* bythe |character.
 */function lookupPhoneBackendSuitelet(request, response)
{
  if (request.getMethod() == 'GET' )
  {
    //null checkon therequired parameterif (request.getParameter('custparam_phone') != null )
    {
      //Setting upthe filtersand columnsvar filters = newArray ();
      var columns = newArray ();

      //Use thesupplied custparam_phonevalue asfilter
      filters[0] = new nlobjSearchFilter('phone', null, 'is', request.getParameter('custparam_phone' ));
      columns[0] = new nlobjSearchColumn('entityid', null, null );

      //Search forcustomer recordsthat matchthe filtersvar results = nlapiSearchRecord('customer', null, filters, columns);

      if (results != null )
      {
        var resultString = '' ;
        //Loop throughthe resultsfor (var i = 0; i < results.length ; i++)
        {
          //constructing theresult stringvar result = results[i];
          resultString = resultString + result.getValue('entityid' );

          //adding the| seperatorif (i != parseInt (results.length - 1))
          {
            resultString = resultString + '|' ;
          }
          nlapiLogExecution('DEBUG', 'resultString', resultString);
        }
        response.write(resultString);
      }
      else
      {
        response.write('none found' );
      }
    }
  }
}
```

Notice that this Suitelet does not use any UI Object APIs. Communication with the Suitelet is done strictly with the `request` and `response` objects. NetSuite generates a URL to invoke this Suitelet. To correctly invoke it, the custparam_phone value (bold) needs to be appended at the end of the invoking URL:

```
https://<accountID>.app.netsuite.com/app/site/hosting/scriptlet.nl?script=6&deploy=1&custparam_phone=(123)-456-7890
```

The code that calls this backend Suitelet needs to do the following:

1. Use nlapiResolveURL to dynamically obtain the invoking URL
2. Supply required parameters
3. Process the returned results

Note: Backend Suitelets should not be used to get around SuiteScript usage governance. Suitelets designed with this intention are considered abusive by NetSuite.

Reserved Parameter Names in Suitelet URLs

Certain names are reserved and should not be referenced when naming custom parameters for Suitelet URLs.

The following table contains a list of reserved parameter names:

Reserved Suitelet URL Parameter Names	
e	print
id	email
cp	q
l	si
popup	st
s	r
d	displayonly
_nodrop	nodisplay
sc	deploy
sticky	script

If any of your parameters are named after any of the reserved parameter names, your Suitelet may throw an error saying, "There are no records of this type." To avoid naming conflicts, you should name your custom URL parameters with a **custom** prefix. For example, use *custom_id* instead of *id*.

SuiteScript and Externally Available Suitelets

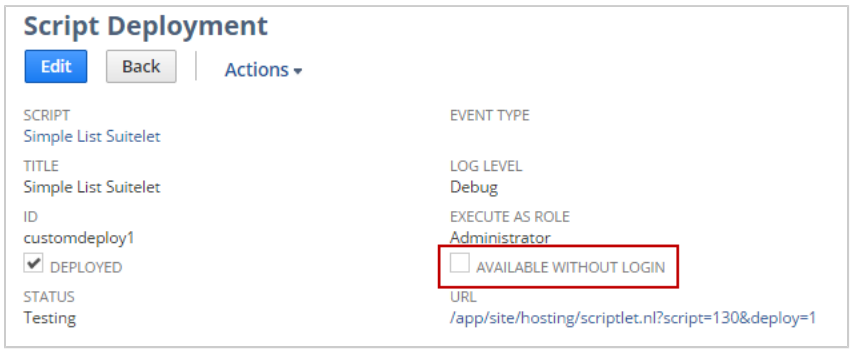
Only a subset of the SuiteScript API is supported in **externally** available Suitelets (Suitelets set to Available Without Login on the Script Deployment page). For a list of these APIs, in the NetSuite Help Center see these topics related to online forms.

- Working with Online Forms
- Why are only certain APIs supported on online forms?

Note: The same concepts that apply to online forms also apply to externally available Suitelets.

Note that if you want to use all available SuiteScript APIs in a Suitelet, your Suitelet will require a valid NetSuite session. (A valid session means that users have authenticated to NetSuite by providing their email address and password.)

On the Script Deployment page, leave the Available Without Login box unselected if you want to deploy a Suitelet that requires a valid session. (See also Setting Available Without Login for more information about this runtime option.)



Important: UI Objects can be used without a valid session. Therefore, they **are** supported in externally available Suitelets.

Running a Suitelet in NetSuite

To run a Suitelet in NetSuite, you must:

1. Create a JavaScript file for your Suitelet.
2. Load the file into NetSuite.
3. Create a Script record.
4. Define all runtime options on the Script Deployment page.

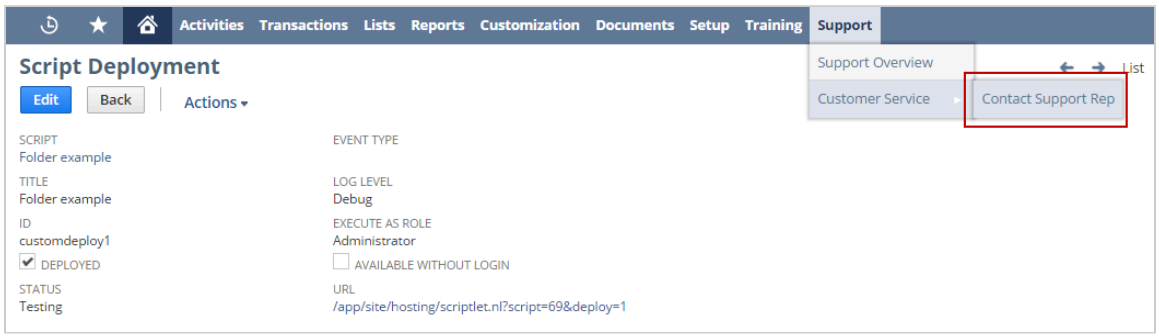
If you are new to SuiteScript and need information about each of these steps, see Running SuiteScript 1.0 in NetSuite Overview.

Note that if you want users to be able to access/launch a Suitelet from the UI, you can create a menu item for the Suitelet.

The following figure shows the Links tab on the Script Deployment page for a Suitelet. Select the Center where the link to the Suitelet will be accessible (for example, Customer Center, Vendor Center, etc). Next, set the Section (top-level menu tab) in the Center, then choose a category under the section. Finally, create a UI label for the link. Be sure to click Add when finished.

Note: The Classic Center is a default center. It is not specific to customers, partners, or vendors.

When the Script Deployment page is saved, a link to the Suitelet appears (see figure).



Suitelets Samples

The following sample Suitelets show how to return HTML and XML documents, as well as how to create forms and lists.

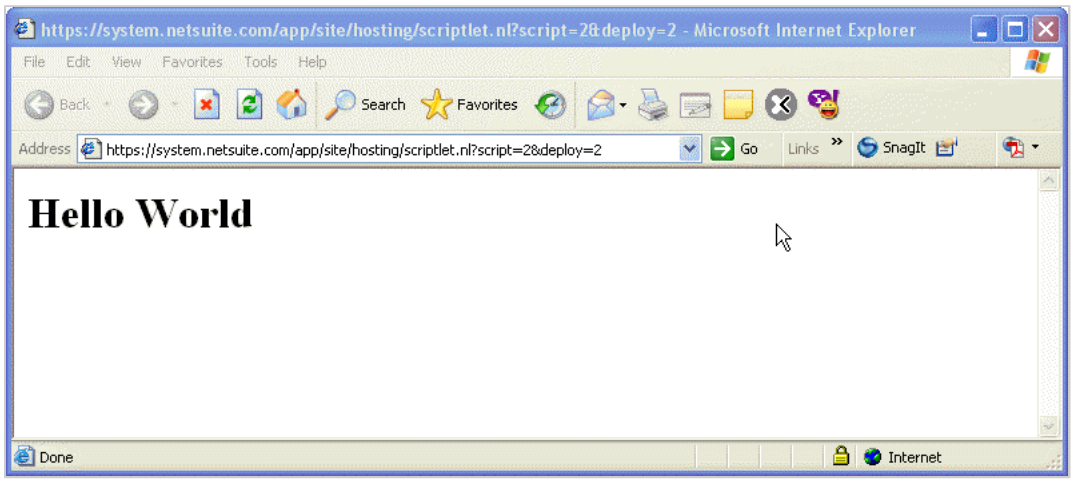
- Writing Your First Suitelet
- Return a Simple XML Document
- Create a Simple Form
- Create a Simple List
- Add a Suitelet to a Tab
- Create a Suitelet Email Form
- Create a Form with a URL Field
- Using Embedded Inline HTML in a Form

Writing Your First Suitelet

This basic Hello World! sample shows how to return an HTML document in a Suitelet.

Script:

```
function demoHTML(request, response)
{
    var html =  '<html><body><h1>Hello World</h1></body></html>';
    response.write( html );
   //prefix header with Custom-Header. See nlobjResponse.setHeader(name, value)
    response.setHeader('Custom-Header-Demo', 'Demo');
}
```

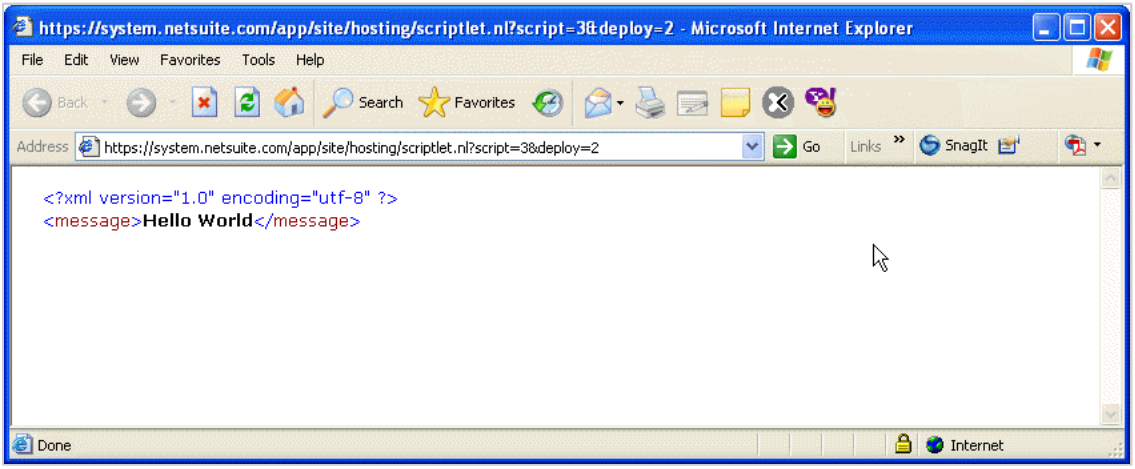


Return a Simple XML Document

Script:

```
function demoXML(request, response)
{
    var xml =   '<?xml version="1.0" encoding="utf-8" ?>'+
                '<message>'+'Hello World'+'</message>';
```

```
    response.write( xml );
    response.setHeader('Custom-Header-Demo', 'Demo');
}
```



Create a Simple Form

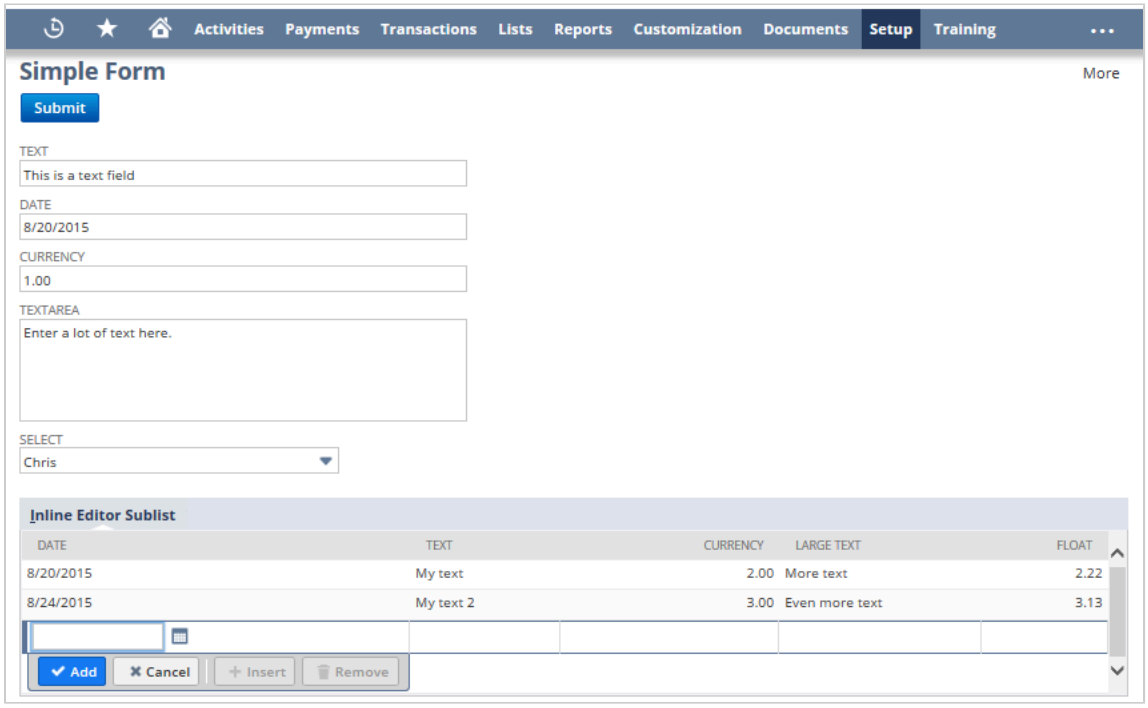
To create a simple form using SuiteScript 2.x, see the help topic N/ui/serverWidget Module Script Samples.

This form is generated using the nlobjForm UI object. Note that Suitelets built with the SuiteScript UI Objects API can be accessed without a valid session. In other words, Suitelets using UI objects can be set to **Available Without Login** on the Script Deployment page. (For information about deploying scripts, see Step 5: Define Script Deployment.)

Script:

```
function demoSimpleForm(request, response)
{
   if ( request.getMethod() == 'GET' )
   {
        var form = nlapiCreateForm('Simple Form');
        var field = form.addField('textfield','text', 'Text');
        field.setLayoutType('normal', 'startcol')
        form.addField('datefield','date', 'Date');
        form.addField('currencyfield','currency', 'Currency');
        form.addField('textareafield','textarea', 'Textarea');

        var select = form.addField('selectfield','select', 'Select');
        select.addSelectOption('','');
        select.addSelectOption('a', 'Albert');
        select.addSelectOption('b', 'Baron');
        select.addSelectOption('c', 'Chris');
        select.addSelectOption('d', 'Drake');
        select.addSelectOption('e', 'Edgar');

        var sublist = form.addSubList('sublist','inlineeditor', 'Inline Editor Sublist');
        sublist.addField('sublist1', 'date', 'Date');
        sublist.addField('sublist2', 'text', 'Text');
        sublist.addField('sublist3', 'currency', 'Currency');
        sublist.addField('sublist4', 'textarea', 'Large Text');
        sublist.addField('sublist5', 'float', 'Float');

        form.addSubmitButton('Submit');

        response.writePage( form );
   }
   else
```

```
        dumpResponse(request,response);
}
```



Create a Simple List

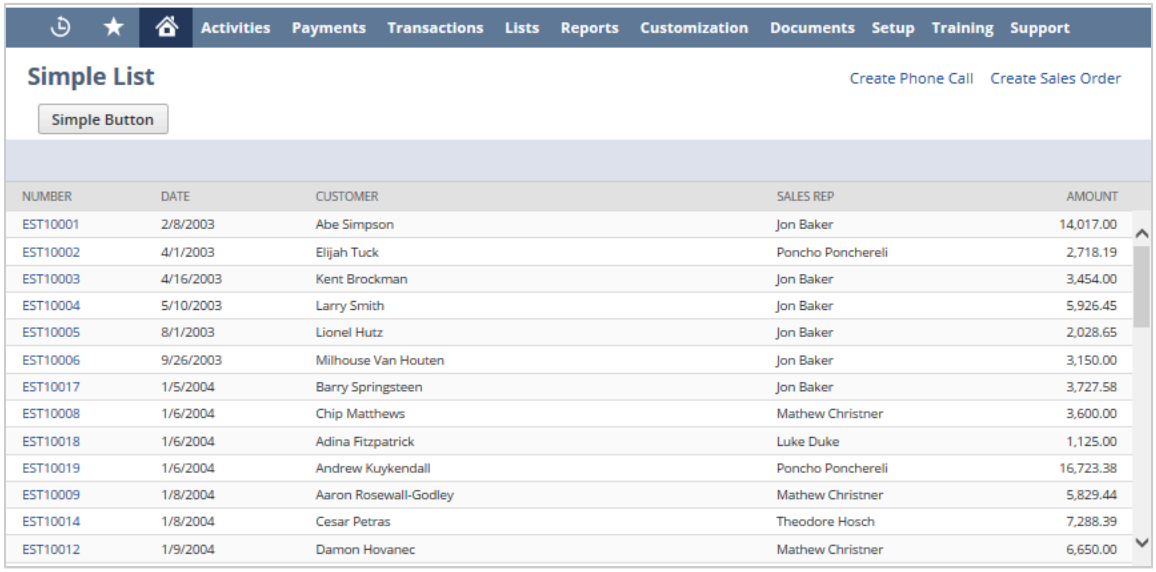
This list is generated using the nlobjList UI object. Note that Suitelets built with the UI Objects API can be accessed without a valid session. In other words, Suitelets using UI objects can be set to **Available Without Login** on the Script Deployment page. (For information about deploying scripts, see Step 5: Define Script Deployment.)

Important: If your browser is inserting scroll bars in this code sample, maximize your browser window, or expand the main frame that this sample appears in.

Script:

```
function demoList(request, response)
{
     var list = nlapiCreateList('Simple List');

     // You can set the style of the list to grid, report, plain, or normal, or you can get the
     // default list style that users currently have specified in their accounts.
     list.setStyle(request.getParameter('style'));

     var column = list.addColumn('number', 'text', 'Number', 'left');
     column.setURL(nlapiResolveURL('RECORD','salesorder'));
     column.addParamToURL('id','id', true);

     list.addColumn('trandate', 'date', 'Date', 'left');
     list.addColumn('name_display', 'text', 'Customer', 'left');
     list.addColumn('salesrep_display', 'text', 'Sales Rep', 'left');
     list.addColumn('amount', 'currency', 'Amount', 'right');

     var returncols = new Array();
     returncols[0] = new nlobjSearchColumn('trandate');
     returncols[1] = new nlobjSearchColumn('number');
```

```
    returncols[2] = new nlobjSearchColumn('name');
    returncols[3] = new nlobjSearchColumn('salesrep');
    returncols[4] = new nlobjSearchColumn('amount');

    var results = nlapiSearchRecord('estimate', null, new nlobjSearchFilter('mainline',null,'is','T'), returncols);
    list.addRows( results );

    list.addPageLink('crosslink', 'Create Phone Call', nlapiResolveURL('TASKLINK','EDIT_CALL'));
    list.addPageLink('crosslink', 'Create Sales Order',
       nlapiResolveURL('TASKLINK','EDIT_TRAN_SALESORD'));

    list.addButton('custombutton', 'Simple Button', 'alert(\'Hello World\')');
    response.writePage( list );
}
```



Add a Suitelet to a Tab

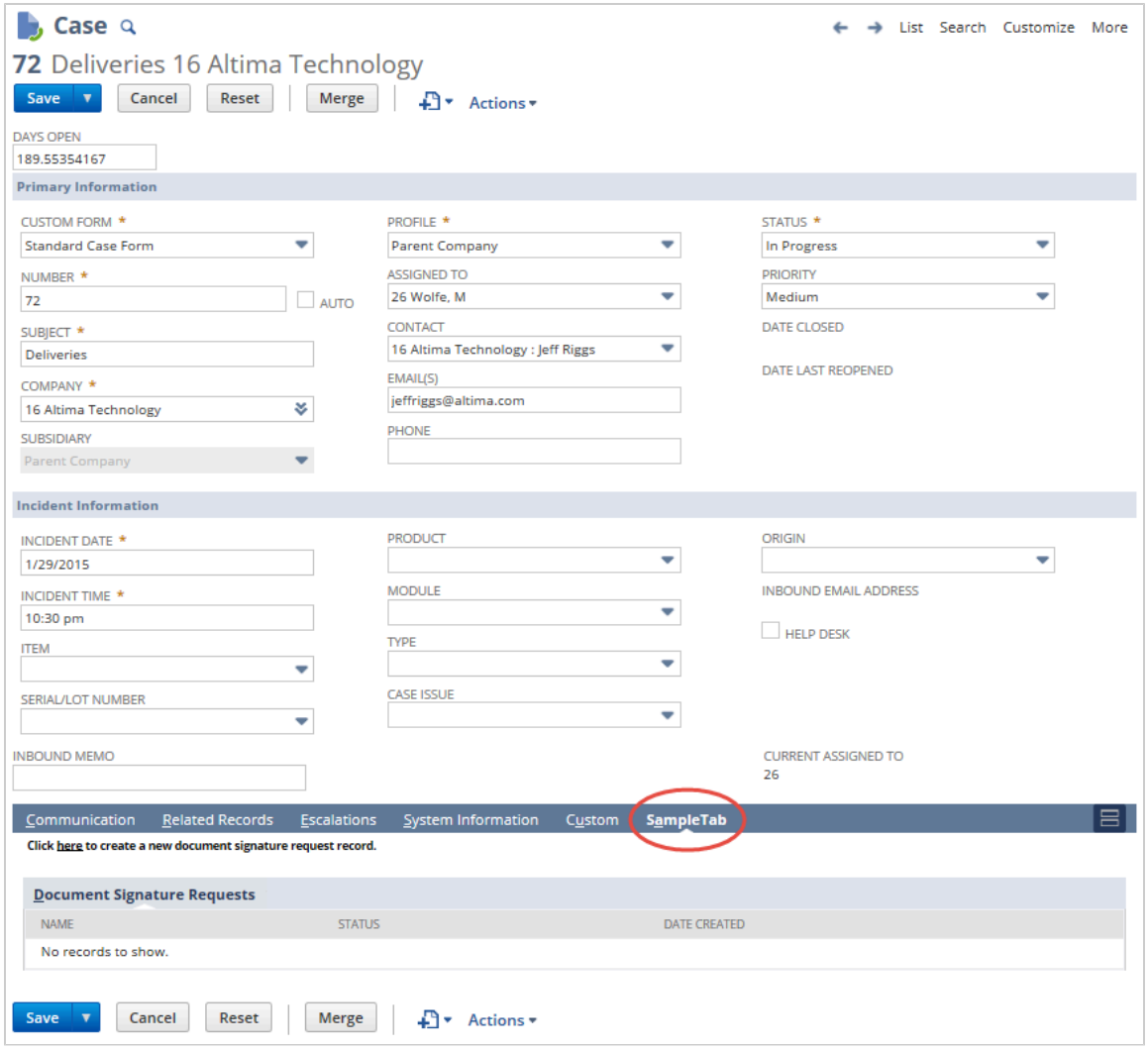
The following user event script shows how to add a tab to a record, and then on the tab, add a Suitelet. In this example a tab called *Sample Tab* is added to a Case record. A link is added to the Sample Tab that, when clicked, opens a Suitelet.

Important: If your browser is inserting scroll bars in this code sample, maximize your browser window, or expand the main frame that this sample appears in.

Script:

```

/*Create a user event beforeLoad function that takes type and form as parameters.
 *Later you will define the script execution context by providing a value for type.
 *The form argument instantiates a SuiteScript nlobjForm object, which allows you
 *to add fields and sublists later on in the script.
*/
function beforeLoadTab(type, form)
{
var currentContext = nlapiGetContext();
var currentUserID = currentContext.getUser();

/*Define the value of the type argument. If the Case record is edited or viewed,
 *a tab called Sample Tab is added. Note that the script execution context is set to
 * userinterface. This ensures that this script is ONLY invoked from a user event
 *occurring through the UI.
```

```
*/
if( (currentContext.getExecutionContext() == 'userinterface') && (type == 'edit' | type == 'view'))
   {
   var SampleTab = form.addTab('custpage_sample_tab', 'SampleTab');

   //On Sample Tab, create a field of type inlinehtml.
   var createNewReqLink = form.addField('custpage_new_req_link','inlinehtml', null, null,
   'custpage_sample_tab');

   //Define the parameters of the Suitelet that will be executed.
    var linkURL = nlapiResolveURL('SUITELET', 'customscript12','customdeploy1', null)
    + '&customerid=' + nlapiGetRecordId();

   //Create a link to launch the Suitelet.
   createNewReqLink.setDefaultValue('<B>Click <A HREF="' + linkURL +'">here</A> to
    create a new document signature request record.</B>');

   //Add a sublist to Sample Tab.
   var signatureRequestSublist = form.addSubList('custpage_sig_req_sublist',
   'list', 'Document Signature Requests','custpage_sample_tab');

   signatureRequestSublist.addField('custpage_req_name', 'text', 'Name');
   signatureRequestSublist.addField('custpage_req_status', 'text', 'Status');
   signatureRequestSublist.addField('custpage_req_created', 'date', 'Date
   Created');
   }
}
```



Note: This screenshot displays the NetSuite user interface that was available before Version 2010 Release 2.

Create a Suitelet Email Form

The following sample shows how to create a Suitelet form to email the results.

Script:

```
/**
 * A simple Suitelet for building an email form and sending out an email
 * from the current user to the recipient email address specified on the form.
 */
function simpleEmailForm(request, response)
{
   if ( request.getMethod() == 'GET' )
   {
        var form = nlapiCreateForm('Email Form');
        var subject = form.addField('subject','text', 'Subject');
        subject.setLayoutType('normal','startcol')
        subject.setMandatory( true );
        var recipient = form.addField('recipient','email', 'Recipient email');
        recipient.setMandatory( true );
        var message = form.addField('message','textarea', 'Message');
        message.setDisplaySize( 60, 10 );
        form.addSubmitButton('Send Email');
        response.writePage(form);
   }
   else
   {
            var currentuser = nlapiGetUser();
            var subject = request.getParameter('subject')
            var recipient = request.getParameter('recipient')
            var message = request.getParameter('message')
        nlapiSendEmail(currentuser, recipient, subject, message);
   }
}
```

Create a Form with a URL Field

This example shows how to add a URL field to a basic form. In this example, when the URL is clicked the user will be redirected to the New Employee record.

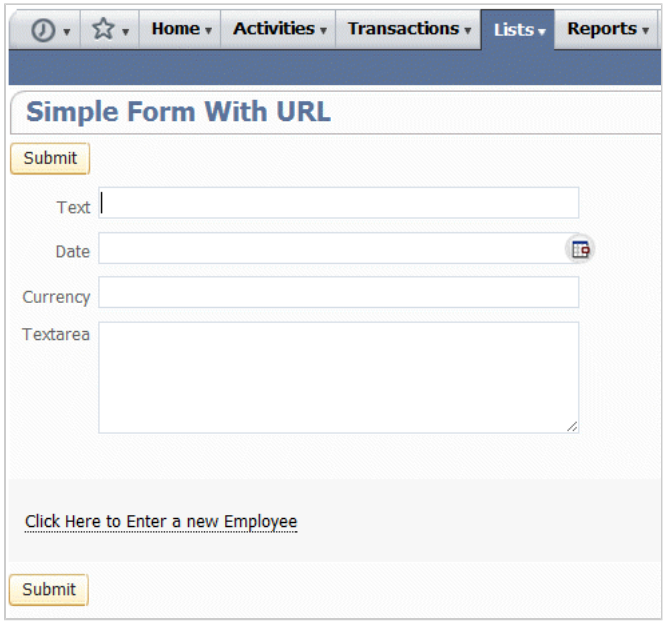
Several methods of nlobjField are used to change the displayed field to the correct parameters. The address called is built from the system address plus the address of the tasklink (in this case the New Employee form) which is retrieved using nlapiResolveURL(type, identifier, id, displayMode).

Note: You need to specify 'https://' if the destination link is to be accessible from an authenticated NetSuite session.

Script:

```
function SimpleFormWithUrl(request, response)
{
   if ( request.getMethod() == 'GET' )
   {
        var form = nlapiCreateForm('Simple Form');
```

```
        var field = form.addField('textfield','text', 'Text');
        field.setLayoutType('normal', 'startcol');
        form.addField('datefield','date', 'Date');
        form.addField('currencyfield','currency', 'Currency');
        form.addField('textareafield','textarea', 'Textarea');

        form.addField("enterempslink", "url", "", null, "enteremps" ).setDisplayType( "inline" ).setLinkText( "Click Here to Enter Employee Records").setDefaultValue( "https://<accountID>.app.netsuite.com" + nlapiResolveURL( 'tasklink', 'EDIT_EMPLOYEE') );

        form.addSubmitButton('Submit');

        response.writePage( form );
    }
    else
        dumpResponse(request,response);
}
```



Using Embedded Inline HTML in a Form

Use HTML to define a custom field to be included on custom pages generated by Suitelets. For more information, see Suitelets.

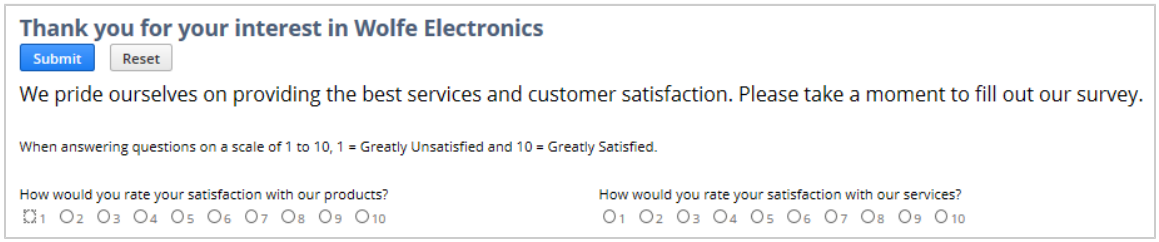
Important: NetSuite Forms does not support manipulation through the Document Object Model (DOM) and custom scripting in Inline HTML fields. The use of Inline HTML fields on Form pages (except for Forms generated by Suitelets) will be deprecated in a future release.

Here is an inline HTML example where the inline HTML is embedded in a NetSuite form. To embed inline HTML, add a field with the nlobjForm.addField method of the type 'inlinehtml', and usethe nlobjField.setDefaultValue method to provide the HTML code.

Script:

```
function SurveyInlineHTML(request,response)
{
    var form = nlapiCreateForm('Thank you for your interest in Wolfe Electronics', true);

```

```
    var htmlHeader = form.addField('custpage_header', 'inlinehtml').setLayoutType('outsideabove', 'startrow');
    htmlHeader
        .setDefaultValue("<p style='font-size:20px'>We pride ourselves on providing the best
            services and customer satisfaction.  Please take a moment to fill out our survey.</p><br><br>");

    var htmlInstruct = form.addField('custpage_p1', 'inlinehtml').setLayoutType('outsideabove', 'startrow');
    htmlInstruct
        .setDefaultValue("<p style='font-size:14px'>When answering questions on a scale of 1 to 10,
            1 = Greatly Unsatisfied and 10 = Greatly Satisfied.</p><br><br>");

    form.addField('custpage_lblproductrating', 'inlinehtml')
        .setLayoutType('normal', 'startrow')
        .setDefaultValue("<p style='font-size:14px'>How would you rate your satisfaction with our products?</p>");

    form.addField('custpage_rdoproductrating', 'radio', '1', 'p1').setLayoutType('startrow');
    form.addField('custpage_rdoproductrating', 'radio', '2', 'p2').setLayoutType('midrow');
    form.addField('custpage_rdoproductrating', 'radio', '3', 'p3').setLayoutType('midrow');
    form.addField('custpage_rdoproductrating', 'radio', '4', 'p4').setLayoutType('midrow');
    form.addField('custpage_rdoproductrating', 'radio', '5', 'p5').setLayoutType('midrow');
    form.addField('custpage_rdoproductrating', 'radio', '6', 'p6').setLayoutType('midrow');
    form.addField('custpage_rdoproductrating', 'radio', '7', 'p7').setLayoutType('midrow');
    form.addField('custpage_rdoproductrating', 'radio', '8', 'p8').setLayoutType('midrow');
    form.addField('custpage_rdoproductrating', 'radio', '9', 'p9').setLayoutType('midrow');
    form.addField('custpage_rdoproductrating', 'radio', '10', 'p10').setLayoutType('endrow');

    form.addField('custpage_lblservicerating', 'inlinehtml')
        .setLayoutType('normal', 'startrow')
        .setDefaultValue("<p style='font-size:14px'>How would you rate your satisfaction with our services?</p>");
    form.addField('custpage_rdoservicerating', 'radio', '1', 'p1').setLayoutType('startrow');
    form.addField('custpage_rdoservicerating', 'radio', '2', 'p2').setLayoutType('midrow');
    form.addField('custpage_rdoservicerating', 'radio', '3', 'p3').setLayoutType('midrow');
    form.addField('custpage_rdoservicerating', 'radio', '4', 'p4').setLayoutType('midrow');
    form.addField('custpage_rdoservicerating', 'radio', '5', 'p5').setLayoutType('midrow');
    form.addField('custpage_rdoservicerating', 'radio', '6', 'p6').setLayoutType('midrow');
    form.addField('custpage_rdoservicerating', 'radio', '7', 'p7').setLayoutType('midrow');
    form.addField('custpage_rdoservicerating', 'radio', '8', 'p8').setLayoutType('midrow');
    form.addField('custpage_rdoservicerating', 'radio', '9', 'p9').setLayoutType('midrow');
    form.addField('custpage_rdoservicerating', 'radio', '10', 'p10').setLayoutType('endrow');

    form.addSubmitButton('Submit');

    response.writePage(form);
}
```



RESTlets

You can deploy server-side scripts that interact with NetSuite data following RESTful principles. RESTlets extend the SuiteScript API to allow custom integrations with NetSuite. Some benefits of using RESTlets include the ability to:

- Find opportunities to enhance usability and performance, by implementing a RESTful integration that is more lightweight and flexible than SOAP-based web services.
- Support stateless communication between client and server.
- Control client and server implementation.

- Use built-in authentication based on token or user credentials in the HTTP header.
- Develop mobile clients on platforms such as iPhone and Android.
- Integrate external Web-based applications such as Gmail or Google Apps.
- Create backends for Suitelet-based user interfaces.

RESTlets offer ease of adoption for developers familiar with SuiteScript and support more behaviors than SOAP-based web services, which are limited to those defined as SOAP web services operations. RESTlets are also more secure than Suitelets, which are made available to users without login. For a more detailed comparison, see the help topic RESTlets vs. Other NetSuite Integration Options.

For more information about the REST architectural style, a description of the constraints and principles is available at http://en.wikipedia.org/wiki/Representational_State_Transfer.

For details about working with RESTlets, see:

- Working with RESTlets
- RESTlets vs. Other NetSuite Integration Options
- Creating a RESTlet
- Debugging a RESTlet
- Sample RESTlet Code
- Sample RESTlet Input Formats
- RESTlet Status Codes and Error Message Formats
- Tracking and Managing RESTlet Activity

Important: To maintain the highest security standards for connectivity, NetSuite periodically updates server certificates and the trusted certificate store we use for https requests. We expect client applications that integrate with NetSuite to use an up-to-date certificate store, to ensure that integrations do not break due to certificates expiring or changing security standards. This issue typically does not affect modern browsers because these clients are maintained to use the latest certificates.

You can use SuiteCloud Development Framework (SDF) to manage RESTlet scripts as part of file-based customization projects. For information about SDF, see the help topic SuiteCloud Development Framework Overview. You can use the Copy to Account feature to copy an individual RESTlet script to another of your accounts. Each RESTlet script page has a clickable Copy to Account option in the upper right corner. For information about Copy to Account, see the help topic Copy to Account Overview.

Working with RESTlets

You can invoke a RESTlet using an HTTP request to a system-generated URL. RESTlets send and receive content in a request-response model using HTTP verbs, HTTP headers, HTTP status codes, URLs, and standard data formats.

RESTlets support the entire SuiteScript API and general SuiteScript features such as debugging. For general information about using SuiteScript, see SuiteScript 1.0 - The Basics.

The following topics provide details specific to the RESTlet type of script:

- RESTlet Script Execution
- RESTlet URL and Domain

- Supported Input and Output Content Types for RESTlets
- Supported Functions for RESTlets
- RESTlet Governance and Session Management
- RESTlet Debugging
- RESTlet Error Handling
- RESTlet Security

RESTlet Script Execution

The following steps provide an overview of the RESTlet execution process:

1. A client initiates an HTTP request for a system-generated URL. This request can originate from an external client or from a client hosted by NetSuite.
2. The sender of the HTTP request is authenticated either through request-level credentials passed by the NetSuite-specific method NLAuth, the NetSuite implementation of OAuth 1.0, or through a check for an existing NetSuite session.
   - For an externally hosted client using NLAuth, request-level credentials are required.
   - For an externally hosted client using OAuth 1.0, a token is required.
   - For a NetSuite-hosted client, the existing NetSuite session is reused.

   See the help topic Authentication for RESTlets.
3. The RESTlet script is invoked, providing access to the server SuiteScript API.
4. A string, or an object that has been deserialized according to a predefined specification, is passed in to the RESTlet. See Supported Input and Output Content Types for RESTlets.
5. The RESTlet interrogates the string or object, and can perform any of the full range of SuiteScript actions.
6. The RESTlet returns results as a string or a serialized object. nlobjResponse

Important: The URL used to invoke a RESTlet varies according to whether the RESTlet client is externally hosted or hosted by NetSuite. See RESTlet URL and Domain.

RESTlet URL and Domain

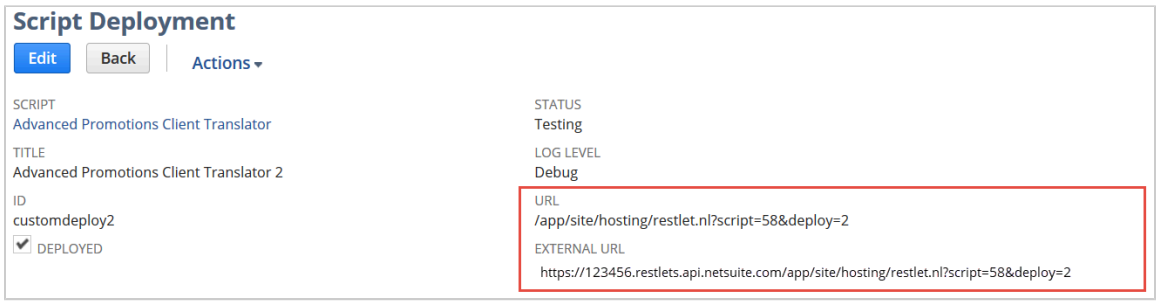
The URL used for a RESTlet HTTP request varies according to whether the RESTlet is called from an external client or from a client hosted by NetSuite.

- For a RESTlet called from an external client, the URL must be an absolute URL. This URL includes your account-specific RESTlet domain. If you do not know your account-specific URL, use the REST roles service to dynamically discover the correct domain. For details, see the help topic The REST Roles Service.
- For a RESTlet called from a client hosted by NetSuite, the URL should be a relative URL. That is, the URL does not include the domain.

  When the NetSuite account is hosted at `<accountID>.app.netsuite.com`, the RESTlet also uses the same base URL. For example, if you use `/app/site/hosting/restlet.nl` (as seen in the following screenshot) and deploy that RESTlet in accounts located in different data centers, the application will correctly prefix `/app/site/hosting/restlet.nl` with the base URL, as appropriate for the location of each account.

The following RESTlet deployment record shows examples of URLs.

- The URL field displays the relative URL, used by NetSuite-hosted clients for any references made to objects inside NetSuite.
- The External URL field displays an absolute URL that can be used to call the RESTlet from outside of NetSuite. However, any integrations you create must dynamically discover the domain that makes up the first part of this URL (the parts that differ from the text shown under the URL field).



Note: RESTlets use the same debugging domain as other SuiteScript types, **https://debugger.netsuite.com**. Whether the RESTlet client is hosted externally or by NetSuite does not change the debugger domain used. See RESTlet Debugging.

For more information about NetSuite domains, see the help topic Understanding NetSuite URLs.

Supported Input and Output Content Types for RESTlets

RESTlets support JSON and plain text content types for input and output. For each RESTlet, output content type is the same as input content type.

You must set the content type in the HTTP Content-Type header. You can use the following values to specify the input/output content type for a RESTlet:

- application/json
- text/plain

If you specify a content type other than JSON or text, a 415 error is returned with the following message:

```
Invalid content type. You can only use application/json or text/plain with RESTlets.
```

Using JSON Objects and Arrays

JSON is an acronym for JavaScript Object Notation, which is a subset of JavaScript. This special object notational construct is a syntax used to pass JavaScript objects containing name/value pairs, arrays, or other objects. The following JSON formatting is used for RESTlets:

- Each JSON object is an unordered set of name/value pairs, or members, enclosed in curly braces.
  - Each member is followed by a comma, which is called a value separator.
  - Within each member, the name is separated from the value by a colon, which is called a name separator.
  - Each name and each value is enclosed in quotation marks.

- Each JSON array is an ordered sequence of values, enclosed in square braces. Array values are separated by commas.

For examples of how to format JSON input for restlets, see Sample RESTlet Input Formats.

Supported Functions for RESTlets

RESTlets currently support a subset of HTTP methods, as shown in the following table:

HTTP Method	Input	Output	Description
GET	Parameter Object	Object	Requests a representation of the specified resource.
POST	Object	Object	Submits data to be processed, for example from an HTML form. Data is included in the body of the request, and can result in creation of a new resource, updates of existing resource(s), or both.
DELETE	Parameter Object	No Content	Passes in the ID and record type of the resource to be deleted, so that nlapiDeleteRecord or other delete-related logic can be called. This method does not return anything.
PUT	Object	Object	Uploads a representation of the specified resource.

The functions that implement these methods are specified on a RESTlet's script record. Each RESTlet must have a function for at least one of these HTTP methods. Each HTTP method can call any SuiteScript nlapi functions. See Create the RESTlet Script Record.

For examples of these functions in RESTlets, see Sample RESTlet Code.

RESTlet Governance and Session Management

The SuiteScript governance model tracks usage units on two levels: API level and script level. At the API level, RESTlets have the same usage limits as other types of SuiteScripts. At the script level, RESTlets allow 5,000 usage units per script, a limit five times greater than Suitelets and most other types of SuiteScripts. For more information, see the help topic SuiteScript Governance and Limits.

There is a limit of 10MB per string used as RESTlet input or output.

SuiteScript currently does not support a logout operation similar to the one used to terminate a session in SOAP web services.

Important: Starting from version 2017.2, web services and RESTlet concurrency is governed per account. The new account governance limit applies to the combined total of web services and RESTlet requests. For details about this change, see the help topic Web Services and RESTlet Concurrency Governance.

RESTlet Debugging

You can use the SuiteScript Debugger to debug RESTlet scripts, in the same manner that you use it to debug other types of server SuiteScripts. RESTlets use the same debugging domain as other SuiteScript types, **https://debugger.netsuite.com**. Both on demand debugging and deployed debugging are

supported for RESTlets. For general instructions for using the Debugger, see the help topic SuiteScript Debugger.

Important: For deployed debugging of a RESTlet, you need to set the session cookies of the client application that run the RESTlet to the same cookies listed for the RESTlet in the Debugger. Also, you must remove the authorization header from your RESTlet before debugging. For more details, see Debugging a RESTlet.

RESTlet Error Handling

RESTlets return standard HTTP status codes for their contained HTTP requests. A standard success code is returned for a successful request. Standard error codes are returned for errors due to unparsable input, authentication failure, lack of server response, use of an unsupported method, and use of an invalid content type or data format for input. In most cases, generic HTTP error messages are returned. The format used for error messages is the same as the specified format for input: JSON or plain text. For more details, see RESTlet Status Codes and Error Message Formats.

RESTlets also support the SuiteScript nlapiCreateError function. You can include this API in your code to abort script execution when an error occurs. For details, see the help topic Error Handling APIs.

RESTlet Security

The URLs for RESTlet HTTP requests can vary, but all resources are protected by TLS encryption.

Only requests sent using TLS 1.2 encryption are granted access. For more on RESTlet domains, see RESTlet URL and Domain. For more information, see the help topic Supported TLS Protocol and Cipher Suites.

Creating a RESTlet

To run a RESTlet in NetSuite, you must first define your client code and behavior, then define your RESTlet and its required functions. The client will send requests to the RESTlet URL generated by NetSuite.

To define a RESTlet:

1. Create a JavaScript file for your RESTlet code.
2. Load the file into NetSuite.
3. Create a script record where you define SuiteScript functions for one or more HTTP methods.
4. Define all runtime options on the Script Deployment page.

See the following for instructions for these tasks:

- Create the RESTlet File and Add It to the File Cabinet
- Create the RESTlet Script Record
- Define RESTlet Deployment(s)

Create the RESTlet File and Add It to the File Cabinet

1. Create a .js file and add your code to it, in the same manner that you create other types of SuiteScript files, as described in Step 1: Create Your Script.

This single script file should include GET, POST, DELETE, or PUT function(s) as necessary.

2. After you have created a .js file with your RESTlet code, you need to add this file to the NetSuite File Cabinet.

   The following steps describe how to add the file manually. If you are using SuiteCloud IDE, this process is automated. For more information, see Step 2: Add Script to NetSuite File Cabinet.

   1. Go to Documents > Files > File Cabinet, and select the folder where you want to add the file.

      You should add your file to the SuiteScripts folder, but it can be added to any other folder of your choice.

   2. Click **Add File**, and browse to the .js file.

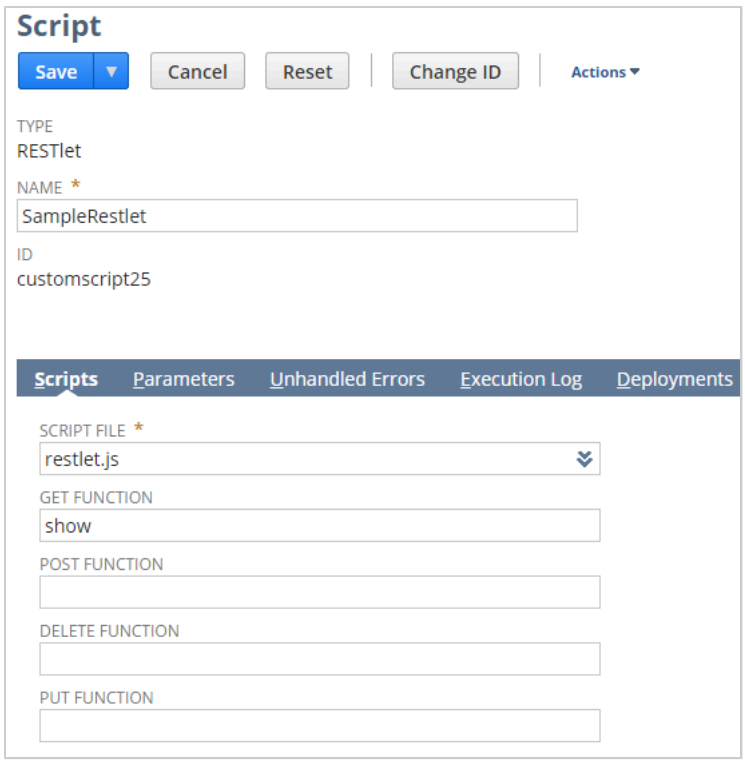
Create the RESTlet Script Record

After you have added a RESTlet file to the File Cabinet, you can create a NetSuite script record.

To create a RESTlet script record:

1. Go to Setup > Customization > Scripts > New, and click **RESTlet**.
2. Complete fields in the script record and save.

   Although you do not need to set every field on the Script record, at a minimum you must provide a **Name** for the Script record, load your SuiteScript file to the record, and specify at least one of the following executing functions on the Scripts tab: GET, POST, DELETE, or PUT.



   You can specify more than one of these functions as desired. These functions should all be in the main script file. If these functions call functions in other script files, you need to list those files as library script files.

   For more details about creating a script record, see Steps for Creating a Script Record.

Define RESTlet Deployment(s)

After you have created a RESTlet script record, you need to define at least one deployment. For details about defining script deployments, see Step 5: Define Script Deployment and Steps for Defining a Script Deployment

You can define multiple deployments per RESTlet.

To define a RESTlet script deployment.

1. Do one of the following:
   - When you save your Script record, you can immediately create a Script Deployment record by selecting **Save and Deploy** from the Script record **Save** button.
   - If you clicked **Save**, immediately afterwards you can click **Deploy Script** on the script record.
   - If you want to update a deployment that already exists, go to Customization > Scripting > Script Deployments and click **Edit**.
2. Complete fields in the script deployment record and click **Save**.

   If you want to debug the script, set the **Status** to **Testing**.

> **Note:** After you have saved a RESTlet deployment, the deployment record includes the URL used to invoke the RESTlet. For a RESTlet called from an externally hosted client, use the **External URL**. For a RESTlet called from a client hosted by NetSuite, use the **URL** that does not include the domain. See RESTlet URL and Domain.

Debugging a RESTlet

You can use the NetSuite Debugger to debug RESTlet code in the same manner that you debug other types of SuiteScript code, as described in the NetSuite Help Center topic Debugging SuiteScript.

- To debug code snippets before you have a RESTlet script record that has been deployed, called on demand debugging, follow the instructions in Ad-hoc Debugging. (Be sure not to include the RESTlet's authorization header in the code snippets to be debugged, as this header can prevent the debugger from working.)
- To debug an existing script that has a defined deployment, called deployed debugging, follow the steps below.

> **Important:** In addition to debugging RESTlet script code, you should test the HTTP request to be sent to the RESTlet. Free tools are available for this purpose. See RESTlet HTTP Testing Tools.

To debug a deployed RESTlet:

1. Before you deploy a RESTlet to be debugged, ensure that the script does not include the HTTP authorization header, as this header can prevent the debugger from working.
2. Ensure that on the script deployment record, the **Status** value is set to **Testing**.
3. Go to Customization > Scripting > Script Debugger, or log in to the debugger domain **https://debugger.netsuite.com**. (See Deployed Debugging for details.)
4. Click the **Debug Existing** button in the main Script Debugger page.

> **Note:** This button only appears when you have deployed scripts with the status is set to **Testing**.

5. Select the RESTlet script that you want to debug in the Script Debugger popup.

   After you click the **Select** option button, the RESTlet's cookies display in a banner.
6. Copy the cookies and paste them into a text file so that you have them available.
7. Click the **Select and Close** button in the Script Debugger popup.

   The main Script Debugger page displays a message that it is waiting for user action.
8. Set the cookies in your client application to the values you copied in step 6, and send the RESTlet request.

   The main Script Debugger page displays the script execution as pending at the NetSuite function `restletwrapper(request)`.
9. You have the following options for debugging your script code:
   - Click the **Step Over** button to begin stepping through each line of code.
   - Add watches and evaluate expressions.
   - Set break points and click the **Run** button to run the code. The Debugger will stop code execution at the first break point set.
   - Set no break points, click the **Run** button, and have the Debugger execute the entire piece of code.

   See the help topic Script Debugger Interface for information about stepping into/out of functions, adding watches, setting and removing break points, and evaluating expressions.

Debugging Timeout Errors

If a timeout error occurs during debugging, check for the following:

- Invalid or missing NetSuite version

  Ensure that you have correctly copied the session cookies, as described in the steps above. These cookies includes a valid NetSuite version.
- Incorrect domain such as https://rest.netsuite.com or https://restlets.api.netsuite.com

  Ensure that you have logged in to https://debugger.netsuite.com.

RESTlet HTTP Testing Tools

You can use the tools of your choosing to test the HTTP request to be sent to a RESTlet.

If you have installed and set up SuiteCloud IDE, a debug client is available for your use. The RESTlet/Suitelet Debug Client enables you to debug deployed RESTlet and Suitelet SuiteScripts with the SuiteCloud IDE Debugger. The client is only accessible after a debug session is started. For details about this client, see the help topic Using the RESTlet/Suitelet Debug Client in SuiteCloud IDE Plug-in for Eclipse. For information about SuiteCloud IDE, see the help topic SuiteCloud IDE Plug-in for Eclipse Usage.

In addition, the following free tools are available:

- **Send HTTP Tool**

  This tool is a free HTTP Request builder that you can use to send an HTTP request to the RESTlet and analyze the response.
- **Fiddler**

This tool is a free Web debugging proxy that you can use to log HTTP traffic, and inspect the HTTP request and response for the RESTlet.

http://www.fiddler2.com/fiddler2/

Warning: The above information about free tools is provided as a courtesy and is not intended as an endorsement of these tools.

Sample RESTlet Code

The following examples provide sample RESTlet code:

- Simple Example to Get Started
- Example Code Snippets of HTTP Methods
- Example RESTlet Called from a Portlet Script
- Example RESTlet Request from Android
- Example RESTlet Request Using nlapiRequestURL

Simple Example to Get Started

Use the following example as a basic GET method test when you are getting started with RESTlets. Try this example with a content-type header value of application/json.

```
function sayhi()
{
    var o = new Object();
    o.sayhi = 'Hello World! ';
    return o;
}
```

Alternatively, you can use the following example with a content-type header value of text/plain:

```
function sayhi()
{
    var o = new Object();
    o.sayhi = 'Hello World! ';
    return JSON.stringify(o);
}
```

Note: RESTlets support the entire SuiteScript API and general SuiteScript features such as debugging. For more information, see Working with RESTlets.

Example Code Snippets of HTTP Methods

The following code snippets provide examples of RESTlet functions.

GET Method

```
// Get a standard NetSuite record
function getRecord(datain)
{
```

```
    return nlapiLoadRecord(datain.recordtype, datain.id); // e.g recordtype="customer", id=769
}
```

Query parameters: `recordtype=customer&id=769`

POST Method

```
// Create a standard NetSuite record
function createRecord(datain)
{
    var err = new Object();

    // Validate if required record type is set in the request
    if (!datain.recordtype)
    {
        err.status = "failed";
        err.message= "missing recordtype";
        return err;
    }

    var record = nlapiCreateRecord(datain.recordtype);

    for (var fieldname in datain)
    {
     if (datain.hasOwnProperty(fieldname))
     {
         if (fieldname != 'recordtype' && fieldname != 'id')
         {
             var value = datain[fieldname];
             if (value && typeof value != 'object') // ignore other type of parameters
             {
                 record.setFieldValue(fieldname, value);
             }
         }
     }
    }
    var recordId = nlapiSubmitRecord(record);
    nlapiLogExecution('DEBUG','id='+recordId);

    var nlobj = nlapiLoadRecord(datain.recordtype,recordId);
    return nlobj;
}
```

Request Payload:

```
{"recordtype":"customer","entityid":"John Doe","companyname":"ABCTools Inc","subsidiary":"1","email":"jdoe@example.com"}
```

DELETE Method

```
// Delete a standard NetSuite record
function deleteRecord(datain)
{
    nlapiDeleteRecord(datain.recordtype, datain.id); // e.g recordtype="customer", id="769"
}
```

Query parameters: `recordtype=customer&id=769`

Example RESTlet Called from a Portlet Script

- Portlet Script Code - Calls RESTlet and Sets Search Criteria
- RESTlet Code - Gets Data based on Portlet Script Criteria

Portlet Script Code - Calls RESTlet and Sets Search Criteria

```
function getAccount () { return '1234567'; }

function getRESTletURL()
{
   return 'https://<accountID>.restlets.api.netsuite.com/app/site/hosting/restlet.nl?script=27&deploy=1&c='+getAccount(); // for phasing
}

function credentials()
{
    this.email='jsmith@example.com';
    this.account=getAccount();
    this.role='37';
    this.password='mysecretpwd';
}

// Portlet function
function displayOpportunities(portlet)
{
   portlet.setTitle('Opportunities');
        var col = portlet.addColumn('id','text', 'ID', 'LEFT');
        col.setURL(nlapiResolveURL('RECORD','opportunity'));
        col.addParamToURL('id','id', true);
        portlet.addColumn('title','text', 'Opportunity', 'LEFT');
        portlet.addColumn('customer','text', 'Customer', 'LEFT');
        portlet.addColumn('salesrep','text', 'Sales Rep', 'LEFT');
        portlet.addColumn('amount','currency', 'Amount', 'RIGHT');
        portlet.addColumn('probability','currency', 'Probability', 'RIGHT');

   var opps = getOpportunites();
   if ( opps != null && opps.length > 0 )
   {
      for ( var i=0; i < opps.length ; i++ )
      {
          portlet.addRow(opps[i]);
      }
   }
}

function getOpportunites()
{
   var url = getRESTletURL() + '&daterange=daysAgo90&&probability=10';
   var cred = new credentials();

   var headers = new Array();
   headers['User-Agent-x'] = 'SuiteScript-Call';
   headers['Authorization'] = 'NLAuth nlauth_account='+cred.account+', nlauth_email='+cred.email+', nlauth_signature='+cred.password+', nlauth_role='+cred.role;
   headers['Content-Type'] = 'application/json';

   var response = nlapiRequestURL( url, null, headers );

        var responsebody = JSON.parse(response.getBody());

   var error = responsebody['error'];
        if (error)
        {
             var code = error.code;
             var message = error.message;
             nlapiLogExecution('DEBUG','failed: code='+code+'; message='+message);
             nlapiCreateError(code, message, false);
        }

        return responsebody['nssearchresult'];
}

function opportunity(internalid, title, probability, amount, customer, salesrep)
{
   this.id = internalid;
   this.title = title;
```

```
    this.probability = probability;
    this.amount = amount;
    this.customer = customer;
    this.salesrep = salesrep;
}
```

RESTlet Code - Gets Data based on Portlet Script Criteria

```
function opportunity(internalid, title, probability, amount, customer, salesrep)
{
    this.id = internalid;
    this.title = title;
    this.probability = probability;
    this.amount = amount;
    this.customer = customer;
    this.salesrep = salesrep;
}

// RESTlet Get NetSuite record data
function getRecords(datain)
{
    var filters = new Array();
    var daterange = 'daysAgo90';
    var projectedamount = 0;
    var probability = 0;
    if (datain.daterange) {
        daterange = datain.daterange;
    }
    if (datain.projectedamount) {
        projectedamount = datain.projectedamount;
    }
    if (datain.probability) {
        probability = datain.probability;
    }

    filters[0] = new nlobjSearchFilter( 'trandate', null, 'onOrAfter', daterange ); // like daysAgo90
    filters[1] = new nlobjSearchFilter( 'projectedamount', null, 'greaterthanorequalto', projectedamount);
    filters[2] = new nlobjSearchFilter( 'probability', null, 'greaterthanorequalto', probability );

    // Define search columns
    var columns = new Array();
    columns[0] = new nlobjSearchColumn( 'salesrep' );
    columns[1] = new nlobjSearchColumn( 'expectedclosedate' );
    columns[2] = new nlobjSearchColumn( 'entity' );
    columns[3] = new nlobjSearchColumn( 'projectedamount' );
    columns[4] = new nlobjSearchColumn( 'probability' );
    columns[5] = new nlobjSearchColumn( 'email', 'customer' );
    columns[6] = new nlobjSearchColumn( 'email', 'salesrep' );
    columns[7] = new nlobjSearchColumn( 'title' );

    // Execute the search and return results

    var opps = new Array();
    var searchresults = nlapiSearchRecord( 'opportunity', null, filters, columns );

    // Loop through all search results. When the results are returned, use methods
    // on the nlobjSearchResult object to get values for specific fields.
    for ( var i = 0; searchresults != null && i < searchresults.length; i++ )
    {
        var searchresult = searchresults[ i ];
        var record = searchresult.getId( );
        var salesrep = searchresult.getValue( 'salesrep' );
        var salesrep_display = searchresult.getText( 'salesrep' );
        var salesrep_email = searchresult.getValue( 'email', 'salesrep' );
        var customer = searchresult.getValue( 'entity' );
        var customer_display = searchresult.getText( 'entity' );
        var customer_email = searchresult.getValue( 'email', 'customer' );
        var expectedclose = searchresult.getValue( 'expectedclosedate' );
        var projectedamount = searchresult.getValue( 'projectedamount' );
        var probability = searchresult.getValue( 'probability' );
        var title = searchresult.getValue( 'title' );
```

```
        opps[opps.length++] = new opportunity( record,
               title,
               probability,
               projectedamount,
               customer_display,
               salesrep_display);
    }

    var returnme = new Object();
    returnme.nssearchresult = opps;
    return returnme;
}
```

Example RESTlet Request from Android

```
HttpPost post = new HttpPost( URL + urlParams );

HttpParams httpParameters = new BasicHttpParams();
HttpConnectionParams.setConnectionTimeout( httpParameters, 20000 );
HttpConnectionParams.setSoTimeout( httpParameters, 42000 );

String authorization = "NLAuth nlauth_account=" + account + ", nlauth_email=" + email + ", nlauth_signature="+password+",
 nlauth_role="+role+"";
post.setHeader( "Authorization", authorization );
post.setHeader( "Content-Type", "application/json" );
post.setHeader( "Accept", "*/*" );

post.setEntity( new StringEntity( "{\"name\":\"John\"}" /*input data*/ ) );

HttpClient client = new DefaultHttpClient( httpParameters );
BufferedReader in = null;

HttpResponse response = client.execute( post );

in = new BufferedReader( new InputStreamReader( response.getEntity().getContent() ) );
StringBuffer sb = new StringBuffer( "" );
String line;
String NL = System.getProperty( "line.separator" );
while ( (line = in.readLine()) != null )
{
                sb.append( line + NL );
}
in.close();
String result = sb.toString();
```

Example RESTlet Request Using nlapiRequestURL

```
function credentials(){
    this.email = "msmith@example.com";
    this.account = "1234567";
    this.role = "37";
    this.password = "******";
}

function replacer(key, value){
    if (typeof value == "number" && !isFinite(value)){
        return String(value);
    }
    return value;
}

//Setting up URL
var url = "https://<accountID>.restlets.api.netsuite.com/app/site/hosting/restlet.nl?script=260&deploy=1";

//Calling credential function
var cred = new credentials();

```

```
//Setting up Headers
var headers = {"User-Agent-x": "SuiteScript-Call",
               "Authorization": "NLAuth nlauth_account=" + cred.account + ", nlauth_email=" + cred.email +
                                ", nlauth_signature= " + cred.password + ", nlauth_role=" + cred.role,
               "Content-Type": "application/json"};

//Setting up Datainput
var jsonobj = {"recordtype": "customer",
               "entityid": "John Doe",
               "companyname": "ABC Company",
               "subsidiary": "1",
               "email": "jdoe@example.com"}

//Stringifying JSON
var myJSONText = JSON.stringify(jsonobj, replacer);

var response = nlapiRequestURL(url, myJSONText, headers);

//The following is being used to place a breakpoint in the debugger
var i=0;

//*************RESTLET Code****************

// Create a standard NetSuite record
function createRecord(datain)
{
    var err = new Object();

    // Validate if required record type is set in the request
    if (!datain.recordtype)
    {
        err.status = "failed";
        err.message = "missing recordtype";
        return err;
    }

    var record = nlapiCreateRecord(datain.recordtype);

    for (var fieldname in datain)
    {
        if (datain.hasOwnProperty(fieldname))
        {
            if (fieldname != 'recordtype' && fieldname != 'id')
            {
                var value = datain[fieldname];
                if (value && typeof value != 'object') // ignore other type of parameters
                {
                    record.setFieldValue(fieldname, value);
                }
            }
        }
    }
    var recordId = nlapiSubmitRecord(record);
    nlapiLogExecution('DEBUG','id='+recordId);

    var nlobj = nlapiLoadRecord(datain.recordtype,recordId);
    return nlobj;
}
```

Sample RESTlet Input Formats

The following examples illustrate how to format input for RESTlets for the JSON content type:

- Customer Record Format
- Item Record Format
- Item Pricing Formats
- Sales Order Record Format

For a general explanation of JSON, see Using JSON Objects and Arrays.

Customer Record Format

JSON

```
{
    "shipcomplete":false,
    "giveaccess":false,
    "globalsubscriptionstatus":"1",
    "isperson":false,
    ... more body fields...,
    "consoldepositbalance":0.00,
    "entityid":"John Doe",
    "addressbook":
    [
        {"zip":"94404","phone":"650-627-1000"},
        {"zip":"94403","phone":"650-627-1001"}
    ],
    "consoloverduebalance":0.00,
    "overduebalance":0.00,
    "creditholdoverride":"AUTO",
    "resubscribelink":"Send Subscription Email"
}
```

XML

```
<shipcomplete>F</shipcomplete>
<globalsubscriptionstatus>1</globalsubscriptionstatus>
<giveaccess>F</giveaccess>
<isperson>F</isperson>
<datecreated>12/19/2010 10:26 pm</datecreated>
<salesrep>-5</salesrep><currency>1</currency>
<lastmodifieddate>12/20/2010 10:14 am</lastmodifieddate>
<id>1185</id>
<emailtransactions>F</emailtransactions>
<balance>0.00</balance>
<entitystatus>13</entitystatus>
<isbudgetapproved>F</isbudgetapproved>

... more fields...

<entityid>John Doe</entityid>
<isinactive>F</isinactive>
<addressbook>
   <zip>94404</zip>
   <phone>650-627-1000</phone>
   <defaultshipping>T</defaultshipping>
   <addrtext>Netsuite100 Mission StreetFoster City CA 94404United States</addrtext>
   <state>CA</state>
   <addressee>Netsuite</addressee>
   <isresidential>F</isresidential>
   <label>Home</label>
   <city>Foster City</city>
   <country>US</country>
   <displaystate>California</displaystate>
   <dropdownstate>CA</dropdownstate>
   <addr1>100 Mission Street</addr1>
   <override>F</override>
   <defaultbilling>T</defaultbilling>
</addressbook>
<addressbook>
   <zip>94403</zip>
   <phone>650-627-1001</phone>
   <defaultshipping>F</defaultshipping>
   <addrtext>Netsuite2955 Campus DriveSan Mateo CA 94403United States</addrtext>
   <state>CA</state>
   <addressee>Netsuite</addressee>
   <isresidential>F</isresidential>
   <label>Work</label>
```

```
    <city>San Mateo</city>
    <country>US</country>
    <displaystate>California</displaystate>
    <dropdownstate>CA</dropdownstate>
    <addr1>2955 Campus Drive</addr1>
    <override>F</override>
    <defaultbilling>F</defaultbilling>
</addressbook>
```

Item Record Format

Note: The format for item pricing varies according to the related features that are enabled in your account. See Item Pricing Formats for examples.

JSON

```
{
    "salesdescription":"Cat 5 Patch Cable 10 ft",
    "vendorname":"CABL0002-64",
    "averagecost":3.50,

    ... more fields...,

"pricing":
[
.
{
                        "currency":
                        {
                                    "name":"British pound",
                                    "internalid":"2"
                        },
                        "pricelist":
                        [
                                    {
                                    "pricelevel":
                                    {
                                                "name":"Alternate Price 1",
                                                "internalid":"2"
                                    },
                                    "price":
                                    [
                                                {
                                                            "price":9.03,
                                                            "quantitylevel":"1",
                                                            "quantity":0
                                                },
                                                {
                                                            "price":8.55,
                                                            "quantitylevel":"2",
                                                            "quantity":10
                                                }
                                    ],
                                    "discount":
                                    {
                                                "name":"-5.0%",
                                                "value":"-5.0%"
                                    }
                        },
                        {
                                    "pricelevel":
                                    {
                                                "name":"Alternate Price 2",
                                                "internalid":"3"
                                    },
                                    "price":
                                    [
                                                {
```

```
                                                    "price":8.55,
                                                    "quantitylevel":"1",
                                                    "quantity":0
                                          },
                                          {
                                                    "price":8.10,
                                                    "quantitylevel":"2",
                                                    "quantity":10
                                          }
                                ],
                                "discount":
                                {
                                          "name":"-10.0%",
                                          "value":"-10.0%"
                                }
                      },
        .
        ]
}
Repeat for other currencies
],



    "productfeed":["FROOGLE","SHOPPING","SHOPZILLA","NEXTAG","YAHOO"],
    "weight":"1",
    "itemid":"Cable - Cat 5, 10 ft",

    ... more fields...,

    "availabletopartners":false,
    "sitecategory":
    [
   {"categorydescription":"Cables",
    "category":"12",
    "isdefault":false}
    ],
    "costingmethoddisplay":"Average",
    "offersupport":true
}
```

XML

```
<salesdescription>Cat 5 Patch Cable 10 ft</salesdescription>
<vendorname>CABL0002-64</vendorname>

... more ...

<excludefromsitemap>F</excludefromsitemap>
<isdonationitem>F</isdonationitem>
<recordtype>inventoryitem</recordtype>
<createddate>10/12/2006 8:37 pm</createddate>
<cost>3.50</cost>
<price4>
   <pricelevelname>Base Price</pricelevelname>
</price4>
<price4>
   <pricelevelname>Alternate Price 3</pricelevelname>
</price4>
<price4>
   <discountdisplay>-10.0%</discountdisplay>
   <pricelevelname>Corporate Discount Price</pricelevelname>
</price4>
<price4>
   <discountdisplay>-15.0%</discountdisplay>
   <pricelevelname>Employee Price</pricelevelname>
</price4>
<price4>
   <pricelevelname>Online Price</pricelevelname>
</price4>
```

```
<price3>
   <pricelevelname>Base Price</pricelevelname>
</price3>
<price3>
   <pricelevelname>Alternate Price 3</pricelevelname>
</price3>
<price3>
   <discountdisplay>-10.0%</discountdisplay>
   <pricelevelname>Corporate Discount Price</pricelevelname>
</price3>
<price3>
   <discountdisplay>-15.0%</discountdisplay>
   <pricelevelname>Employee Price</pricelevelname>
</price3>
<price3>
   <pricelevelname>Online Price</pricelevelname>
</price3>
<price2>
   <price[1]>5.00</price[1]>
   <pricelevelname>Base Price</pricelevelname>
   <price[2]>4.00</price[2]>
</price2>
<price2>
   <pricelevelname>Alternate Price 3</pricelevelname>
</price2>
<price2>
   <discountdisplay>-10.0%</discountdisplay>
   <price[1]>4.50</price[1]>
   <pricelevelname>Corporate Discount Price</pricelevelname>
   <price[2]>3.60</price[2]>
</price2>
<price2>
   <discountdisplay>-15.0%</discountdisplay>
   <price[1]>4.25</price[1]>
   <pricelevelname>Employee Price</pricelevelname>
   <price[2]>3.40</price[2]>
</price2>
<price2>
   <pricelevelname>Online Price</pricelevelname>
</price2>
<price1>
   <price[1]>10.95</price[1]>
   <pricelevelname>Base Price</pricelevelname>
   <price[2]>10.00</price[2]>
</price1>
<price1>
   <pricelevelname>Alternate Price 3</pricelevelname>
</price1>
<price1>
   <discountdisplay>-10.0%</discountdisplay>
   <price[1]>9.86</price[1]>
   <pricelevelname>Corporate Discount Price</pricelevelname>
   <price[2]>9.00</price[2]>
</price1>
<price1>
   <discountdisplay>-15.0%</discountdisplay>
   <price[1]>9.31</price[1]>
   <pricelevelname>Employee Price</pricelevelname>
   <price[2]>8.50</price[2]>
</price1>
<price1>
   <price[1]>10.95</price[1]>
   <pricelevelname>Online Price</pricelevelname>
   <price[2]>10.00</price[2]>
</price1>
<productfeed>FROOGLE</productfeed>
<productfeed>SHOPPING</productfeed>
<productfeed>SHOPZILLA</productfeed>
<productfeed>NEXTAG</productfeed>
<productfeed>YAHOO</productfeed>
<weight>1</weight>
<itemid>Cable - Cat 5, 10 ft</itemid>

```

```
... more fields...

<sitecategory>
    <category>12</category>
    <categorydescription>Cables</categorydescription>

    <isdefault>F</isdefault>
</sitecategory>
<offersupport>T</offersupport>
```

Item Pricing Formats

The format for item pricing varies according to which of the following features are enabled in your account: Multiple Prices, Quantity Pricing, and Multiple Currencies. The following examples show the JSON format for item pricing when these features are enabled.

- Single Price (no additional pricing features enabled)
- Multiple Prices Only Enabled
- Quantity Pricing Only Enabled
- Multiple Prices, Multiple Currencies Enabled
- Multiple Prices, Quantity Pricing, Multiple Currencies Enabled

Single Price (no additional pricing features enabled)

```
"pricing":
[
    {
        "pricelist":
        [
            {
                "price":
                [
                    {"price":100.00,"quantitylevel":"1","quantity":0}
                ]
            }
        ],
        "currency":{"name":"USA","internalid":"1"}
  }
]
```

Multiple Prices Only Enabled

```
"pricing":
[
    {
        "pricelist":
        [
            {
                "pricelevel":{"name":"Base Price","internalid":"1"},
                "price":
                [
                    {"price":100.00,"quantitylevel":"1","quantity":0}
                ]
            },
            {
                "pricelevel":{"name":"Alternate Price 1","internalid":"2"},
                "price":
                [
```

```
                    {"price":99.00,"quantitylevel":"1","quantity":0}
                ]
            },
            {
                "pricelevel":{"name":"Alternate Price 2","internalid":"3"},
                "price":
                [
                    {"price":98.00,"quantitylevel":"1","quantity":0}
                ]
            },
            {
                "pricelevel":{"name":"Alternate Price 3","internalid":"4"},
                "price":
                [
                    {"price":97.00,"quantitylevel":"1","quantity":0}
                ]
            },
            {
                "pricelevel":{"name":"Online Price","internalid":"5"},
                "price":
                [
                    {"price":96.00,"quantitylevel":"1","quantity":0}
                ]
            }
        ],
        "currency":{"name":"USA","internalid":"1"}
    }
]
```

Quantity Pricing Only Enabled

```
"pricing":
[
    {
        "pricelist":
        [
            {
                "pricelevel":{"name":"Base Price","internalid":"1"},
                "price":
                [
                    {"price":100.00,"quantitylevel":"1","quantity":0},
                    {"price":95.00,"quantitylevel":"2","quantity":100},
                    {"price":90.00,"quantitylevel":"3","quantity":150},
                    {"price":85.00,"quantitylevel":"4","quantity":200},
                    {"price":80.00,"quantitylevel":"5","quantity":250}
                ]
            },
            {
                "pricelevel":{"name":"Alternate Price 1","internalid":"2"},
                "price":
                [
                    {"price":99.00,"quantitylevel":"1","quantity":0},
                    {"price":94.00,"quantitylevel":"2","quantity":100},
                    {"price":89.00,"quantitylevel":"3","quantity":150},
                    {"price":84.00,"quantitylevel":"4","quantity":200},
                    {"price":79.00,"quantitylevel":"5","quantity":250}
                ]
            },
            {
                "pricelevel":{"name":"Alternate Price 2","internalid":"3"},
                "price":
                [
                    {"price":98.00,"quantitylevel":"1","quantity":0},
                    {"price":93.00,"quantitylevel":"2","quantity":100},
                    {"price":88.00,"quantitylevel":"3","quantity":150},
                    {"price":83.00,"quantitylevel":"4","quantity":200},
                    {"price":78.00,"quantitylevel":"5","quantity":250}
                ]
             },
             {
```

```
                "pricelevel":{"name":"Alternate Price 3","internalid":"4"},
                "price":
                [
                    {"price":97.00,"quantitylevel":"1","quantity":0},
                    {"price":92.00,"quantitylevel":"2","quantity":100},
                    {"price":87.00,"quantitylevel":"3","quantity":150},
                    {"price":82.00,"quantitylevel":"4","quantity":200},
                    {"price":77.00,"quantitylevel":"5","quantity":250}
                ]
            },
            {
                "pricelevel":{"name":"Online Price","internalid":"5"},
                "price":
                [
                    {"price":96.00,"quantitylevel":"1","quantity":0},
                    {"price":91.00,"quantitylevel":"2","quantity":100},
                    {"price":86.00,"quantitylevel":"3","quantity":150},
                    {"price":81.00,"quantitylevel":"4","quantity":200},
                    {"price":76.00,"quantitylevel":"5","quantity":250}
                ]
            }
        ],
        "currency":{"name":"USA","internalid":"1"}
    }
]
```

Multiple Prices, Multiple Currencies Enabled

```
"pricing":
[
    {
        "pricelist":
        [
            {
                "pricelevel":{"name":"Base Price","internalid":"1"},
                "price":[{"price":110.00,"quantitylevel":"1","quantity":0}]
            },
            {
                "pricelevel":{"name":"Alternate Price 1","internalid":"2"},
                "price":[{"price":105.00,"quantitylevel":"1","quantity":0}]
            },
            {
                "pricelevel":{"name":"Alternate Price 2","internalid":"3"},
                "price":[{"price":100.00,"quantitylevel":"1","quantity":0}]
            },
            {
                "pricelevel":{"name":"Alternate Price 3","internalid":"4"},
                "price":[{"price":95.00,"quantitylevel":"1","quantity":0}]
            },
            {
                "pricelevel":{"name":"Online Price","internalid":"5"},
                "price":[{"price":90.00,"quantitylevel":"1","quantity":0}]
            }
        ],
        "currency":{"name":"British pound","internalid":"2"}
    },
    {
        "pricelist":
        [
            {"pricelevel":{"name":"Base Price","internalid":"1"},"price":[{"price":100.00,"quantitylevel":"1","quantity":0}]},
            {"pricelevel":{"name":"Alternate Price 1","internalid":"2"},"price":[{"price":99.00,"quantitylevel":"1","quanti
ty":0}]},
            {"pricelevel":{"name":"Alternate Price 2","internalid":"3"},"price":[{"price":98.00,"quantitylevel":"1","quanti
ty":0}]},
            {"pricelevel":{"name":"Alternate Price 3","internalid":"4"},"price":[{"price":97.00,"quantitylevel":"1","quanti
ty":0}]},
            {"pricelevel":{"name":"Online Price","internalid":"5"},"price":[{"price":96.00,"quantitylevel":"1","quantity":0}]}
        ],
        "currency":{"name":"USA","internalid":"1"}
}
```

Multiple Prices, Quantity Pricing, Multiple Currencies Enabled

```
"pricing":
[
    {
        "pricelist":
        [
            {
                "pricelevel":{"name":"Base Price","internalid":"1"},
                "price":
                [
                    {"price":110.00,"quantitylevel":"1","quantity":0},
                    {"price":105.00,"quantitylevel":"2","quantity":100},
                    {"price":100.00,"quantitylevel":"3","quantity":150},
                    {"price":95.00,"quantitylevel":"4","quantity":200},
                    {"price":90.00,"quantitylevel":"5","quantity":250}
                ]
            },
            {
                "pricelevel":{"name":"Alternate Price 1","internalid":"2"},
                "price":
                [
                    {"price":105.00,"quantitylevel":"1","quantity":0},
                    {"price":100.00,"quantitylevel":"2","quantity":100},
                    {"price":95.00,"quantitylevel":"3","quantity":150},
                    {"price":90.00,"quantitylevel":"4","quantity":200},
                    {"price":85.00,"quantitylevel":"5","quantity":250}
                ]
            },
            {
                "pricelevel":{"name":"Alternate Price 2","internalid":"3"},
                "price":[{"price":100.00,"quantitylevel":"1","quantity":0},{"price":95.00,"quantitylevel":"2","quantity":100},
{"price":90.00,"quantitylevel":"3","quantity":150},{"price":85.00,"quantitylevel":"4","quantity":200},{"price":80.00,"quantitylev
el":"5","quantity":250}]
            },
            {
                "pricelevel":{"name":"Alternate Price 3","internalid":"4"},
                "price":[{"price":95.00,"quantitylevel":"1","quantity":0},{"price":90.00,"quantitylevel":"2","quantity":100},
{"price":85.00,"quantitylevel":"3","quantity":150},{"price":80.00,"quantitylevel":"4","quantity":200},{"price":75.00,"quantitylev
el":"5","quantity":250}]
            },
            {
                "pricelevel":{"name":"Online Price","internalid":"5"},
                "price":[{"price":90.00,"quantitylevel":"1","quantity":0},{"price":85.00,"quantitylevel":"2","quantity":100},
{"price":80.00,"quantitylevel":"3","quantity":150},{"price":75.00,"quantitylevel":"4","quantity":200},{"price":70.00,"quantitylev
el":"5","quantity":250}]
            }
        ],
        "currency":{"name":"British pound","internalid":"2"}
    },
    {
        "pricelist":
        [
            {
                "pricelevel":{"name":"Base Price","internalid":"1"},
                "price":[{"price":100.00,"quantitylevel":"1","quantity":0},{"price":95.00,"quantitylevel":"2","quantity":100},
{"price":90.00,"quantitylevel":"3","quantity":150},{"price":85.00,"quantitylevel":"4","quantity":200},{"price":80.00,"quantitylev
el":"5","quantity":250}]
            },
            {
                "pricelevel":{"name":"Alternate Price 1","internalid":"2"},
                "price":[{"price":99.00,"quantitylevel":"1","quantity":0},{"price":94.00,"quantitylevel":"2","quantity":100},
{"price":89.00,"quantitylevel":"3","quantity":150},{"price":84.00,"quantitylevel":"4","quantity":200},{"price":79.0","quantitylev
el":"5","quantity":250}]
            },
            {
                "pricelevel":{"name":"Alternate Price 2","internalid":"3"},
                "price":[{"price":98.0","quantitylevel":"1","quantity":0},{"price":93.00,"quantitylevel":"2","quantity":100},
{"price":88.00,"quantitylevel":"3","quantity":150},{"price":83.00,"quantitylevel":"4","quantity":200},{"price":78.00,"quantitylev
el":"5","quantity":250}]
            },
            {
```

```
                "pricelevel":{"name":"Alternate Price 3","internalid":"4"},
                "price":[{"price":97.00,"quantitylevel":"1","quantity":0},{"price":92.00,"quantitylevel":"2","quantity":100},
{"price":87.00,"quantitylevel":"3","quantity":150},{"price":82.00,"quantitylevel":"4","quantity":200},{"price":77.00,"quantitylev
el":"5","quantity":250}]
            },
            {
                "pricelevel":{"name":"Online Price","internalid":"5"},
                "price":[{"price":96.00,"quantitylevel":"1","quantity":0},{"price":91.00,"quantitylevel":"2","quantity":100},
{"price":86.00,"quantitylevel":"3","quantity":150},{"price":81.00,"quantitylevel":"4","quantity":200},{"price":76.00,"quantitylev
el":"5","quantity":250}]
            }
        ],
        "currency":{"name":"USA","internalid":"1"}
    }
]
```


Sales Order Record Format

JSON


```
{
   "total":64.04,
   "altshippingcost":5.67,
   "taxtotal":4.45,
   "tranid":"120",
   "orderstatus":"E",
   "shipcomplete":false,
   "discounttotal":0.00,
   "entity":"76",
   "billaddress":"Doug Fabre\r\nChess\r\nChess Art Gallery\r\n150 N Ocean Dr\r\nMonterey CA 93940",
   "salesrep":"-5",
   "ccapproved":false,
   "linkedtrackingnumbers":["1Z6753YA0394527573","1Z6753YA0394249981"],
   "shipmethod":"92",
   "exchangerate":1.00
   "lastmodifieddate":"1/9/2011 11:34 pm",
   "taxrate":"8.25%",
   "id":"769",
   "shipaddresslist":"55",
   "istaxable":true,
   "tobefaxed":false,
   "altsalestotal":0.00,
   "getauth":false,
   "tobeprinted":false,
   "shippingcost":5.67,
   "recordtype":"salesorder",
   "trandate":"10/14/2006",
   "fax":"831-555-5230",
   "customform":"88",
   "links":
   [
      {"trandate":"10/14/2006","tranid":"8","type":"Item Fulfillment","linktype":"Receipt/Fulfillment"}
   ],
   "taxitem":"-112",
   "custbody1":"831-555-5229",
   "custbody2":"Do not leave the item outside the house",
   "shipdate":"10/14/2006",
   "createddate":"10/14/2006 2:58 pm",
   "subtotal":53.92,
   "currencyname":"USA",
   "revenuestatus":"A",
   "saleseffectivedate":"10/14/2006",
   "email":chessart@example.com,
   "item":
   [
      {
         "isclosed":false,"fromjob":false,"amount":8.96,"rate":8.96,"price":"2",
         "istaxable":"T","description":"10 ft Serial Cable DB25M DB25F",
```

```
         "custcol6":429,"custcol7":2.5,
         "item":"46","quantity":1,"isestimate":false,"commitinventory":"1",
         "options":
         {
            "CUSTCOL3":792,"CUSTCOL1":4
         }
      },
      {
         "isclosed":false,"fromjob":false,"amount":44.96,"rate":44.96,"price":"2",
         "istaxable":true,
         "item":"80","quantity":1,"isestimate":false,"commitinventory":"1"
      }
   ],
   "excludecommission":false,
   "shipaddress":"Chess\r\nChess Art Gallery\r\n150 N Ocean Dr\r\nMonterey CA 93940","tobeemailed":false
}
```

XML

```
<altshippingcost>5.67</altshippingcost>
<total>64.04</total>
<taxtotal>4.45</taxtotal>
<orderstatus>E</orderstatus>
<tranid>120</tranid>
<shipcomplete>F</shipcomplete>
<discounttotal>0.00</discounttotal>
<entity>76</entity>
<billaddress>Doug FabreChessChess Art Gallery150 N Ocean DrMonterey CA 93940</billaddress>
<salesrep>-5</salesrep>
<linkedtrackingnumbers>1Z6753YA0394527573</linkedtrackingnumbers>
<linkedtrackingnumbers>1Z6753YA0394249981</linkedtrackingnumbers>
<ccapproved>F</ccapproved>
<shipmethod>92</shipmethod>
<exchangerate>1.00</exchangerate>
<lastmodifieddate>1/9/2011 11:34 pm</lastmodifieddate>
<taxrate>8.25</taxrate>
<id>769</id>
<shipaddresslist>55</shipaddresslist>
<istaxable>T</istaxable>
<tobefaxed>F</tobefaxed>
<altsalestotal>0.00</altsalestotal>
<getauth>F</getauth>
<tobeprinted>F</tobeprinted>
<shippingcost>5.67</shippingcost>
<recordtype>salesorder</recordtype>
<trandate>10/14/2006</trandate>
<fax>831-555-5230</fax>
<customform>88</customform>
<custbody1>831-555-5229</custbody1>
<custbody2>Do not leave the item outside the house</custbody2>
<shipdate>10/14/2006</shipdate>
<taxitem>-112</taxitem>
<links>
   <trandate>10/14/2006</trandate>
   <tranid>8</tranid>
   <type>Item Fulfillment</type>
   <linktype>Receipt/Fulfillment</linktype>
</links>
<createddate>10/14/2006 2:58 pm</createddate>
<subtotal>53.92</subtotal>
<currencyname>USA</currencyname>
<revenuestatus>A</revenuestatus>
<saleseffectivedate>10/14/2006</saleseffectivedate>
<email>chessart@example.com</email>
<excludecommission>F</excludecommission>
<item>
     <amount>8.96</amount>
      <fromjob>F</fromjob>
     <isclosed>F</isclosed>
      <price>2</price>
```

```
    <rate>8.96</rate>
     <description>10 ft Serial Cable DB25M DB25F</description>
     <istaxable>T</istaxable>
     <item>46</item>
     <quantity>1</quantity>
     <commitinventory>1</commitinventory>
  <custcol6>429</custcol6>
     <custcol7>2.5</custcol7>
     <isestimate>F</isestimate>
     <options>
          <CUSTCOL3>792</CUSTCOL3>
          <CUSTCOL1>4</CUSTCOL1>
  </options>
</item>
<item>
  <amount>44.96</amount>
    <fromjob>F</fromjob>
     <isclosed>F</isclosed>
     <price>2</price>
     <rate>44.96</rate>
     <istaxable>T</istaxable>
     <item>80</item>
     <quantity>1</quantity>
     <commitinventory>1</commitinventory>
     <isestimate>F</isestimate>
</item>
<shipaddress>ChessChess Art Gallery150 N Ocean DrMonterey CA 93940</shipaddress>
<tobeemailed>F</tobeemailed>
```

RESTlet Status Codes and Error Message Formats

For details about RESTlet errors, see:

- Success Code
- Error Codes
- Notes about RESTlet Errors
- Error Message Formatting
- Error for Incorrect URL

Success Code

RESTlets support the following HTTP success code:

- **200 OK:** The RESTlet request was executed successfully.

  The response depends on the request method used. For a GET request, the response contains an entity corresponding to the requested resource. For a POST request the response contains an entity describing or containing the result of the action

Error Codes

RESTlets support the following HTTP error codes:

- **302 Moved Temporarily:** The request was sent to a different data center than the data center in which your company's account resides. When you receive a 302 response, you must recalculate the signature on the request to the correct data center, because the signature is also computed from URL.
- **400 BAD_REQUEST:** The RESTlet request failed with a user error.
- **401 UNAUTHORIZED:** There is not a valid NetSuite login session for the RESTlet calls.

- **403 FORBIDDEN:** RESTlet request sent to invalid domain, meaning a domain other than https://<accountID>.restlets.api.netsuite.com.
- **404 NOT_FOUND:** A RESTlet script is not defined in the RESTlet request.
- **405 METHOD_NOT_ALLOWED:** The RESTlet request method is not valid.
- **415 UNSUPPORTED_MEDIA_TYPE:** An unsupported content type was specified. (Only JSON and text are allowed.)
- **500 INTERNAL_SERVER_ERROR (unexpected errors):** Occurs for non-user errors that cannot be recovered by resubmitting the same request.

  If this type of error occurs, contact Customer Support to file a case.
- **503 SERVICE_UNAVAILABLE:** The NetSuite database is offline or a database connection is not available.

For additional information about HTTP status codes, see http://www.w3schools.com/tags/ref_httpmessages.asp

Notes about RESTlet Errors

- Any errors encountered at run time that are unhandled return a 400 error. If the user code catches the error, a 200 error is returned.
- An unexpected error is returned with an error ID, for example:

  `Code = UNEXPECTED_ERROR Msg = An unexpected error occurred. Error ID: fevsjhv41tji2juy3le73`
- An INVALID_REQUEST error is returned due to malformed syntax in the OAuth header. For example, when the signature method, version, or timestamp parameters are rejected.
- An INVALID_LOGIN_ATTEMPT error is returned when the nonce, consumer key, token, or signature in the OAuth header is invalid.
- The DELETE method is not expected to return anything. In this case, the message is returned: `Return was ignored in DELETE operation.`
- If users specify a content type other than JSON or TEXT, a 415 error is returned with the following message: `Invalid content type. You can only use application/json, application/xml or text/plain with RESTlets.`
- If users do not provide two-factor authentication when it is required, the following error is returned: `{"error" : {"code" : "TWO_FA_REQD", "message" : "Two-Factor Authentication required"}}`
- If users provide data in a format different from specified type, the following error is returned with one of the following messages:

```
Error code = INVALID_RETURN_DATA_FORMAT
Error message = Invalid data format. You should return TEXT.
Error message = Invalid data format. You should return a JavaScript object.
```

Error Message Formatting

The following examples show RESTlet error message formatting for JSON and text content types.

JSON

```
{
    "error":
   {
      "code":"SSS_INVALID_SCRIPTLET_ID",
```

```
      "message":"That Suitelet is invalid, disabled, or no longer exists."
   }
}
```

XML

```
<error>
        <code>SSS_INVALID_SCRIPTLET_ID</code>
        <message>That Suitelet is invalid, disabled, or no longer exists.</message>
</error>
```

Text

```
<error code: SSS_INVALID_SCRIPTLET_ID
error message: That Suitelet is invalid, disabled, or no longer exists.
```

Error for Incorrect URL

If you receive the following error, make sure that the URL is correct and that it points to the correct RESTlet script ID.

`SSS_INVALID_SCRIPTLET_ID: That Suitelet is invalid, disabled, or no longer exists.`

Tracking and Managing RESTlet Activity

If you use token-based authentication, you have the ability to track calls that were made by external applications to RESTlets hosted in your NetSuite account. You can track RESTlet activity by using integration records.

When using token-based authentication (TBA), you create an integration record to represent each external application that calls RESTlets. You use the integration record to generate the consumer key and secret needed by the application to authenticate. You can also use the record to do the following:

- Block requests that reference the record's consumer key. You can block requests by setting the record's State field to Blocked.
- Enable or disable token-based authentication for an external application. Note that an integration record can also be used to track web services requests. An additional authentication option on the record, User Credentials, applies only to web services requests. Checking or clearing the User Credentials box has no effect on whether the application can call RESTlets. The User Credentials option affects only SOAP web services calls.

Integration records are located at Setup > Integration > Manage Integrations. The record can be accessed only by administrative users.

For full details on using the integration record to set up token-based authentication, see the help topic Token-based Authentication (TBA).

For more information about using integration records in conjunction with RESTlets, see the following topics:

- Ownership of Integration Records
- Using the RESTlets Execution Log
- Finding System Notes for an Integration Record

- More Information

Ownership of Integration Records

When you create an integration record, it is automatically available to you in your NetSuite account. Your NetSuite account is considered to be the owner of the integration record, and the record is fully editable by administrators in your account.

You can also install records in your account that were created elsewhere. For example, an integration record can be packaged, distributed, and installed outside the account where it was created. If you install a SuiteApp that includes an integration record, the record is considered to be an installed record. It is owned by a different NetSuite account. On such records, you can make changes to only two fields: the Note field and the State field. All other fields, including the authentication and Description fields, can be changed only by an authorized user in the account that owns the record. When the owner makes changes to these fields, the new settings are pushed automatically to your account. These changes are not reflected in the system notes that appear in your account.

Using the RESTlets Execution Log

Each integration record includes a subtab labeled RESTlets under the Execution Log subtab. This log lists RESTlet calls that are uniquely identified with that integration record. That is, the log includes those requests that use token-based authentication and reference the integration record's consumer key.

Note: Calls made using the NLAuth method of authentication are not logged on any integration record.

For each logged request, the RESTlets Execution Log includes details such as the following:

- The date and time that the call was made.
- The duration of the request.
- The email address of the user who made the request.
- The action taken.
- The corresponding script ID and deployment ID.

Tracking Changes to Integration Records

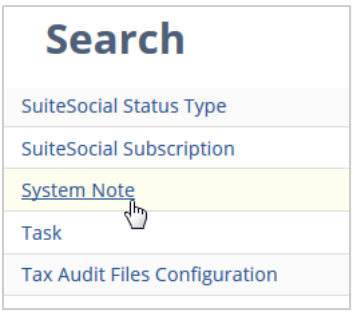
If you want to review changes that were made to an integration record, you can refer to the system notes for that record. System notes are used to track events such as the creation of the record, the initial values of its fields, and subsequent updates. For example, if a user changed the State field from Blocked to Enabled, a system note would provide a record of that change.

For each event, the system records details such as the ID of the user who made the change and the timestamp of the change. If a user assigns a value to a field that already had a value, the system note also shows the field's former setting.

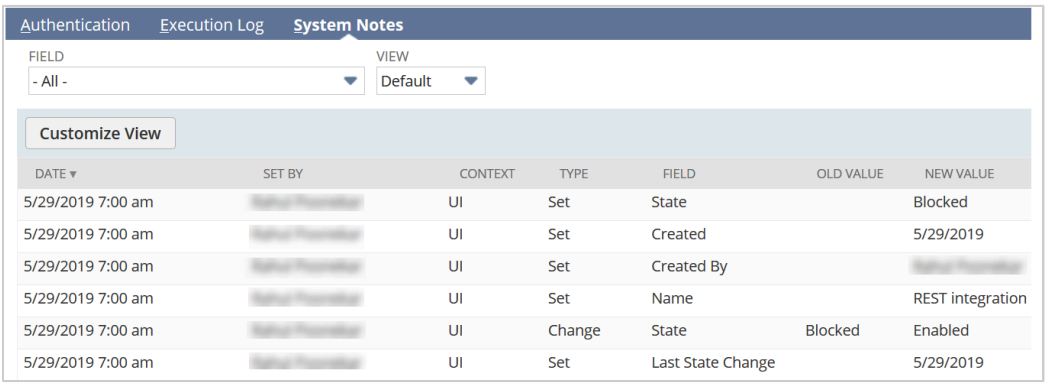
Be aware that in general, system notes are created only for those fields that you are permitted to change. For additional details, see the help topic Special Cases Related to System Notes Logging. Be aware that this section is part of the SOAP Web Services Platform Guide. Some of the detail in this guide pertains only or primarily to web services.

You can locate system notes for integration records in either of the following ways:

- By using the System Note search type, at Reports > New Search.



- By clicking on the System Notes subtab of any integration record.



More Information

For more information about managing integration records, see the help topic Integration Management, which is part of the SOAP Web Services Platform Guide. Be aware that some of the detail in this guide pertains only or primarily to SOAP web services:

- Adding an Integration Record
- Regenerating a Consumer Key and Secret
- Distributing by Bundling
- Removing Integration Records
- Special Cases Related to System Notes Logging

Scheduled Scripts

- Overview of Scheduled Script Topics
- What Are Scheduled Scripts?
- When Will My Scheduled Script Execute?
- Submitting a Script for Processing
- Creating Multiple Deployments for a Scheduled Script
- Using nlapiScheduleScript to Submit a Script for Processing
- Understanding Scheduled Script Deployment Statuses
- Scheduled Scripts on Accounts with Multiple Processors (SuiteCloud Plus)
- Executing a Scheduled Script in Certain Contexts
- Setting Recovery Points in Scheduled Scripts

- Understanding Memory Usage in Scheduled Scripts
- Monitoring a Scheduled Script's Runtime Status
- Monitoring a Scheduled Script's Governance Limits
- Scheduled Script Samples
- Scheduled Script Best Practices
- Best Practice: Handling Server Restarts in Scheduled Scripts (SuiteScript 1.0)

Overview of Scheduled Script Topics

The following topics are covered in this section. If you are not familiar with NetSuite scheduled scripts, you should read these topics. They do not need to be read in order. However, if you are learning about NetSuite scheduled scripts, you should read the general overview topics first.

General overview of scheduled scripts

- What Are Scheduled Scripts?
- When Will My Scheduled Script Execute?

Submitting scheduled scripts for processing

- Submitting a Script for Processing
- Creating Multiple Deployments for a Scheduled Script
- Using nlapiScheduleScript to Submit a Script for Processing
- Understanding Scheduled Script Deployment Statuses
- Scheduled Scripts on Accounts with Multiple Processors (SuiteCloud Plus)

Scheduled script optimization, recovery points, and monitoring

- Executing a Scheduled Script in Certain Contexts
- Setting Recovery Points in Scheduled Scripts
- Understanding Memory Usage in Scheduled Scripts
- Monitoring a Scheduled Script's Runtime Status
- Monitoring a Scheduled Script's Governance Limits

Scheduled script samples and best practices

- Scheduled Script Samples
- Scheduled Script Best Practices
- Best Practice: Handling Server Restarts in Scheduled Scripts (SuiteScript 1.0)

Important: All NetSuite accounts are provided a **single** SuiteCloud Processor for running scripts. You can increase the number of processors by purchasing a SuiteCloud Plus license. See the help topics SuiteCloud Plus Settings and Scheduled Scripts on Accounts with Multiple Processors (SuiteCloud Plus) for more information.

You can use SuiteCloud Development Framework (SDF) to manage scheduled scripts as part of file-based customization projects. For information about SDF, see the help topic SuiteCloud Development Framework Overview. You can use the Copy to Account feature to copy an individual scheduled script to

another of your accounts. Each scheduled script page has a clickable Copy to Account option in the upper right corner. For information about Copy to Account, see the help topic Copy to Account Overview.

What Are Scheduled Scripts?

Scheduled scripts run on the NetSuite server. Compared to all other NetSuite script types, scheduled scripts are given a higher limit of usage governance (10,000 units, as opposed to 1,000 units for most other script types). Therefore, scheduled scripts are ideal for long running tasks and batch jobs.

You can submit scheduled scripts for processing on an on-demand basis. You can also submit scheduled scripts for processing at a future time, or recurring future times.

NetSuite enables you to submit scheduled scripts using the scheduled script Script Deployment page in the UI. You can also call nlapiScheduleScript to submit a script for processing.

To understand when your script will execute after it has been submitted, see When Will My Scheduled Script Execute?

Important: Be aware that scheduled scripts that do mass modifications of data may prompt the system to generate automatic email notifications. For example, if the data being modified is on an Activity record that is set to automatically send an email each time the record is updated, an email will be sent when the scheduled script runs and data is updated.

When Will My Scheduled Script Execute?

Whether you submit a scheduled script manually (through the Script Deployment page in the UI) or programmatically (using nlapiScheduleScript), all scheduled script instances are stored as pending script jobs. For scheduled scripts, there is one job per script instance.

Next, a scheduler distributes the script jobs to the SuiteCloud Processor in a particular order. Processing of the next script in order starts as soon as a free processor in the pool is available.

Note that there is a single SuiteCloud Processor for all scripts in your company's NetSuite account, and one SuiteCloud Processor by default. SuiteCloud Plus accounts have access to a SuiteCloud Processor containing multiple processors.

As soon as one script completes, the corresponding processor is available to process the next script in order. Although multiple scheduled scripts can be executing in the SuiteCloud Processor at the same time, only a single script can be executed by one processor at any particular time. In SuiteCloud Plus accounts, where multiple processors are available, scripts can be executed in parallel.

When your script executes depends on the number of unfinished jobs in the system. It also depends on how long-running those script jobs are, and their assigned priority.

Important: Even when there are no other scripts pending or running, and you submit a script for execution, there may be a short system delay before your script is executed.

When you use the Script Deployment page to schedule the deployment of a script, the times you set on the Schedule subtab are the times the script is being submitted for processing. **The times you set on the Schedule subtab are not necessarily the times the script will execute**.

Example

Starting on February 13, 2017, a script is submitted for daily processing at 10:00pm, and then every 15 minutes after that until midnight of the same day. Be aware that the schedule starts over each day, so the start time does not apply only to the starting day, but to each cycle of the schedule. Therefore, on Feb 14, 2017, the script is not be submitted until 10pm. Following 10pm, the script is submitted

at the set interval of 15 minutes until midnight of Feb 14th. The script deployment schedule does not mean the script is executed precisely at 10:00pm, 10:15pm, 10:30pm, and so on. It means the script is submitted for processing, but its runtime is based on the order in which pending jobs are sent to the SuiteCloud Processor determined by the scheduler.

To learn more about how to deploy a script, see Submitting a Script for Processing.

Important: All companies using NetSuite are provided a **single** SuiteCloud Processor for running their scheduled scripts. You can increase the number of processors by purchasing a SuiteCloud Plus license. See the help topics SuiteCloud Plus Settings and Scheduled Scripts on Accounts with Multiple Processors (SuiteCloud Plus) for more information.

Submitting a Script for Processing

You can create a script deployment for on-demand processing. Alternatively, you can schedule the deployment so that the script is submitted at a scheduled time in the future, or recurring times in the future.

Important: After a script is submitted for processing, there may be a short system delay before the script is executed, even if no scripts are before it. If there are scripts already waiting to be executed, the script may need to wait until other scripts have completed.

See these topics for more details:

- Creating an On Demand Deployment of a Scheduled Script
- Creating a Future or Recurring Deployment of a Scheduled Script

Note: You can create multiple on demand or future deployments of the same script. There are specific use cases for why you may want to create multiple deployments for the same scheduled script. See Creating Multiple Deployments for a Scheduled Script to learn more.

Creating an On Demand Deployment of a Scheduled Script

Scheduled scripts can be submitted for processing on an on demand basis. To do this, you will use the **Save and Execute** command on the Script Deployment page. The Status field on the Script Deployment page must be set to either **Not Scheduled** or **Testing**.

Note: To understand why you may want to set the deployment status to either Not Scheduled or Testing for an on demand deployment, see Understanding the Difference Between Not Scheduled and Testing Deployment Statuses.

Deployments set to **Not Scheduled** can also be submitted for processing on-demand with a call to nlapiScheduleScript. For details, see Using nlapiScheduleScript to Submit a Script for Processing.

To initiate an on demand deployment of a scheduled script:

1. After creating your SuiteScript JavaScript file, create a new Script record for your script. Go to Customization > Scripting > Scripts > New.
2. On the **Upload Script File** page, for the **Script File** field, choose your script file.
3. Select **Create Script Record**.
4. Select **Scheduled**.
5. On the Script page, provide a **Name** for the Script record, and specify the script's executing function.

6. Select **Save**.
7. Click the **Deploy Script** button on the page that appears.
8. When the Script Deployment page opens, select the **Deployed** box if it is not already checked.
9. Select **Not Scheduled** (or Testing) from the **Status** field.
10. Select the **Single Event** radio button.
11. Select **Save**.
12. Select **Edit**, and then from the **Save** dropdown list, select **Save and Execute**.

Important: Even if you initiate an on demand deployment of a script that immediately submits the script for processing, this does not mean the script will execute right away. After a script is submitted for processing, there may be a short system delay before the script is executed, even if no scripts are before it. If there are scripts already waiting to be executed, the script may wait to be executed until other scripts have completed.

Understanding the Difference Between Not Scheduled and Testing Deployment Statuses

To submit a scheduled script on-demand, set the deployment status to either **Not Scheduled** or **Testing** for the following reasons:

Deployment Status	Use Case
Testing	■ You want to test the script by running it immediately. The script will run only in the script owner's account. After setting the deployment status to Testing, click **Save and Execute** on the Script Deployment page to run the script. ■ You want to load the script into the SuiteScript Debugger. Only scheduled scripts with the deployment status set to Testing can be loaded into the Debugger.
Not Scheduled	■ Set to Not Scheduled after all of the scheduling options have been set (time, date, frequency). However, the script is not yet ready to be executed. ■ If you set your scheduling options, set the deployment status to Not Scheduled, and click **Save**, the script will not run at the times you have specified. **Important:** Scripts with a deployment status set to Not Scheduled will run only on an on demand basis. For example, when you click **Save and Execute** from the deployment page, or make a call to nlapiScheduleScript.

Notes:

- If you set the deployment status to **Not Scheduled**, and then select either Daily Event, Weekly Event (or any event type other than **Single Event**), and click **Save and Execute**, the script will still only run one time.
- After a **Not Scheduled** script completes its on demand execution, NetSuite automatically re-sets the deployment status back to **Not Scheduled**.

Creating a Future or Recurring Deployment of a Scheduled Script

Scheduled scripts with a deployment status set to **Scheduled** can be submitted for processing one time at a pre-defined time in the future, or repeatedly on a regular daily, weekly, monthly, or yearly basis for example.

Deployment times can be scheduled with a frequency of every 15 minutes, for example 2:00 pm, 2:15 pm, 2:30 pm, and so on.

When you use the Script Deployment page to create the deployment of a script, the times you set on the Schedule subtab are the times the script is being submitted for processing. **The times you set on the Schedule subtab are not necessarily the times the script will execute**. Script deployment does not mean the script will execute precisely at 2:00 pm, 2:15 pm, 2:30 pm , and so on.

A scheduled script's deployment status should be set to **Scheduled** for the following reasons:

- The script was set to Testing, but is now ready for production.
- The script does not need to be executed immediately.
- The script must run at recurring times.

To initiate a scheduled deployment of a script:

1. After creating your SuiteScript JavaScript file, create a new Script record for your script. Go to Customization > Scripting > Script > New.
2. On the **Upload Script File** page, for the **Script File** field, choose your script file.
3. Select **Create Script Record**.
4. Select **Scheduled**.
5. On the Script page, provide a **Name** for the Script record, and specify the script's executing function.
6. Select **Save**.
7. Select the **Deploy Script** button on the page that appears.
8. When the Script Deployment page opens, click the **Deployed** box if it is not already checked.
9. Select **Scheduled** from the **Status** dropdown list.
10. On the **Schedule** tab, set all deployment options.
11. Click **Save**.

Note: You can create multiple on demand or future deployments of the same script. There are specific use cases for why you may want to create multiple deployments for the same scheduled script. See Creating Multiple Deployments for a Scheduled Script to learn more.

Creating Multiple Deployments for a Scheduled Script

On either the Script page or the Script Deployment page you can create multiple deployments for the same script. See these sections for more information:

- Use Cases for Creating Multiple Deployments
- Steps for Creating Multiple Deployments
- Multiple Deployments and nlapiScheduleScript

Note: You can set unique deployment options for a scheduled scripts on it's **Script** page or **Script Deployment** page.

Use Cases for Creating Multiple Deployments

The following use cases describe possible scenarios in which script owners may want to create multiple deployments for a single scheduled script.

Use Case 1 - Same Script Requires Different Deployment Schedules

Create multiple deployments when you want the same script to execute according to different deployment schedules.

For example, you might have a scheduled script that you want deployed and executed on the last day of every month. This would be your first deployment. You might also want this script to be deployed and executed every Monday morning around 2:00 am. This would be your second, separate deployment for this script.

Use Case 2 - Same Script Receives Multiple Requests for Execution

There may be times when you have concurrent (simultaneous) requests to kick off long-running tasks. In situations like this, you want to be able to schedule the next available deployment for each request that comes in. For example, if you think that at peak load you might receive five requests in a specific time frame, then you would want at least that many script deployments available to ensure that each request gets its own dedicated script deployment.

Note that if one of the deployments already happens to be in progress or pending, the best practice is to take the number of requests you expect over some time period (for example, over a five minute period - which is more than the average script execution time) and then multiply by two to get the number of script deployments you would need to ensure that you can handle the load. In this case you would want to create 10 deployments for the same script.

Use Case 3 - Script May Exceed Unit-based Governance Limits

Another reason to create multiple deployments is if you have a script you think will exceed governance limits. After creating different deployments, you can specify a different deployment through the **deployId** parameter in nlapiScheduleScript(scriptId, deployId, params).

Note: See the help topic SuiteScript Governance and Limits for information about scheduled script governance limits.

Important: All NetSuite accounts are provided a **single** processor for running scripts. You can increase the number of processors by purchasing a SuiteCloud Plus license. See the help topics SuiteCloud Plus Settings and Scheduled Scripts on Accounts with Multiple Processors (SuiteCloud Plus) for more information.

Steps for Creating Multiple Deployments

To schedule multiple deployments for the same script:

1. From the **Deployments** tab on the Script record page, set the deployment options.
2. After setting your deployment values, select **Add**.
3. Create another deployment (if necessary).
4. Optionally, create your own unique deployment ID in the **ID** column. If you do not add your own custom deployment ID, an ID is automatically generated.
5. When finished creating all deployments, click **Save**.

To edit or access each deployment, go to Customization > Scripting > Script Deployments..

You can create additional script deployments from existing deployments. This is useful for quickly adding extra deployments for scheduled scripts.

To create additional deployments from the same deployment:

1. On the Script Deployments page, click **View** on an existing deployment.
2. From the **More Actions** dropdown list, click **Make Copy**.
3. Check the details for the deployment, correcting where necessary.
4. Click **Save**.

Note: Make Copy is only available for scheduled scripts.

Multiple Deployments and nlapiScheduleScript

You can create multiple on-demand deployments of the same script. This allows you to run multiple instances of the same **Not Scheduled** script. The nlapiScheduleScript API will call the first of the script deployments that is ready, and continue to call this same script until each instance of the script is submitted for processing. This means that users can create as many instances of the **Not Scheduled** script through User Event scripts as they have deployments.

In this scenario, you should not set the **deployId** parameter in nlapiScheduleScript. (Being able to specify the deployId would require additional logic to determine which deployments are not already submitted for processing while nlapiScheduleScript does this automatically if deployId is not specified.)

Note: For more information about nlapiScheduleScript, see Using nlapiScheduleScript to Submit a Script for Processing. Also see the API documentation for nlapiScheduleScript(scriptId, deployId, params).

Using nlapiScheduleScript to Submit a Script for Processing

You can programmatically submit a scheduled script for processing using the nlapiScheduleScript API.

Using nlapiScheduleScript you can:

- Resubmit a currently executing scheduled script.
- Call another scheduled script from within a scheduled script.
- Submit a scheduled script for processing from another script such as a user event script or a Suitelet.

Note: The nlapiScheduleScript(scriptId, deployId, params) function may return NULL if the script that was called has not yet been deployed or does not exist in NetSuite.

The ability to call nlapiScheduleScript from within a scheduled script allows developers to **automatically** resubmit their currently executing script for processing. Otherwise, scheduled scripts must be resubmitted manually through the Script Deployment page. See **Example 2** in Scheduled Script Samples for code that shows how to programmatically resubmit a scheduled script.

Developers should call nlapiScheduleScript in a scheduled script if they think the script is coming close to exceeding the 10,000 unit limit allotted to scheduled scripts. The call to nlapiScheduleScript will resubmit the script, and the script can then run to completion without exceeding any governance limits.

Note that if nlapiScheduleScript is used in a scheduled script to call **another** scheduled script, instruction count limits are applied to **each** script separately, since (technically) you are running two different

scheduled scripts. In other words, both “scheduled script **A** ” and “scheduled script **B**,” which was called by “scheduled script **A** ” can **each** consume 10,000 units.

See the help topic SuiteScript Governance and Limits for information about unit-based governance limits.

Note: If submitted scheduled scripts are requested at the same time or rapidly in succession, the scheduled tasks may not proceed to the Queued status.

Understanding Scheduled Script Deployment Statuses

The following describes each of the deployment statuses of a scheduled script deployment. These deployment statuses appear in the Status field of the **Script Deployment** page.

- **Not Scheduled:** Means that the script is not currently scheduled for processing. By clicking Save and Execute on the Script Deployment page, scheduled scripts with a Not Scheduled deployment status are submitted for processing on demand. Scripts with a Not Scheduled deployment status can also be programmatically submitted through a call to nlapiScheduleScript.
- **Scheduled** : Means that the script will be submitted for processing at the time(s) specified on the Schedule subtab of the Script Deployment page. If the submission is set to happen on a recurring basis, the script's deployment status will remain as **Scheduled**, even after the script completes its execution. The script will then be resubmitted at its specified time(s).
- **Testing** : Means that when the scheduled script is executed, it will run only in the script owner's account. Note that when the deployment status is set to Testing, the only way to submit the script for processing is by clicking Save and Execute on the Script Deployment page. You cannot schedule testing times and then click Save.
  - Also note that only scheduled scripts with a deployment status set to Testing can be loaded into the SuiteScript Debugger.

This table summarizes what you can and cannot do with a scheduled script depending on the script's deployment status on the Script Deployment page.

The **second column** states whether the script can be submitted through the UI.

The **third column** states whether the script can be submitted using nlapiScheduleScript.

Deployment Status	UI	nlapiScheduleScript
Not Scheduled	**YES**. You must click the Save and Execute button on the Script Deployment page.	**YES**. See Using nlapiScheduleScript to Submit a Script for Processing for details.
Scheduled	**-**. No action is required by the user. With a deployment status set to Scheduled, the script will run again at the next scheduled time.	**NO**
Testing	**YES**. On the Script Deployment page, users must click Save and Execute if they want to run the script for testing purposes. You cannot schedule testing times and then click Save. Only Save and Execute will run a script that has a Testing status. Also note that only scheduled scripts with a deployment status set to Testing can be loaded into the SuiteScript Debugger. Finally, scheduled scripts set to Testing will run in the script owner's account only.	**NO**

Note: You cannot execute another instance of the same deployment until the current instance completes. A submitted instance can be canceled from the Scheduled Script Status page if it hasn't started processing yet. See Use the Status Page or Status Links.

Executing a Scheduled Script in Certain Contexts

When creating a scheduled script, you can associate a type argument that is passed by the system to the script's executing function. The type argument provides a context for when the scheduled script is invoked.

Important: The type argument is an auto-generated argument passed by the system. You cannot set this as a parameter for a specific deployment like other function arguments.

Valid values for the type argument are:

- **scheduled** - normal execution according to the deployment options specified in the UI
- **ondemand** - the script is executed using a call to nlapiScheduleScript
- **userinterface** - the script is executed using the UI (the Save & Execute button has been clicked)
- **aborted** - re-executed automatically following an aborted execution (system went down during execution)
- **skipped** - executed automatically following downtime during which the script should have been executed

Example

```
function processOrdersCreatedToday( type )
{
 //only execute when run based on the schedule specified on the deployment record in the UI.
  if ( type != 'scheduled' && type != 'skipped' ) return ;

.... //process rest of script

}
```

Setting Recovery Points in Scheduled Scripts

Occasionally when running a scheduled script, a failure may occur. This could be due to a major NetSuite upgrade, or an unexpected failure of the execution environment. Therefore, NetSuite gives developers the ability to create recovery points in scheduled scripts. These recovery points allow the state of the script at a certain point to be saved. In the event of an unexpected system failure, the script can be restarted from the last successful recovery point.

To set a script recovery point, use nlapiSetRecoveryPoint(). When the system restarts, the script will resume where it left off.

Developers can also use nlapiYieldScript(). In addition to setting a recovery point, this API terminates the current script instance, creates a new one, and submits the new instance for processing. When processing resumes, it begins its execution from the specified recovery point.

Possible use cases for nlapiSetRecoveryPoint() and nlapiYieldScript() include:

- If the script is unexpectedly aborted, it can be restarted from the last successful recovery point.
- Pause (yield) the execution of a script at a specified point.
- Governance usage limits have been reached.

- Yield a script because an external resource is temporarily unavailable.

See the API documentation on nlapiSetRecoveryPoint() and nlapiYieldScript() for more details and example usage.

Understanding Memory Usage in Scheduled Scripts

The memory limit for a scheduled script is 50 Megabytes. Therefore, if you have a long-running scheduled script, and you are concerned the script will exceed the 50 MB memory limit, you should use nlapiSetRecoveryPoint() or nlapiYieldScript() APIs to track memory size. This is accomplished by examining the returned size property in the status object returned by nlapiSetRecoveryPoint() and nlapiYieldScript(). Note, however, calling nlapiSetRecoveryPoint() costs 100 governance units. Therefore, you will want to use the API only at key points in your script. The alternative approach is to use nlapiYieldScript(). If the call is successful, the script will yield and then the size property can be examined after the script resumes.

Important: Scripts that are resumed after an nlapiYieldScript() or nlapiSetRecoveryPoint() call will have their governance units reset. However, this does not reset the memory footprint of the script, which will continue to change until the script terminates. Therefore it is possible for the script to stay under the governance limit but exceed the memory size limit, at which point an SS_EXCESSIVE_MEMORY_FOOTPRINT error is thrown during the call to nlapiYieldScript() or nlapiSetRecoveryPoint().

Reducing Script Memory Footprint

The table below shows an estimation of how various parts of a script can consume memory.

Empty Function	111 bytes
Each Instruction within a function	32 bytes
Each local variable reference	32 bytes
Standard mode record (customer)	40 kilobytes (depends on record type)
Dynamic mode record (customer)	460 kilobytes (depends on record type)
Number Instance	32 bytes
String Instance	32 bytes + 4 bytes per character
Empty Object	32 bytes

There are several useful techniques to reduce the memory overhead of scripts. Certain returned objects, for example nlobjRecord and nlobjSearchResult can be quite large. To reduce the memory consumed by these objects, convert them to native JavaScript objects and then operate on those objects instead.

Monitoring a Scheduled Script's Runtime Status

The Scheduled Script Status page shows the current and past runtime statuses of all scheduled scripts that have been executed in your account. There are different ways you can access the Scheduled Script Status page. How you access the page determines the view that the page opens with.

Use the Customization Menu

Access the Scheduled Script Status page through Customization > Scripting > Scheduled Script Status.

This view will show all scheduled scripts in your accounts and each deployment of the script. You can use filtering options at the top of the page to see the runtime status of specific scripts and deployments.

Important: Script execution details are purged after 30 days. Also note that the statuses listed on the Scheduled Script Status page are not related to the statuses that appear on a Script Deployment page. The statuses on Script Deployment pages (Not Scheduled, Scheduled, Testing) indicate the type of deployment. The statuses on the Scheduled Script Status page indicate where the scheduled script is in terms of its execution.

Use the Status Page or Status Links

You can also access the Scheduled Script Status page by clicking the Status Page link (associated with the See Instances field). This link appears on the Script Deployment page.

In both cases, when the Scheduled Script Status page opens, it shows the runtime statuses of all deployed instances of a particular script.

If you want to see the runtime statuses of other deployments of a particular script, use the **Deployment ID** filter to choose another deployment of this script. You can also choose ALL to see the runtime statuses of every scheduled script deployment of this script.

If you want to see the runtime statuses of other scripts, use the **Script** filter to choose another script. Then use the **Deployment ID** filter to specify which deployments of the script you want to see the runtime status for.

Monitoring a Scheduled Script's Governance Limits

The following APIs are helpful when working with and monitoring scheduled scripts:

- nlapiLogExecution(type, title, details)
- nlobjContext methods:
  - setPercentComplete(pct)
  - getPercentComplete()
  - getRemainingUsage()
  - getScriptId()
  - getDeploymentId()

Scheduled Scripts on Accounts with Multiple Processors (SuiteCloud Plus)

The number of SuiteCloud processors and CSV imports available are directly dependent upon the number of SuiteCloud Plus (SC+) licenses a company has. Companies can upgrade their number of processors in the SuiteCloud Processor with the purchase of additional SuiteCloud Plus licenses. SuiteCloud Plus allows larger accounts to run multiple scripts in parallel.

For a summary of SuiteCloud Plus capabilities, see the help topics SuiteCloud Plus Settings and NetSuite Service Tiers. To purchase or learn more about SuiteCloud Plus, contact your NetSuite account manager.

When you upgrade your account to include multiple script processors, multiple scripts can run in parallel. For example, if two scripts are submitted for processing, the second script may begin before the first script is complete. If you have one script with five deployment instances, all of them could run at the same time.

To process multiple instances of the same scheduled script, developers can create multiple deployments for the same script. Script instances submitted for processing are automatically assigned free processors in an order determined by a scheduler. When there is no free processor, the next script in order waits for one to become free.

Scheduled Scripts Using Queues

Scheduled script deployments created before the introduction of the SuiteCloud Processor may be using queues. A script deployment using a queue submitted for processing is sent to the SuiteCloud Processor according to the order determined by the scheduler - like a deployment without a queue. However, when determining the order, the scheduler must respect the queue for jobs that use it. A script job using a queue must not start before all script jobs submitted earlier using the same queue have completed.

You can use a **queue** argument to obtain the queue number assigned to a scheduled script. The *queue* argument is the second argument passed to the main executing function of a scheduled script. (The first argument is the *type* argument, which you can use regardless of whether you have a SuiteCloud Plus license. For details on the *type* argument, Executing a Scheduled Script in Certain Contexts.)

The *queue* argument provides the id of the queue selected in the Queue field of the scheduled script's Script Deployment page, if that deployment uses a queue assignment.

Important: The *queue* argument is an auto-generated argument passed by the system. You cannot set this as a parameter for a specific deployment like other function arguments.

If you have purchased one or more SuiteCloud Plus licenses, see the help topic SuiteCloud Plus Settings for valid *queue* argument values. This argument is not applicable when the script instance was created from a deployment that does not use queues.

Example

```
function processOrdersCreatedToday( type, queue )
{
 //only execute when run based on the schedule specified on the deployment record in the UI, and the script is in queue 5

  if ( type != 'scheduled' && type != 'skipped' && queue == 5 ) return ;

.... //process rest of script

}
```

Scheduled Script Samples

The following scheduled script code samples are provided in this section:

- Example 1 - Fulfill and Bill Sales Orders on a Daily Basis
- Example 2 - Resubmit a Script Depending on Units Remaining
- Example 3 - Create a Drip Marketing Campaign
- Example 4 - Automatically Send 'Thank You' Emails to Valued Customers
- Example 5 - Passing Script Parameters in a Scheduled Script

Example 1 - Fulfill and Bill Sales Orders on a Daily Basis

This script fulfills and bills all sales orders created today. This is a batch operation that would normally take a long time to execute, making it an ideal candidate for a scheduled script if using SuiteScript 1.0.

```
function processOrdersCreatedToday( type )
{
 //only execute when run from the scheduler
  if ( type != 'scheduled' && type != 'skipped' ) return;

   var filters = new Array();
   filters[0] = new nlobjSearchFilter( 'mainline', null, 'is', 'T' );
   filters[1] = new nlobjSearchFilter( 'trandate', null, 'equalTo', 'today' );

   var searchresults = nlapiSearchRecord( 'salesorder', null, filters, null, new nlobjSearchColumn('terms') );
   for ( var i = 0; searchresults != null && i < searchresults.length; i++ )
   {
      var id = searchresults[i].getId();
      var fulfillRecord = nlapiTransformRecord('salesorder', id, 'itemfulfillment');
      nlapiSubmitRecord( fulfillRecord );

      var billType = searchresults[i].getValue('paymentmethod') == null ? 'invoice' : 'cashsale';
      var billRecord = nlapiTransformRecord('salesorder', id, billType);
      nlapiSubmitRecord( billRecord );
   }
}
```

Example 2 - Resubmit a Script Depending on Units Remaining

Use nlapiScheduleScript, nlobjContext.getScriptId(), and nlobjContext.getDeploymentId() to create another instance of the currently executing scheduled script if there are more sales orders to update when the unit usage limit is reached.

```
function updateSalesOrders()
{
   var context = nlapiGetContext();
   var searchresults = nlapiSearchRecord('salesorder', 'customscript_orders_to_update')
   if ( searchresults == null )
      return;
   for ( var i = 0; i < searchresults.length; i++ )
   {
      nlapiSubmitField('salesorder', searchresults[i].getId(), 'custbody_approved', 'T')
      if ( context.getRemainingUsage() <= 0 && (i+1) < searchresults.length )
      {
         var status = nlapiScheduleScript(context.getScriptId(), context.getDeploymentId())
         if ( status == 'QUEUED' )
            break;
      }
   }
}
```

Example 3 - Create a Drip Marketing Campaign

This example illustrates a daily scheduled script for processing a drip marketing campaign with two touch points and one branch point. The basic workflow involves:

- Schedule Campaign for new leads (clone and schedule an existing campaign)
- Schedule follow-up Campaign for leads that are seven days old but whose statuses have not changed
- Schedule follow-up phone calls for sales reps assigned to these leads
- Schedule Campaign for leads that are seven days old whose statuses have since been upgraded

Parameters & Setup

- custscript_newleads campaign list parameter containing base campaign used for emailing new leads
- custscript_weekold campaign list parameter containing base campaign used for emailing week old unchanged leads

- custscript_converted campaign list parameter containing base campaign used for emailing week old upgraded leads
- custscript_leadstatus entitystatus list parameter containing the status used to define what a "new" lead is.

```
function processDripMarketing( type )
{
   if ( type != 'scheduled' ) return; /* script should only execute during scheduled calls. */

   /* process new leads */
   scheduleCampaign( custscript_newleads );
   /* process one-week old unchanged leads */
   scheduleCampaign( custscript_weekold );
   /* process follow-up for one-week old unchanged leads */
   scheduleFollowUpPhoneCall();
   /* process follow-up email for one-week old converted leads. */
   scheduleCampaign( custscript_converted );
}

function scheduleCampaign( base_campaign )
{
   var today = nlapiDateToString( new Date() );
   var campaign = nlapiCopyRecord('campaign', base_campaign);
   campaign.setFieldValue('startdate', today);
   campaign.setFieldValue('title',campaign.getFieldValue('title') + ' (' + today + ')');

   campaign.setLineItemValue('campaignemail','status',1,'EXECUTE');
   campaign.setLineItemValue('campaignemail','datescheduled',1,today);
   nlapiSubmitRecord( campaign );
}

function scheduleFollowUpPhoneCall()
{
   var filters = new Array();
   filters[0] = new nlobjSearchFilter('datecreated',null,'on','daysago7');
   filters[1] = new nlobjSearchFilter('status',null,'equalto', custscript_leadstatus);

   var columns = new Array();
   columns[0] = new nlobjSearchColumn('salesrep');
   columns[1] = new nlobjSearchColumn('phone');
   columns[2] = new nlobjSearchColumn('entityid');

   var today = nlapiDateToString( new Date() );
   var leads = nlapiSearchRecord('customer',null,filters,columns);
   for ( var i = 0; leads != null && i < leads.length; i++ )
   {
      var leadId = leads[i].getId();
      var salesrep = leads[i].getValue('salesrep');
      var phonenumber = leads[i].getValue('phone');
      var leadName = leads[i].getValue('entityid');
      /* Schedule Phone Call only if the lead is assigned and has a number. */
      if ( salesrep != null && phonenumber != null )
      {
         var call = nlapiCreateRecord('phonecall');
         call.setFieldValue('title','Follow up Call for '+leadName);
         call.setFieldValue('startdate', today );
         call.setFieldValue('assigned', salesrep );
         call.setFieldValue('phone', phonenumber );
         call.setFieldValue('company', leadId );
         call.setFieldValue('status','SCHEDULED');
         nlapiSubmitRecord( call );
      }
   }
}
```

Example 4 - Automatically Send 'Thank You' Emails to Valued Customers

This sample shows how to create a scheduled script to send thank you notes to valued, repeated customers. A scheduled script may be executed to perform daily searches for sales orders that are

placed today and within the last 30 days from the same customer. After retrieving the results, the scheduled script then sends these customers an email on behalf of the sales rep to thank them for their repeated business.

Note there are several nlobjContext.getRemainingUsage() API calls in the sample. This API provides the remaining SuiteScript usage to help scripts monitor how close they are to running into SuiteScript usage governance.

```
/**************************************************
 * This scheduled script looks for customers that
 * have placed multiple orders in the last 30 days.
 * It will send a thank you email to these customers
 * on behalf of their sales reps.
 */
function findHotCustomerScheduled(type)
{
  //Invoke only when it is scheduled
  if(type == 'scheduled')
  {
    //Obtaining the context object and logging the remaining usage available
    var context = nlapiGetContext();
    nlapiLogExecution('DEBUG', 'Remaining usage at script beginning', context.getRemainingUsage());

    //Setting up filters to search for sales orders
    //with trandate of today.
    var todaySOFilters = new Array();
    todaySOFilters[0] = new nlobjSearchFilter('trandate', null, 'on', 'today');

    //Setting up the columns.  Note the join entity.salesrep column.
    var todaySOColumns = new Array();
    todaySOColumns[0] = new nlobjSearchColumn('tranid', null, null);
    todaySOColumns[1] = new nlobjSearchColumn('entity', null, null);
    todaySOColumns[2] = new nlobjSearchColumn('salesrep', 'entity', null);

    //Search for the sales orders with trandate of today
    var todaySO = nlapiSearchRecord('salesorder', null, todaySOFilters, todaySOColumns);
    nlapiLogExecution('DEBUG', 'Remaining usage after searching sales orders from today', context.getRemainingUsage());

    //Looping through each result found
    for(var i = 0; todaySO != null && i < todaySO.length; i++)
    {
      //obtain a result
      var so = todaySO[i];

      //Setting up the filters for another sales order search
      //that are of the same customer and have trandate within
      //the last 30 days
      var oldSOFilters = new Array();
      var thirtyDaysAgo = nlapiAddDays(new Date(), -30);
      oldSOFilters[0] = new nlobjSearchFilter('trandate', null, 'onorafter', thirtyDaysAgo);
      oldSOFilters[1] = new nlobjSearchFilter('entity', null, 'is', so.getValue('entity'));
      oldSOFilters[2] = new nlobjSearchFilter('tranid', null, 'isnot', so.getValue('tranid'));

      //Search for for the repeated sales in the last 30 days
      var oldSO = nlapiSearchRecord('salesorder', null, oldSOFilters, null);
      nlapiLogExecution('DEBUG', 'Remaining usage after in for loop, i=' + i, context.getRemainingUsage());

      //If results are found, send a thank you email
      if(oldSO != null)
      {
        //Setting up the subject and body of the email
        var subject = 'Thank you!';
        var body = 'Dear ' + so.getText('entity') + ', thank you for your repeated business in the last 30 days.';

        //Sending the thank you email to the customer on behalf of the sales rep
        //Note the code to obtain the join column entity.salesrep
        nlapiSendEmail(so.getValue('salesrep', 'entity'), so.getValue('entity'), subject, body);
        nlapiLogExecution('DEBUG', 'Remaining usage after sending thank you email', context.getRemainingUsage());
      }
    }
  }
```

```
64 }
```

Example 5 - Passing Script Parameters in a Scheduled Script

The following sample shows how to retrieve passed parameters (by calling the getSetting(...) method on the nlobjContext object) within a scheduled script. It also shows how to execute a scheduled script by passing the custom ID (*scriptId*) of the Script record and the custom deployment ID (`deployId`) of the Script Deployment record. For details on working with script parameters, see the help topic Creating Script Parameters Overview.

```
//retrieve parameters inside a scheduled script
function scheduled_main()
{
//get script parameter values
var context = nlapiGetContext();
var strStartDate = context.getSetting('SCRIPT', 'custscriptstartdate');

var subsidiary = context.getSetting('SCRIPT', 'custscriptsubsidiary');
var startDate = new Date(strStartDate);

//schedule the script execution and define script parameter values
var startDate = new Date();
var params = {
    custscriptstartdate: startDate.toUTCString(),
    custscriptsubsidiary: 42
}

//so that the scheduled script API knows which script to run, set the custom ID
//specified on the Script record. Then set the custom ID on the Script Deployment
nlapiScheduleScript('customscript_audit_report', 'customdeploy_audit_report_dp', params);
}
```

Note: Since scheduled scripts also trigger user event scripts, developers may need to revisit the design of their user event scripts to ensure they will be invoked by the correct execution contexts.

Best Practice: Handling Server Restarts in Scheduled Scripts (SuiteScript 1.0)

Occasionally, a scheduled script failure may occur due to an application server restart. This could be due to a NetSuite update or maintenance, or an unexpected failure of the execution environment. Restarts can terminate an application forcefully at any moment. Therefore, robust scripts need to account for restarts and be able to recover from an unexpected interruption.

To prevent data issues that result from re-processing, the script should use idempotent operations. This means that any operation handled by the script can repeat itself with the same result (for example, prevent creating duplicate records). Or, the script must be able to recover (for example, by creating a new task to remove duplicates).

In SuiteScript 1.0, if there is an unexpected termination, the scheduled script restarts from either the beginning, the most recent yield, or most recent recovery point. It depends on the individual script. See the following topics for more information:

- Simple Scheduled Script
- Scheduled Script with Continuations and Recovery Points

Simple Scheduled Script

In this example, if there is a forceful termination in any part of the script, the script is always restarted from the beginning. It can find out whether it is the restarted execution by examining the `type` argument. The script is being restarted if and only if `(type == "aborted")`.

```
function scheduled(type)
{
    if (type == 'aborted')
    {
        // I might do something differently
    }
    .
    .
    .
}
```

Scheduled Script with Continuations and Recovery Points

In this example, the script contains nlapiYieldScript() and nlapiSetRecoveryPoint(). If the forceful termination occurs before the `nlapiYieldScript()` function is executed, the script is restarted from the beginning. It can find out whether it is the restarted execution by examining the type argument. The script is being restarted if and only if `(type === "aborted")`.

If the script is terminated after `nlapiYieldScript()` function is executed, and before the `nlapiSetRecoveryPoint()` function is executed, the script is restarted from the beginning of the continuation initiated by the `nlapiYieldScript()` function.

If the script is terminated after the `nlapiSetRecoveryPoint()` function is executed, the script is restarted from the recovery point set by the `nlapiSetRecoveryPoint()` function .

In both of these cases, the `type` argument value becomes obsolete. You need to examine the script context to find out whether it is the restarted execution. The script is being restarted if and only if `(nlapiGetContext().getTypeArgument() === "aborted")`.

Note that when a script performs a successful yield using `nlapiYieldScript()`, a new job is created that processes the continuation. The previous job that executed the script up to the yield is considered finished, and it is not restarted.

```
function scheduled(type)
{
    if (type == 'aborted')
    {
        // I might do something differently
    }
    .
    .
    .
    nlapiYieldScript();
    if (nlapiGetContext().getTypeArgument() == 'aborted')
    {
        // I might do something differently
    }
    .
    .
    .
    nlapiSetRecoveryPoint();
    if (nlapiGetContext().getTypeArgument() == 'aborted')
    {
        // I might do something differently
    }
    .
    .
    .
}
```

Portlet Scripts

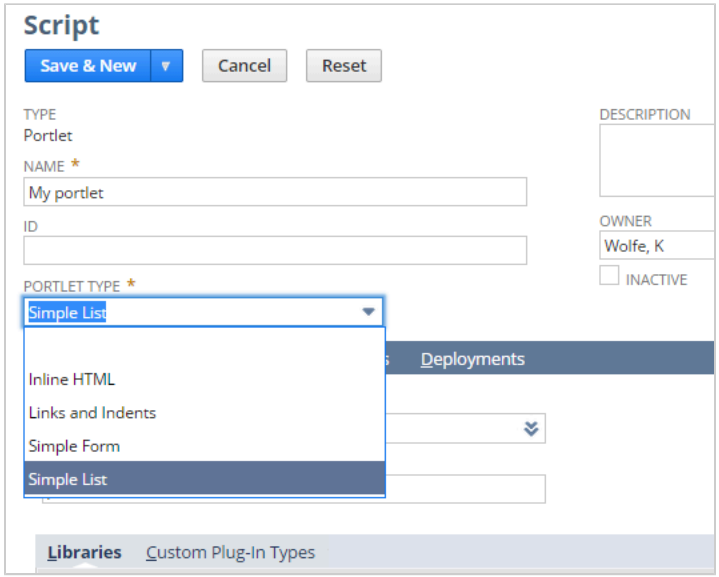
The following topics are covered in this section:

What Are Portlet Scripts?

Portlet scripts can be used to define and publish custom dashboard content. The following portlet types can be created:

- **LIST -** A standard list of user-defined column headers and rows (for example a Search Results portlets). See List Portlet for an example of a list portlet.
- **FORM -** A basic data entry form with up to one submit button embedded into a portlet (for example a Quickadd portlet). This type of portlet supports APIs to refresh and resize the portlet, as well as the use of record-level client-side script to implement validation. See Form-level and Record-level Client Scripts for details about this type of script. See Form Portlet for an example of a form portlet.
- **HTML** - An HTML-based portlet, the most flexible presentation format used to display free-form HTML (images, Flash, custom HTML). See HTML Portlet for an example of an HTML portlet.
- **LINKS** - This default portlet consists of rows of formatted content (for example an RSS portlet). See Links Portlet for an example of a links portlet.

Be aware that the portlet type (LIST, FORM, HTML, LINKS) is not passed as a value in the portlet script itself, rather it is defined on the portlet Script record page (see figure below). After you create your portlet.js file, you will load your .js file into the File Cabinet, create a new script record for your file (Customization > Scripts > New), and then select the portlet type from the Portlet Type drop-down menu.



You can use SuiteCloud Development Framework (SDF) to manage portlet scripts as part of file-based customization projects. For information about SDF, see the help topic SuiteCloud Development Framework Overview. You can use the Copy to Account feature to copy an individual portlet script to

another of your accounts. Each portlet script page has a clickable Copy to Account option in the upper right corner. For information about Copy to Account, see the help topic Copy to Account Overview.

Portlet Script Execution

Portlet scripts run on the server and are rendered in the NetSuite dashboard. A user-defined portlet function is executed whenever a SuiteScript-generated portlet is opened or refreshed by the user.

When writing portlet scripts, NetSuite automatically passes two arguments to your user-defined function. These arguments are:

- *portlet* - References a nlobjPortlet object
- *column* - Column index for this portlet on the dashboard (valid values are: **1** = left column, **2** = middle column, **3** = right column)

For custom portlets on Customer Dashboards, NetSuite can pass the following additional argument:

- *entity* - References the customer ID for the selected customer.

Example

```
function mySamplePortlet( portlet, column )
{
remainder of portlet script...
}
```

Note that *column* is an optional argument. If you choose not to pass a column value in your script, you can write:

```
function mySamplePortlet( portlet )
{
remainder of portlet script...
}
```

Portlet scripts can only run after users have added the scripts to their dashboards. After the scripts have been added, users must then open their dashboards for a portlet script to execute.

Note: To add portlet scripts to the dashboard, see Displaying Portlet Scripts on the Dashboard.

Assigning the Portlet Preference to a Script Parameter

Portlet script parameters (custom fields) can be configured to be customizable as portlet settings. This allows users to modify their script parameters for each portlet. For information about setting the *portlet* preference on script parameters, see the help topic Creating Script Parameters. If you are not familiar with the concept of script parameters, see the help topic Creating Script Parameters Overview.

Running a Portlet Script in NetSuite

To run a portlet script in NetSuite, you must:

1. Create a JavaScript file for your portlet script.
2. Load the file into NetSuite.
3. Create a Script record.
4. Define all runtime options on the Script Deployment page.

If you are new to SuiteScript and need information about each of these steps, see Running SuiteScript 1.0 in NetSuite Overview.

Important: Portlets scripts require that you reference the script from a custom portlet. See Displaying Portlet Scripts on the Dashboard for details.

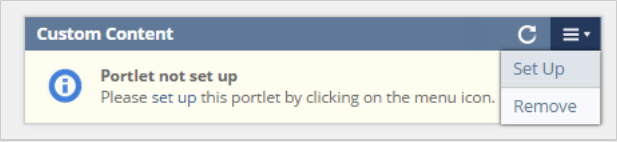
Displaying Portlet Scripts on the Dashboard

If you have created a portlet using SuiteScript, use these steps to display the custom portlet on your dashboard. Note that the following steps are to be completed only **after** you have performed all steps in the section Running a Portlet Script in NetSuite.

To display portlet scripts on the dashboard:

1. Go to your dashboard and click the **Personalize Dashboard** link.
2. Click one of the **Custom Portlet** links under the **Standard Content** folder.

   An empty Custom Content portlet appears on your dashboard.
3. Hover over the Portlet Setup arrow and click **Set Up**.



4. In the Set Up Scripted Content popup, select the desired portlet script from the **Source** drop-down list, and then click **Save.**

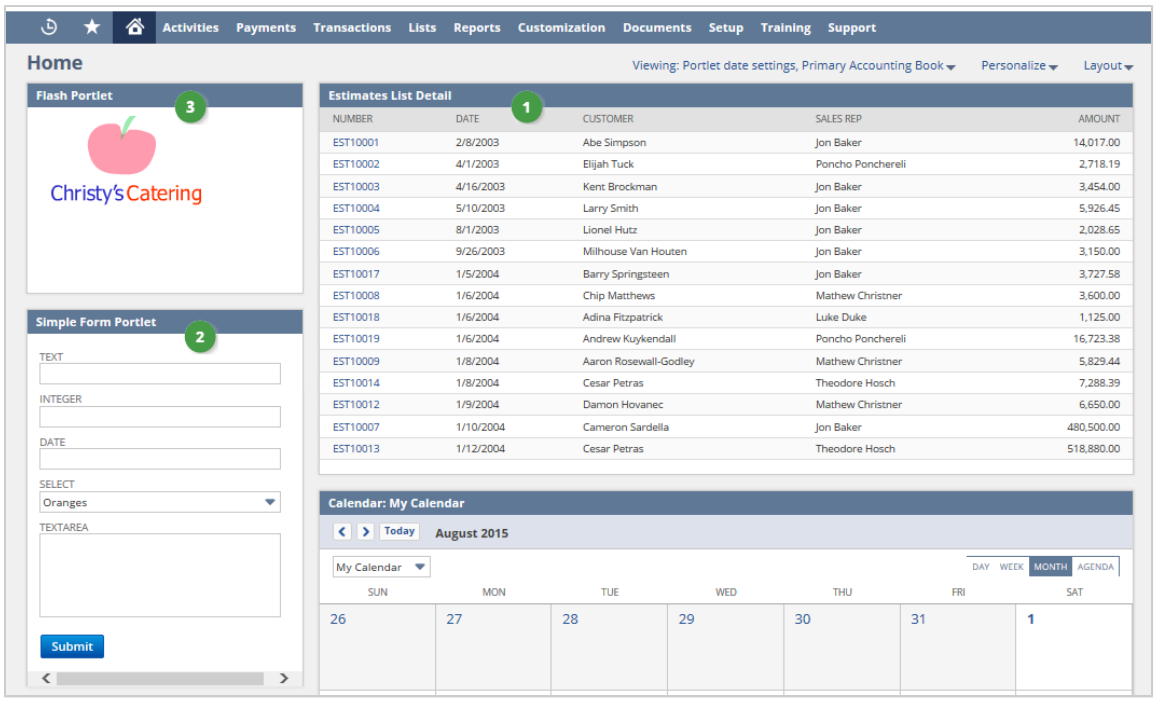
   The portlet will populate with data as defined in your portlet script.

Portlet Scripts Samples

The following sample portlets are provided:

- List Portlet
- Form Portlet
- HTML Portlet
- Links Portlet

The following image shows how the portlet samples appear in NetSuite:



Item	Description
1	Sample list portlet
2	Sample form portlet
3	Sample HTML portlet

List Portlet

This script searches for a list of estimates and displays the results in a list format. See the help topic nlobjPortlet for a list of portlet object methods.

Script:

```
function demoListPortlet(portlet, column)
{
    portlet.setTitle(column != 2 ? "Estimates List" : "Estimates List Detail")
    var col = portlet.addColumn('tranid','text', 'Number', 'LEFT');
    col.setURL(nlapiResolveURL('RECORD','estimate'));
    col.addParamToURL('id','id', true);
    portlet.addColumn('trandate','date', 'Date', 'LEFT');
    portlet.addColumn('entity_display','text', 'Customer', 'LEFT');
    if ( column == 2 )
    {
        portlet.addColumn('salesrep_display','text', 'Sales Rep', 'LEFT');
        portlet.addColumn('amount','currency', 'Amount', 'RIGHT');
    }
    var returncols = new Array();
    returncols[0] = new nlobjSearchColumn('trandate');
    returncols[1] = new nlobjSearchColumn('tranid');
    returncols[2] = new nlobjSearchColumn('entity');
    returncols[3] = new nlobjSearchColumn('salesrep');
    returncols[4] = new nlobjSearchColumn('amount');
    var results = nlapiSearchRecord('estimate', null, new
         nlobjSearchFilter('mainline',null,'is','T'), returncols);
```

```
    for ( var i = 0; i < Math.min((column != 2 ? 5 : 15 ),results.length); i++ )
        portlet.addRow( results[i] )
}
```

Form Portlet

This script builds a simple form in a portlet that POSTs data to a servlet. This form includes one embedded Submit button.

See the help topic nlobjPortlet for a list of portlet object methods.

Script:

```
function demoSimpleFormPortlet(portlet, column)
{
    portlet.setTitle('Simple Form Portlet')
    var fld = portlet.addField('text','text','Text');
    fld.setLayoutType('normal','startcol');
    portlet.addField('integer','integer','Integer');
    portlet.addField('date','date','Date');
    var select = portlet.addField('fruit','select','Select');
    select.addSelectOption('a','Oranges');
    select.addSelectOption('b','Apples');
    select.addSelectOption('c','Bananas');
    portlet.addField('textarea','textarea','Textarea');
    portlet.setSubmitButton(nlapiResolveURL('SUITELET','customscript_simpleformbackend', 'customdeploy_simpleform'),'Submit');
}
```

HTML Portlet

This portlet script generates the HTML required to download and display a FLASH animation in an HTML portlet. See the help topic nlobjPortlet for a list of portlet object methods.

Script:

```
function demoRichClientPortlet(portlet, column)
{
    portlet.setTitle('Flash Portlet')
    var content = "<table align=center border=0 cellpadding=3 cellspacing=0 width=100%><tr><td>"+
                  "<OBJECT CLASSID='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'>"+
                  "<PARAM NAME='MOVIE' VALUE='/images/flash/tomato.swf'>"+
                  "<embed src='/images/flash/tomato.swf'></embed></OBJECT></td>
                     </tr></table>";
    content = '<td><span>'+ content + '</span></td>'
    portlet.setHtml( content );
}
```

Note: To use the HTML from a text file uploaded to the File Cabinet, substitute the HTML string assigned to `var content` with nlapiLoadFile(id).getValue().

Links Portlet

This script makes an external request to slashdot.org to retrieve and display an RSS feed in a LINKS portlet. The APIs used are nlapiRequestURL(url, postdata, headers, callback, httpMethod) to fetch the RSS feed, nlapiStringToXML(text) to convert the feed into an XML document, and nlapiSelectNodes(node, xpath) / nlapiSelectValue(node, xpath) to query the XML document for the RSS data.

See the help topic nlobjPortlet for a list of portlet object methods.

Script:

```
function demoRssPortlet(portlet)
{
   portlet.setTitle('Custom RSS Feed');
   var feeds = getRssFeed();
   if ( feeds != null && feeds.length > 0 )
   {
      for ( var i=0; i < feeds.length ; i++ )
      {
         portlet.addLine('#'+(i+1)+': '+feeds[i].title, feeds[i].url, 0);
         portlet.addLine(feeds[i].description, null, 1);
      }
   }
}
function getRssFeed()
{
   var url = 'http://rss.slashdot.org/Slashdot/slashdot';
   var response = nlapiRequestURL( url, null, null );
   var responseXML = nlapiStringToXML( response.getBody() );
   var rawfeeds = nlapiSelectNodes( responseXML, "//item");
   var feeds = new Array();
   for (var i = 0; i < rawfeeds.length && i < 5 ; i++)
   {
     feeds[feeds.length++] = new rssfeed( nlapiSelectValue(rawfeeds[i], "title"),
                    nlapiSelectValue(rawfeeds[i], "link"),
                    nlapiSelectValue(rawfeeds[i], "description"));
   }
   return feeds;
}

function rssfeed(title, url, description)
{
   this.title = title;
   this.url = url;
   this.description = description;
}
```

Mass Update Scripts

The following topics are covered in this section:

- What Are Mass Update Scripts?
- Mass Update Script Execution
- Running a Mass Update Script in NetSuite
- Mass Update Scripts Samples

What Are Mass Update Scripts?

Mass update scripts allow you to programmatically perform custom mass updates to update fields that are not available through general mass updates. You can also use mass update scripts to run complex calculations, as defined in your script, across many records.

Note: If you are not familiar with mass update functionality in NetSuite, see the help topic Mass Changes or Updates in the NetSuite Help Center.

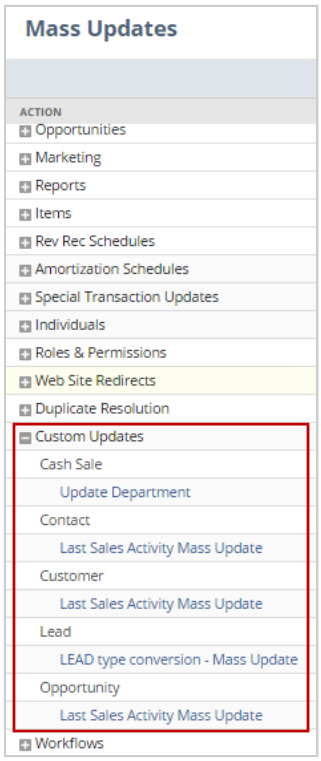
When a custom mass update is performed, the **record type** being updated is passed to a system-defined **rec_type** parameter in the mass update script. Additionally, the **internal ID** of each record in the custom mass update is passed to a system-defined **rec_id** parameter.

Whether you are using a custom mass update to update fields that are (or are not) available through inline editing, or you are updating fields based on the value of a SuiteScript script parameter, the executing function in your script will include the rec_type and rec_id parameters. For example:

```
function updateMemo( rec_type, rec_id )
{
nlapiSubmitField(rec_type, rec_id, 'memo', 'Premiere Customer', true);
}
```

Note: See Mass Update Scripts Samples for more details on working with mass update scripts.

Like all other script types, you must create a Script record and a Script Deployment record for mass update scripts. After you define the deployment for a mass update script and specify which record types the script will run against, the record type(s) will appear under the Custom Updates dropdown list, accessed by going to Lists > Mass Update > Mass Updates and select Custom Updates (see figure).



You can use SuiteCloud Development Framework (SDF) to manage mass update scripts as part of file-based customization projects. For information about SDF, see the help topic SuiteCloud Development Framework Overview. You can use the Copy to Account feature to copy an individual mass update script to another of your accounts. Each mass update script page has a clickable Copy to Account option in the upper right corner. For information about Copy to Account, see the help topic Copy to Account Overview.

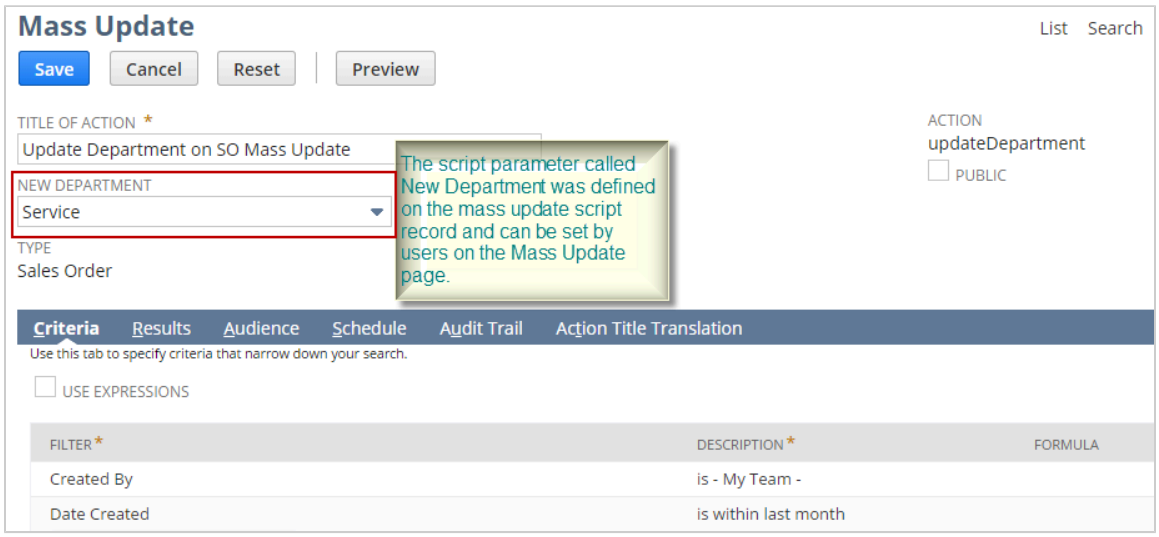
Mass Update Script Metering and Governance

The SuiteScript governance limit is 1000 units **per record** /invocation of a mass update script.

Also note that if multiple rows are returned for the same transaction, the custom mass update will still run only one time per transaction.

Creating Script Parameters for Mass Update Scripts

The mass update script type lets you create script parameters on the Script record page. Script parameters will then appear as fields in the header portion of the custom Mass Update page of the specified record type (see figure). Users executing custom mass updates can set the value(s) of one or more script parameters on the Mass Update page before running the update.



Note that your SuiteScript code must use the nlobjContext.getSetting(type, name) method to get the user-defined value of the script parameter. See Mass Update Scripts Samples for more details on working with script parameters within action scripts.

When you first create your script parameter you can set a parameter value as a **user** or **company** preference. The parameter will default to this value, but users may edit it as they run the mass update.

When you preview a custom mass update, the selected parameters will be shown for reference in the footer of the search results page.

Note: If you are not familiar with script parameters in SuiteScript, see the help topic Creating Script Parameters Overview in the NetSuite Help Center.

Mass Update Script Execution

Mass Update scripts execute on the server. They are not considered to be client scripts that run in the browser.

Mass Update scripts are executed when users click the Perform Update button on the Mass Update Preview Results page.

Important: Mass update scripts can only be invoked from the Mass Update page. They cannot be invoked from another script type. For example, you cannot invoke a mass update script by passing the script's scriptId and deployId to the nlapiScheduleScript(scriptId, deployId, params) function.

You have a choice of running mass update scripts as an administrator or as the logged-in user. As a script owner, you must have the Mass Update permission to test and work with mass update scripts. Users must have the Client SuiteScript and Server SuiteScript features enabled in their accounts for the scripts

to run. Also be aware that users who perform the custom mass update need the appropriate permission (Edit or Full) for the record types they are updating.

If a mass update script encounters an error, the script execution will abort. Only the updates that are completed prior to the error will be committed to the database.

Also note that the execution context for a mass update script is **custommassupdate**. This is important if you are trying to determine the execution context of a script using nlobjContext.getExecutionContext().

Finally, be aware that updates made to records during a custom mass update can trigger user event scripts if there are user event scripts associated with the records being updated.

Running a Mass Update Script in NetSuite

To run a mass update script in NetSuite, you must:

1. Create a JavaScript file for your action script.
2. Load the file into NetSuite.
3. Create a Script record.
4. Define all runtime options on the Script Deployment page.
5. After you define the deployment for a mass update script and specify which record types the script will run against, the record type(s) will appear under the Custom Updates dropdown list, accessed by going to Lists > Mass Update > Mass Updates > Custom Updates.

If you are new to SuiteScript and need information about steps 1–4, see Running SuiteScript 1.0 in NetSuite Overview.

Important: When running mass update scripts in NetSuite, be aware of the following:

- Mass update script deployments and mass updates can both be assigned an audience. It is the script owner's responsibility to ensure the two audiences are in sync. If the two audiences do not match, the mass update script will not run when users click the Perform Update button on the Mass Update page.
- When users run custom mass updates, they must have the appropriate permission (Edit/Full) for the record type(s) they are updating.
- Users must also have SuiteScript enabled in their accounts. (Administrators can go to Setup > Company > Enabled Features. In the SuiteCloud tab, click the Server SuiteScript box and the Client SuiteScript box.)

Mass Update Scripts Samples

The following mass update script samples are provided in this section:

- Updating a field that is available through inline edit
- Updating a field that is not available through inline edit
- Updating a field based on a script parameter value

Updating a field that is available through inline edit

The following sample mass update script shows that the Memo field on every record of a certain type will be updated in a custom mass update. The record type that this script will run against (Sales Order, for

example) is defined on the action Script Deployment page. When the custom mass update is executed by a user, the individual record IDs for **each** record of that type is passed to the mass update script's rec_id parameter.

Note that in the UI, the Memo field can be inline editing. In SuiteScript, fields that are inline editable are updated using the nlapiSubmitField(type, id, fields, values, doSourcing) function.

```
function updateMemo(rec_type, rec_id)
{
    nlapiSubmitField(rec_type, rec_id, 'memo', 'Premiere Customer', true);
}
```

In the sample above, when the user clicks the Perform Update button on the Mass Update Preview Results page, the Memo field on all specified sales orders will be updated to the text value **Premiere Customer**.

Important: Be aware that if the Memo field were not inline editable, you would have to load and submit each record in the custom mass update. The nlapiSubmitField function is used only on fields that can be inline edited through the UI. For example mass update scripts that update fields that are not available through inline edit, see Updating a field that is not available through inline edit.

Updating a field that is not available through inline edit

The second example updates the Probability field on the Opportunity record type. The record type is specified on the Script Deployment page for the action script.

After all custom mass update search criteria are defined on the Mass Update page for the Opportunity record type, this script will run on all Opportunity records that match the criteria. The Probability field will then be updated to the new value.

```
function updProbability(rec_type, rec_id)
{
var recOpportunity = nlapiLoadRecord(rec_type, rec_id);
recOpportunity.setFieldValue('probability', 61);
nlapiSubmitRecord(recOpportunity);
}
```

Note that in this sample, you are required to load and submit the entire record to change the value of the Probability field. You must do this when the field you want to change in the custom mass update cannot be inline edited. In other words, you cannot update the field using nlapiSubmitField(type, id, fields, values, doSourcing) without first loading the entire record object.

Updating a field based on a script parameter value

This sample mass update script updates the Department field on the Sales Order and Estimate record types. The update is based on the value of a script parameter called New Department (custscript_dept_update). Note that because the Department field is not accessible through inline editing, the only way to change the value of the Department field is to load and submit each record that the custom mass update is running against.

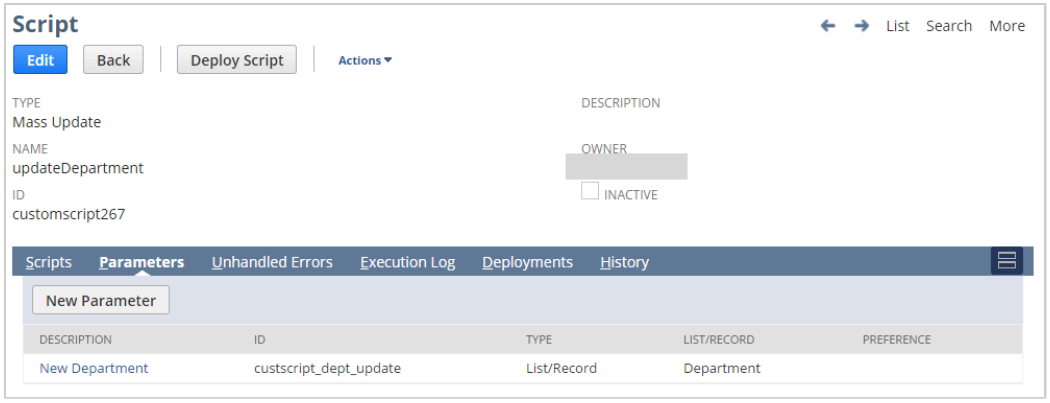
To perform a custom mass update that references a script parameter:

1. Create an action script (see below for an example). The script below will update the **Department** field on the record types specified in the action script's deployment. The update of the **Department** field will be based on the user-defined value specified in Step 6.

   Notice that the script's executing function takes the rec_type and rec_id parameters. When the custom mass update is performed, the record type defined on the deployment and the internal ID of each record will get passed to the executing function.

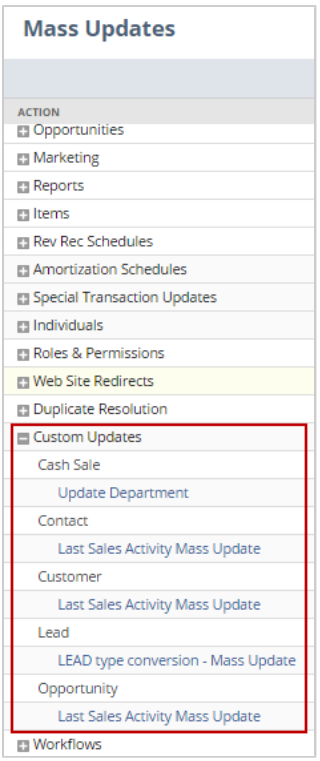
```
function updateDepartment(rec_type, rec_id)
{
  var transaction = nlapiLoadRecord(rec_type, rec_id);
  transaction.setFieldValue('department', nlapiGetContext().getSetting('SCRIPT',
          'custscript_dept_update'));
  nlapiSubmitRecord(transaction, false, true);
      }
```

2. Create a mass update Script record and define the new script parameter (see figure).

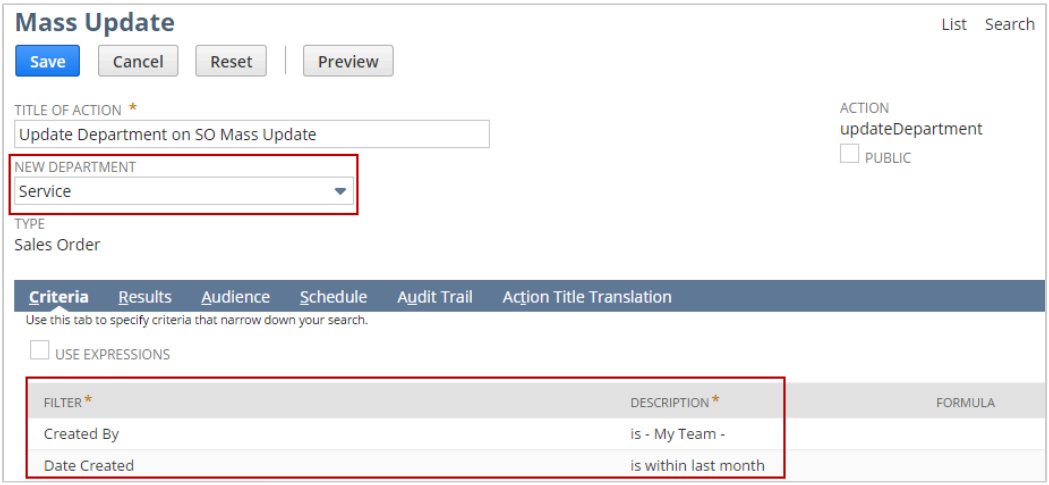


3. Create a script deployment (see figure below). In this example, the Update Department script will be deployed to Sales Order records.

4. After deploying the mass update script, go to Lists > Mass Update > Mass Updates. All custom mass updates referencing a mass update script will appear under the **Custom Updates** dropdown list. The following figure shows that the Update Department script has been deployed to the Estimate and Sales Order record types (see figure).



5. Click the custom mass update you want to execute. In this example, the **Update Department** link under the Sales Order deployment is selected.
6. On the Mass Update page for the record type, specify the value of the script parameter. In this example, the value of the **New Department** script parameter is set to **Services** (see figure).



7. Next, as with any mass update, use tabs on the Mass Update page to:
    1. Define which records the custom mass update will apply to (**Criteria** tab).
    2. Define which records you want to see when you preview the custom mass update (**Results** tab).
    3. Define the Audience that the custom mass update will apply to (**Audience** tab).

> **Important:** Be sure that the audience you define on the Mass Update page matches the audience defined on the Script Deployment page.

4. Set the frequency with which you want the custom mass update to run (**Schedule** tab).

8. Click **Preview** to verify which records the custom mass update will apply to.
9. Click **Perform Update** to run the custom mass update.

Workflow Action Scripts

Workflow Action Scripts are SuiteScript functions that let you create **custom** Workflow Actions that are defined on a record in a workflow. Workflow Action Scripts are useful for performing actions on sublists, as sublist fields are not currently available through the Workflow Manager. Workflow Action Scripts are also useful when you need to create custom actions that execute complex computational logic that is beyond what can be implemented with SuiteFlow's default actions.

The following topics are covered in this section:

- Creating Workflow Action Scripts
- Using Workflow Action Scripts

You can use SuiteCloud Development Framework (SDF) to manage workflow action scripts as part of file-based customization projects. For information about SDF, see the help topic SuiteCloud Development Framework Overview. You can use the Copy to Account feature to copy an individual workflow action script to another of your accounts. Each workflow action script page has a clickable Copy to Account option in the upper right corner. For information about Copy to Account, see the help topic Copy to Account Overview.

Creating Workflow Action Scripts

You define a Workflow Action Script like you would any other SuiteScript type: go to Customization > Scripting > Scripts > New (see figure).

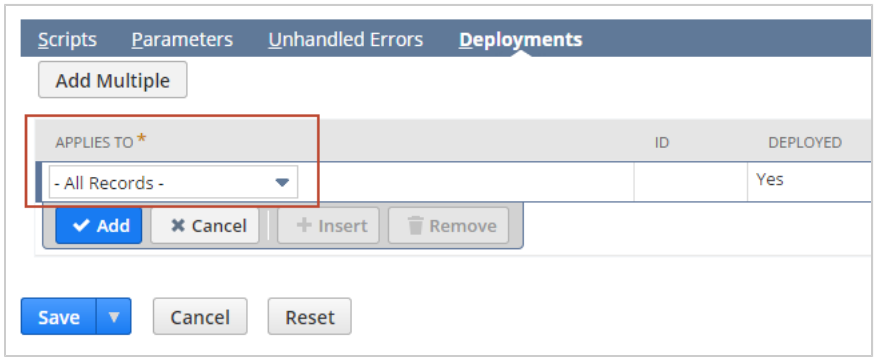
Select Type

TYPE	DESCRIPTION
Suitelet	Build interactive Web applications by scripting web requests
RESTlet	Build custom RESTful web services
User Event	Define business logic that is triggered when records are created, updated, viewed, or deleted
Scheduled	Schedule complex batch operations or queue them on-demand for execution
Client	Define business logic and perform client-side validation on your forms
Portlet	Publish scriptable portlets to your dashboards and centers
Mass Update	Perform an update to a record as part of a mass update
Workflow Action	Defines a custom action on a record that can be used as part of a workflow
Bundle Installation	Define scripts that run as part of bundle installation or update

You can create parameters for the action on the Parameters tab of the Script page (see figure below). On the Deployment tab (or on the Script Deployment record) you can define which record(s) you want the custom action to apply to.

Note that through Workflow Action Scripts you can create generic Custom Action that are available to all record types. Do this by selecting **All Records** in the Applies To dropdown list (see figure).

These custom actions then become available to all workflows, regardless of the underlying record type of the workflow. Through generic custom actions you can (for example) create a parameterized, generic action to set sales rep values. You can then set the parameters from within a workflow and invoke the generic "Set Sales Rep (Custom)" action, which will contain values specific to *that* workflow.



Be aware that if you set a Workflow Action Script to deploy to All Records, and then you try to specify another record type on the script's Script Deployment page, you will receive an error. Also note that if you set the deployment of a Workflow Action script to All Records, the script will appear in the palette of actions (labeled as **custom**) for all workflows.

You can create parameters for a workflow definition in the customization details in the workflow manager.

For additional details on working with Workflow Action scripts, see Using Workflow Action Scripts. For a sample that shows how to store a return value from a custom Action script in a custom workflow field, see the help topic Storing a Return Value from a Custom Action Script in a Workflow Field.

For general information about all SuiteScript types, see SuiteScript 1.0 Script Types Overview.

Using Workflow Action Scripts

Workflow Action Scripts are SuiteScript functions that can be invoked as Custom Action from workflow states.

Parameters

The SuiteScript function in a Workflow Action Script takes three parameters. All are optional. These parameters are described in the following table.

Parameter	Type	Description.
id	integer	The internal id of the workflow that calls the script.
form	nlobjForm	The form through which the script interacts with the record. This parameter is available only in the beforeLoad context.
type	string	An event type, such as create, edit, view, or delete.

The following snippet shows how you could use these parameters to customize the behavior of the script.

```
...
function workflowActionScript(id, type, form)
{
    if (type == 'edit') {
        form.addButton('custpage_testBtn','Test','');
```

```
    }
}
...
```

Note that you can also set the type of the value returned by a Workflow Action Script. You make this choice on the Parameters subtab on the script record. If there are fields of the same type defined in the workflow (or workflow state), you can configure this value to be saved as the value of a specified field.

Additional Usage Notes

The following is a custom action Workflow Action Script that sets the sales rep on the record in the workflow. This SuiteScript function can be executed by a Before Record Submit Trigger.

```
function changeSalesRep()
{
nlapiGetNewRecord().setFieldValue('salesrep', nlapiGetContext().getSetting('SCRIPT', 'custscript_salesrep'));
}
```

Notice there is no use of the record type and record id parameters (they are still sent, however); instead, nlapiGetNewRecord() is used to return all necessary data for the record currently in the workflow.

Also note that when executing Workflow Action Scripts, the current record context is **workflow**. See nlobjContext.getExecutionContext() for details on returning context information about what triggered the current script.

You can access parameters within a script using the following SuiteScript function:

```
nlapiGetContext().getSetting('SCRIPT','custscript_js_param1')
```

Bundle Installation Scripts

Note: SuiteBundler is still supported, but it will not be updated with any new features.

To take advantage of new features for packaging and distributing customizations, you can use the Copy to Account and SuiteCloud Development (SDF) features instead of SuiteBundler.

Copy to Account is an administrator tool that you can use to copy custom objects between your accounts. The tool can copy one object at a time, including dependencies and data. For more information, see the help topic Copy to Account Overview.

SuiteCloud Development Framework is a development framework that you can use to create SuiteApps from an integrated development environment (IDE) on your local computer. For more information, see the help topic SuiteCloud Development Framework Overview.

The following topics are covered in this section:

- What are Bundle Installation Scripts?
- Setting Up a Bundle Installation Script
- Sample Bundle Installation Script

What are Bundle Installation Scripts?

Note: SuiteBundler is still supported, but it will not be updated with any new features.

To take advantage of new features for packaging and distributing customizations, you can use the Copy to Account and SuiteCloud Development (SDF) features instead of SuiteBundler.

Copy to Account is an administrator tool that you can use to copy custom objects between your accounts. The tool can copy one object at a time, including dependencies and data. For more information, see the help topic Copy to Account Overview.

SuiteCloud Development Framework is a development framework that you can use to create SuiteApps from an integrated development environment (IDE) on your local computer. For more information, see the help topic SuiteCloud Development Framework Overview.

Warning: Before the Before Update script is run, the bundle update process checks for custom objects and files that are present in the previous bundle version but are no longer present in the new bundle version. Then, the bundle update process deletes such custom objects and files from the previous bundle version in a target account.

However, in case the Before Update script stops on any failed condition, the bundle update process stops the bundle update and the user is left with the previous bundle version, where the bundle update process has already deleted some custom objects and files, so the bundle may not be functioning correctly anymore.

You should not remove unused custom objects and files from the new bundle version. You should remove such objects from the new bundle only after your whole install base is upgraded to the new bundle version.

Bundle installation scripts are specialized server SuiteScripts that perform processing in target accounts as part of bundle installation, update, or uninstall. This processing can include setup, configuration, and data management tasks that would otherwise have to be completed by account administrators. These scripts enhance solution providers' ability to manage the bundle deployment process.

Every bundle can include a bundle installation script that is automatically run when the bundle is installed, upgraded, or uninstalled. Each bundle installation script can contain triggers to be executed before install, after install, before update, after update, and after uninstall. All triggers should be included in a single script file. This trigger code can ensure that bundles are implemented correctly, and can prevent bundle installation, update, or uninstall if proper setup has not occurred.

Bundle installation scripts have no audience because they are always executed using the administrator role, in the context of bundle installation, update, or uninstall. Bundle installation scripts do not have event types.

A bundle installation script can be associated with multiple bundles. Before a script can be associated with a bundle, it must have a script record and at least one deployment. A bundle creator associates a bundle installation script with a bundle by selecting one of its deployments in the Bundle Builder. The script .js file and script record are automatically included in the bundle when it is added to target accounts. Script file contents can be hidden from target accounts based on an option set for the .js file in the File Cabinet record.

You can use SuiteCloud Development Framework (SDF) to manage bundle installation scripts as part of file-based customization projects. For information about SDF, see the help topic SuiteCloud Development Framework Overview. You can use the Copy to Account feature to copy an individual bundle installation script to another of your accounts. Each bundle installation script page has a clickable Copy to Account option in the upper right corner. For information about Copy to Account, see the help topic Copy to Account Overview.

Bundle Installation Script Functions

Triggered Functions

A bundle installation script's functions are executed automatically during bundle installation, update, or uninstall, based on one or more of the following triggers:

- Before Install - Executed before a bundle is installed for the first time in a target account.
- After Install - Executed after a bundle is installed for the first time in a target account.
- Before Update - Executed before a bundle in a target account is updated.
- After Update - Executed after a bundle in a target account is updated.
- Before Uninstall - Executed before a bundle is uninstalled from a target account.

A bundle installation script file should include a function for at least one of these triggers. If you are using more than one of these, they should all be in the same script file.

The following are example uses for bundle installation script triggered functions:

- Before Install: Check the existing configuration and setup in the target account prior to bundle installation, and halt the installation with an error message if the target account does not meet minimum requirements to run the solution.
- After Install: Automate the setup and configuration of the bundled application after it has been installed in the target account, eliminating manual tasks.
- After Install or After Update: Connect to an external system to fetch some data and complete the setup of the bundled application.
- Before Update: Manage required data changes in the target account prior to executing an upgrade.
- Before Uninstall: Reset configuration settings or remove data associated with the bundle being uninstalled.

Function Parameters

Two specialized parameters are available to functions in bundle installation scripts, to return the version of bundles, as specified on the Bundle Basics page of the Bundle Builder.

- The **toversion** parameter returns the version of the bundle that will be installed in the target account. This parameter is available to Before Install, After Install, Before Update, and After Update functions.
- The **fromversion** parameter returns the version of the bundle that is currently installed in the target account. This parameter is available to Before Update and After Update functions.

Getting Bundle Context

Calls to Functions in Other Script Files

A bundle installation script file can include calls to functions in other script files, if the files are added as library script files on the script record. Any .js files for library script files are automatically included in the bundle when it is added to target accounts.

Bundle installation scripts can call scheduled scripts, but only in the After Install and After Update functions. Calls to scheduled scripts are not supported in the Before Install, Before Update, and Before Uninstall functions.

Bundle Installation Script Governance

Bundle installation scripts are governed by a maximum of 10,000 units per execution.

Defining Deployments for Bundle Installation Scripts

You can create multiple deployments for each bundle installation script, with different parameters for each, but only one deployment can be associated with each bundle. When you associate a bundle installation script with a bundle, you select a specific script deployment.

Bundle installation scripts need to be executed with administrator privileges, so the Execute as Role field should always be set to Administrator on the script deployment record.

Bundle installation scripts can only be run in target accounts if the Status is set to Released. The Status should be set to Testing if you want to debug the script.

Bundle Installation Script Error Handling

Any bundle installation script failure terminates bundle installation, update, or uninstall.

Bundle installation scripts can include their own error handling, in addition to errors thrown by SuiteBundler and the SuiteScript engine. An error thrown by a bundle installation script returns an error code of "Installation Error", followed by the text defined by the script author.

Setting Up a Bundle Installation Script

Note: SuiteBundler is still supported, but it will not be updated with any new features.

To take advantage of new features for packaging and distributing customizations, you can use the Copy to Account and SuiteCloud Development (SDF) features instead of SuiteBundler.

Copy to Account is an administrator tool that you can use to copy custom objects between your accounts. The tool can copy one object at a time, including dependencies and data. For more information, see the help topic Copy to Account Overview.

SuiteCloud Development Framework is a development framework that you can use to create SuiteApps from an integrated development environment (IDE) on your local computer. For more information, see the help topic SuiteCloud Development Framework Overview.

Complete the following tasks to set up a bundle installation script:

- Create the Bundle Installation Script File
- Add the Bundle Installation Script File to the File Cabinet
- Create the Bundle Installation Script Record

- Define Bundle Installation Script Deployment
- Associate the Script with a Bundle

A bundle installation script is a specialized server SuiteScript that is executed automatically in target accounts when a bundle is installed, updated, or uninstalled. For details about how to create a bundle, see the help topic Creating a Bundle with the Bundle Builder.

Create the Bundle Installation Script File

You can create a bundle installation script file in the same manner that you create other types of SuiteScript files, as described in Step 1: Create Your Script.

Bundle installation scripts support the entire SuiteScript API, including error handling and the debugger. For details specific to bundle installation scripts, see Bundle Installation Script Functions.

To create a bundle installation script file:

1. Create a .js script file and add code.

   This single script file should include Before Install, After Install, Before Update, After Update, and Before Uninstall functions as necessary. It can include calls to functions in other files, but you will need to list these files as library script files on the NetSuite script record.

Add the Bundle Installation Script File to the File Cabinet

After you have created a .js file with your bundle installation script code, you need to add this file to the NetSuite File Cabinet.

The following steps describe how to add the file manually. If you are using the SuiteCloud IDE, this process is automated. For more information, see Step 2: Add Script to NetSuite File Cabinet.

To add a bundle installation script file to the File Cabinet:

1. Go to Documents > Files > File Cabinet, and select the folder where you want to add the file.

   You should add your file to the SuiteScripts folder, but it can be added to any other folder of your choice.
2. Click **Add File**, and browse to the .js file.
3. In the File Cabinet folder where you added the bundle installation script file, click the **Edit** link next to file.
4. Check the **Available for SuiteBundles** box.
5. Optionally, you can check the **Hide in SuiteBundle** box.

   Because this script file will be included in the bundle, by default its contents will be accessible to users of target accounts where the bundle is installed. If you do not want these users to see this file, you can set this option to hide it.
6. Click **Save**.

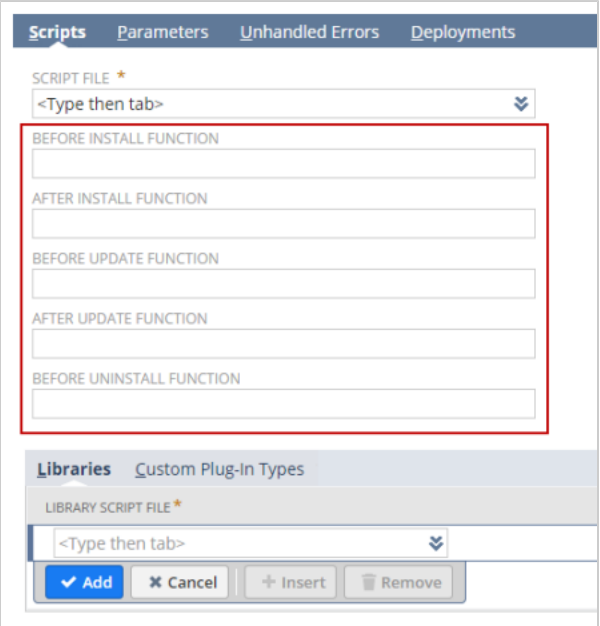
Create the Bundle Installation Script Record

After you have added a bundle installation script file to the File Cabinet, you can create a NetSuite script record.

To create a bundle installation script record:

1. Go to Setup > Customization > Scripts > New, and click **Bundle Installation**.
2. Complete fields in the script record and save.

   Although you do not need to set every field on the Script record, at a minimum you must provide a **Name** for the Script record, load your SuiteScript file to the record, and specify at least one of the executing functions on the Scripts tab.



   These functions should all be in the main script file. If these functions call functions in other script files, you need to list those files as library script files.

   For more details about creating a script record, see Steps for Creating a Script Record.

Define Bundle Installation Script Deployment

After you have created a bundle installation script record, you need to define at least one deployment. For details about defining script deployments, see Step 5: Define Script Deployment and Steps for Defining a Script Deployment

You can define multiple deployments per bundle installation script . When you associate the script with a bundle, you choose a specific deployment.

To define a bundle installation script deployment.

1. Do one of the following:
   - When you save your Script record, you can immediately create a Script Deployment record by selecting **Save and Deploy** from the Script record **Save** button.
   - If you clicked **Save**, immediately afterwards you can click **Deploy Script** on the script record.
   - If you want to update a deployment that already exists, go to Customization > Scripting > Script Deployments. Click **Edit** next to the deployment record you want to edit.
2. Complete fields in the script deployment record and click **Save**.

   Be sure to check the **Execute as Admin** box.

If you want to debug the script, set the **Status** to **Testing**. To enable the script to be run in a target account, you must set the **Status** to **Released**.

Associate the Script with a Bundle

After a bundle installation script has been created and at least one deployment has been defined for it, you can associate the script with bundles as desired.

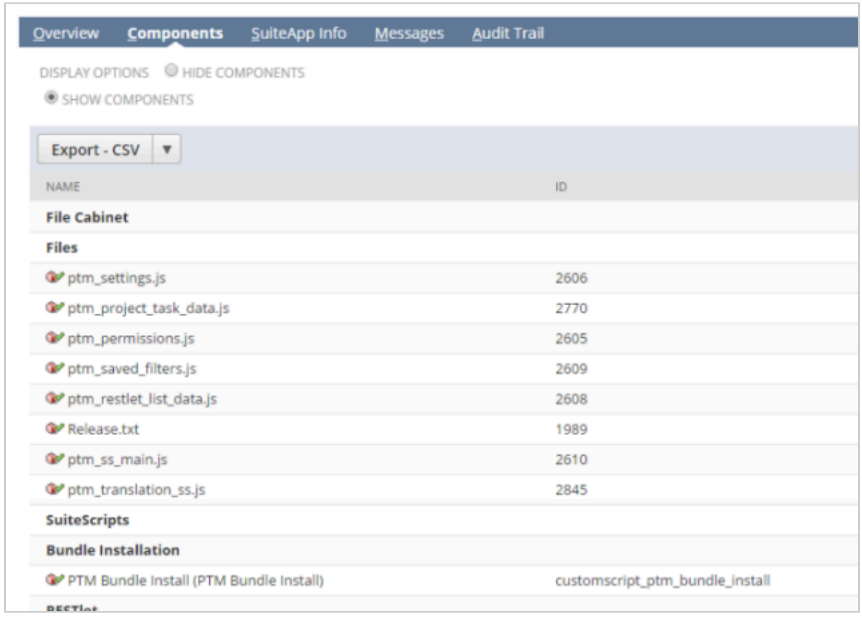
Note: The SuiteBundler feature must be enabled in your account for you to have access to the Bundle Builder where this task is completed.

When you associate a script with a bundle, you select a specific script deployment.

To associate a script with a bundle:

1. Start the Bundle Builder.
   - If you are creating a new bundle, go to Customization > SuiteBuilder > Create Bundle
   - If you are editing an existing bundle, go to Customization > SuiteBundler > Search & Install Bundles > List, and select **Edit** from the Action menu for the desired bundle.
2. On the Bundle Basics page, select a bundle installation script deployment from the **Installation Script** dropdown list.
3. Proceed through the remaining Bundle Builder steps, making definitions as necessary, and click **Save**. Note the following:
   - On the Select Objects page of the Bundle Builder, you do not have to explicitly add the bundle installation script. This script record and the related .js file are included automatically in the bundle, as are any other .js files that are listed as library script files on the script record.
   - For detailed instructions to complete all Bundle Builder steps, see the help topic Creating a Bundle with the Bundle Builder.

After the bundle has been saved, this script record and related file(s) are listed as Bundle Components on the Bundle Details page.



Sample Bundle Installation Script

Note: SuiteBundler is still supported, but it will not be updated with any new features.

To take advantage of new features for packaging and distributing customizations, you can use the Copy to Account and SuiteCloud Development (SDF) features instead of SuiteBundler.

Copy to Account is an administrator tool that you can use to copy custom objects between your accounts. The tool can copy one object at a time, including dependencies and data. For more information, see the help topic Copy to Account Overview.

SuiteCloud Development Framework is a development framework that you can use to create SuiteApps from an integrated development environment (IDE) on your local computer. For more information, see the help topic SuiteCloud Development Framework Overview.

This sample includes a bundle installation script file and a library script file. For details, see the following:

Summary of Sample Script Files

The bundle installation script file includes the following:

- Function that executes before bundle installation, ensuring that the Work Orders feature is enabled in the target NetSuite account, and if the bundle that is being installed is version 2.0, also ensuring that the Multiple Currencies feature is enabled
- Function that executes after bundle installation, creating an account record in the target account (note that accounts are not available to be included in bundles)
- Function that executes before bundle update, ensuring that the Work Orders feature is enabled in the target NetSuite account, and if the target account bundle is being updated to version 2.0, also ensuring that the Multiple Currencies feature is enabled
- Function that executes after bundle update, creating an account record in the target account if the update changed the bundle version number

The library script file includes a function that is called by the bundle installation script functions executed before installation and before update.

- This function checks whether a specified feature is enabled in the target account and returns an error if the feature is not enabled.
- When an error is returned, bundle installation or update terminates.

Sample Bundle Installation Script File Code

The bundle installation script file **SampBundInst.js** contains the following code.

```
function beforeInstall(toversion)
```

```
{
	// Always check that Workorders is enabled
	checkFeatureEnabled('WORKORDERS');

	// Check that Multi Currency is enabled if version 2.0 is being installed
	if ( toversion.toString() == "2.0" )
		checkFeatureEnabled('MULTICURRENCY');
}

function afterInstall(toversion)
{
	// Create an account record
	var randomnumber=Math.floor(Math.random()*10000);
	var objRecord = nlapiCreateRecord('account');
	objRecord.setFieldValue('accttype','Bank');
	objRecord.setFieldValue('acctnumber',randomnumber);
	objRecord.setFieldValue('acctname','Acct '+toversion);
	nlapiSubmitRecord(objRecord, true);
}

function beforeUpdate(fromversion, toversion)
{
	// Always check that Workorders is enabled
	checkFeatureEnabled('WORKORDERS');
	// Check that Multi Currency is enabled if version 2.0 is being installed
	if ( toversion.toString() == "2.0" )
		checkFeatureEnabled('MULTICURRENCY');
}

function afterUpdate(fromversion, toversion)

{
	// Do not create an account if updating with the same version as the one installed
	if (fromversion.toString() != toversion.toString())
	{
		// Create an account record
		var randomnumber=Math.floor(Math.random()*10000);
		var objRecord = nlapiCreateRecord('account');
		objRecord.setFieldValue('accttype','Bank');
		objRecord.setFieldValue('acctnumber',randomnumber);
		objRecord.setFieldValue('acctname','Acct '+toversion);
		nlapiSubmitRecord(objRecord, true);
	}

}
```

Sample Library Script File Code

The library script file **CheckFeat.js** contains the following code.

```
function checkFeatureEnabled(featureId)
{
	nlapiLogExecution('DEBUG','Checking Feature',featureId);
	var objContext = nlapiGetContext();
	var feature = objContext.getFeature(featureId);

	if ( feature )
	{
		nlapiLogExecution('DEBUG','Feature',featureId+' enabled');
	}

	else
	{
		throw new nlobjError('INSTALLATION_ERROR','Feature '+featureId+' must be enabled. Please enable the feature and re-try.');
	}

}
```

Running SuiteScript 1.0 in NetSuite Overview

Running a script in NetSuite includes these basic steps:

- Step 1: Create Your Script
- Step 2: Add Script to NetSuite File Cabinet
- Step 3: Attach Script to Form
- Step 4: Create Script Record
- Step 5: Define Script Deployment

Important: Step 3 is for form-level client scripts **only**. If you are creating a user event, scheduled, portlet, Suitelet, or **record** -level client script, skip Step 3, and perform steps 4 and 5.

Step 5 provides the basic steps required for deploying a script into NetSuite. To learn how to specify additional deployment options, see the help topic Setting Runtime Options.

Step 1: Create Your Script

All SuiteScript files must end with a JavaScript (.js) file extension. Although you can use any text editor (including Notepad) to write your SuiteScript .js files, you should use the SuiteCloud IDE. If you have not installed SuiteCloud IDE, see the help topic Setting Up SuiteCloud IDE Plug-in for Eclipse in the NetSuite Help Center.

Depending on what you are trying to do in NetSuite, the code in your .js file can be as basic as a **client** script that never even touches a NetSuite server. It runs purely client-side in the browser and alerts users after they have loaded a specific NetSuite record, for example:

```
function pageInitAlertUser()
{
   alert ('You have loaded a record');
}
```

Alternatively, your script can be can be as complex as executing a NetSuite search, getting the results, and then transforming the results into a PDF document. See the samples for nlapiXMLToPDF(xmlstring) as an example.

The APIs you use in your code and the logic you write will depend on what you're trying to accomplish in NetSuite. See the help topic What You Can Do with the SuiteScript API if you are unsure of what you can do using the SuiteScript API.

After you have created your .js file, see Step 2: Add Script to NetSuite File Cabinet.

Note: To see which APIs are included in the SuiteScript API, start with SuiteScript 1.0 API Overview.

Step 2: Add Script to NetSuite File Cabinet

If you are writing your script files in SuiteCloud IDE, loading a file into the NetSuite File Cabinet is done by right-clicking on your file in SuiteCloud IDE and selecting **NetSuite** > **Upload Selected File(s)**. For more information, see the help topic Uploading a SuiteScript File in SuiteCloud IDE Plug-in for Eclipse.

If you have written your .js files in anything other than SuiteCloud IDE, you will need to manually upload your files into NetSuite. See the help topic Uploading SuiteScript into the File Cabinet Without SuiteCloud IDEs for details.

Note: The **SuiteScripts** folder in the File Cabinet is provided for convenience, however, you can store your script files in any location.

After your script has been added to the NetSuite File Cabinet, see **either** :

- Step 3: Attach Script to Form (if you want to run a **form-level client script** in NetSuite)
- Step 4: Create Script Record (if you want to run any other script type. For example, if you want to run a user event, scheduled, portlet, Suitelet, action, or record-level client script, proceed to Step 4.)

Step 3: Attach Script to Form

Form-level client scripts are "attached" to the forms they run against. Be aware that in NetSuite, there are two different types of client SuiteScript. The information in this section pertains **ONLY** to **form-level** client scripts.

Important: For the differences between form- and record-level client scripts, see Form-level and Record-level Client Scripts.

To attach a form-level client script to a custom form:

1. Ensure that your client script has been uploaded to the File Cabinet. (See Step 2: Add Script to NetSuite File Cabinet.)
2. Go to the appropriate custom form in NetSuite.

   Form-level client scripts can only be attached to custom entry forms, custom transaction forms, and custom online forms. Click Customization > Forms >*[Form]*.
3. Click **Edit** next to the desired custom form, or click **Customize** next to an existing standard form to create a new custom form that is based on the standard version.

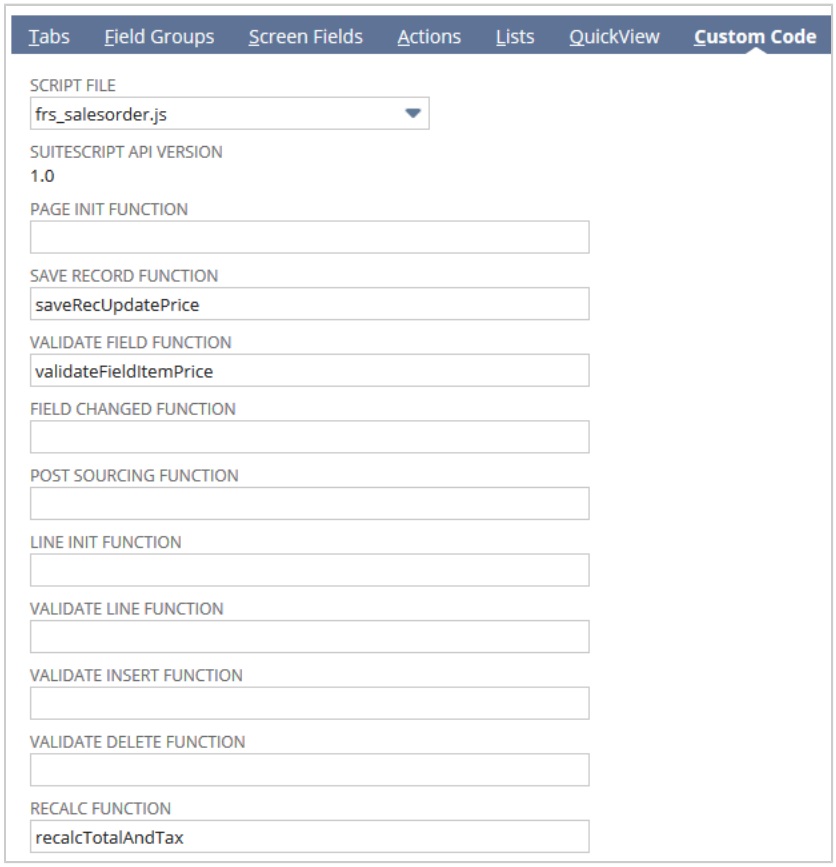
   Note: For more information about creating custom entry, transaction, and online forms, refer to the *SuiteBuilder* (*Customization) Guide*.
4. On the form's **Custom Code** subtab, use the **Script File** dropdown list to select your SuiteScript version 1.0 file.

   The page updates and populates the SuiteScript API Version field. The system also adds several text fields to the page.
5. Use the text fields to enter the functions defined in your script that should be called for each appropriate client event. If you are unsure of which actions trigger each client event, see Client Event Types. To learn how many functions you can execute on one form, see How Many Client Events Can I Execute on One Form?

   The following figure shows a custom sales order form. This form is set as the preferred form for sales orders. What this means is that all customizations made to this form, and any client script file attached to the form, run whenever a NetSuite user navigates to and loads the sales order record. As shown in the screenshot, when a sales order loads, three functions execute:

   - The savRecUpdatePrice function will execute when the record is saved.
   - The valFieldItemPrice function will execute when a particular field on the sales order is changed.
   - The recalcTotalAndTax function will execute when a line item as been added to a sublist.

> **Important:** You must enter function names **exactly** as they appear in your script. However, do not include parenthesis or parameter values when you enter the function name on the custom form.



6. If appropriate, use the **Library Script File** dropdown list to select a library file to associate with the form. The library script file should contain any commonly used functions. The file named in the **Script File** field should contain functions specific to the current form.

After you have attached your form-level client script to a form, your script will execute whenever the triggering action occurs. For a list of possible client event triggers, see Client Event Types.

If you have created a form-level client script, you do not have to complete the procedures described in Step 4: Create Script Record or Step 5: Define Script Deployment.

Step 4: Create Script Record

After writing your SuiteScript .js file and uploading the file to the File Cabinet, you must then create a **Script** record for the file (see figure).

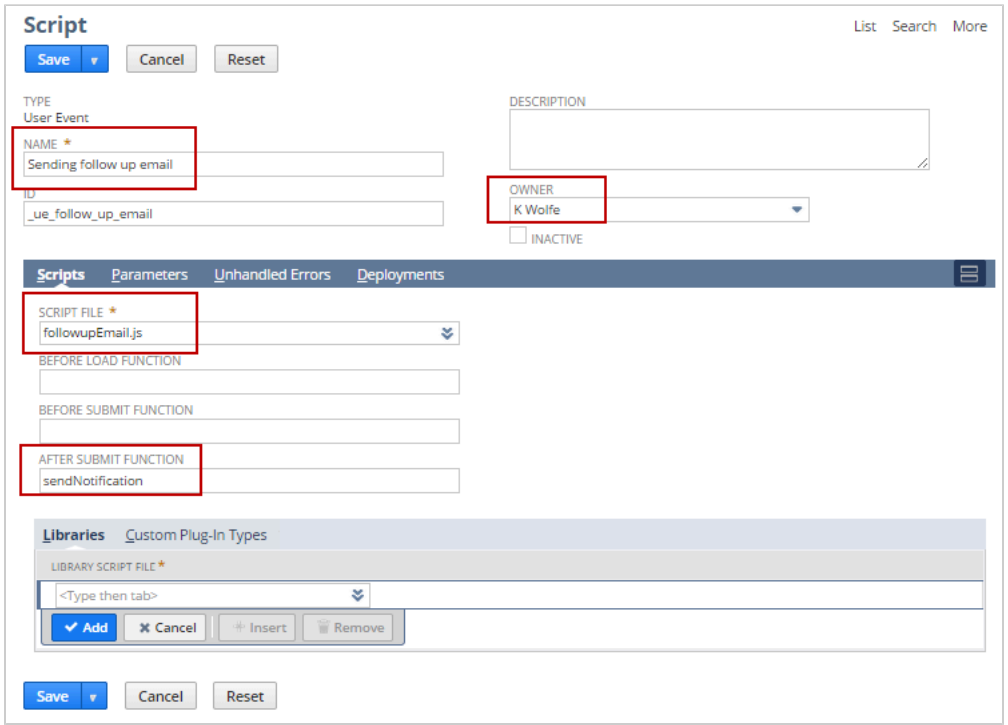
On the **Script** record you will:

- Add your SuiteScript .js file.
- Define the script owner.
- If applicable, add one or more library files.
- Define the function(s) from your SuiteScript file you want executed.
- If applicable, create script parameters (custom fields) that are unique to the Script record.
- Specify who should be contacted if an error is thrown in your script.

Although you do not need to set every field on the Script record, **at a minimum** you must set the following (see figure):

1. Provide a name for the Script record.
2. Specify the script owner.
3. Load your SuiteScript .js file.
4. Specify the main executing function within the file.

Important: See Steps for Creating a Script Record for more detailed information.



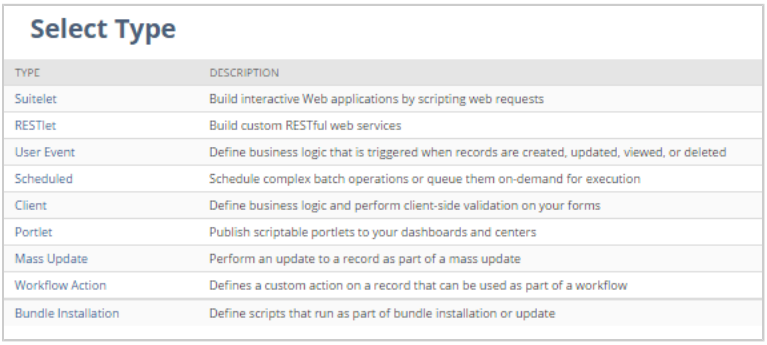
Steps for Creating a Script Record

The following steps provide details for creating a Script record. For an overview that explains the purpose of the Script record, be sure to see Step 4: Create Script Record.

To create a script record:

1. Go to Customization > Scripting > Scripts > New .

   Note that after creating your script record, you can later access the record by going to Customization > Scripting > Scripts to see a list view of all Script records.
2. In the **Script File** field, enter a name for the script record and then click **Create Script Record**.
3. Select the script type (see figure).



> **Note:** The Client scripts listed here are record-level client script. These scripts run in addition to any form-level client scripts that might have already been attached to an existing form. For information about the differences between form- and record-level client scripts, see Form-level and Record-level Client Scripts .

4. In the Script record **Name** field, enter a name for the script record.

   You can have multiple deployments of the same SuiteScript file. Therefore, be sure that the name of the Script record is generic enough to be relevant for all deployments.

   For example, you may want your SuiteScript (.js) file to execute whenever Vendor records are saved. You might also want this script to execute whenever Customer records are saved. You will need to define two different deployments for the script. However, both deployments will reference the same script / Script record. (Information about defining script deployments is covered in Step 5: Define Script Deployment.)
5. In the **ID** field, if desired, enter a custom ID for the script record. If the **ID** field is left blank, a system-generated internal ID is created for you.

   For information about whether you should create your own custom ID, see Creating a Custom Script Record ID.
6. In the **Description** field, if desired, enter a description for the script.
7. In the **Owner** field, select a script owner.

   By default the owner is set to the currently logged-in user. After the Script record is saved, only the owner of the record or a system administrator can modify the record.
8. (Optional) Select the Inactive box if you do not want to deploy the script. When a script is set to Inactive, all of the deployments associated with the script are also inactive. If you want to inactivate a specific deployment rather than all deployments of this scripts, go to the Script Deployments page.
9. On the **Scripts** tab, set the following:
   1. In the **Script File** field, select the SuiteScript .js file to associate with the current script record.

      If you have uploaded your script into the NetSuite File Cabinet, your script appears in the **Script File** dropdown list. For directions on uploading scripts into the File Cabinet, see either of the following sections:

- Uploading a SuiteScript File in SuiteCloud IDE Plug-in for Eclipse - If you are uploading scripts using the SuiteCloud IDE.
- Uploading SuiteScript into the File Cabinet Without SuiteCloud IDEs - If you are not uploading your scripts using SuiteCloud IDE.

> **Note:** If you are maintaining your SuiteScript files outside of the File Cabinet, click the **+** button next to the **Script File** dropdown list. In the popup window that appears, browse for your .js file.

2. (Optional) In the **Library Script File** field, select the library files you want to associate with the Script record.

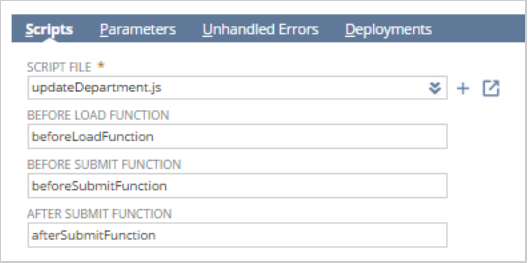
   A library script file should contain any commonly used functions, whereas the SuiteScript file should contain functions specific to the current Script record. Note that multiple library files can be added to a script record.

   The upload order of your library files only matters if you have two functions with the same name. When there is a function name conflict in a library file, the script will execute the function in the library file that was loaded last.

3. In the **Function** field(s), type the name of the function(s) you want executed in the .js file. Do not include the function parentheses or any parameters. For example, type **myFunction** rather than myFunction(param1, param2).

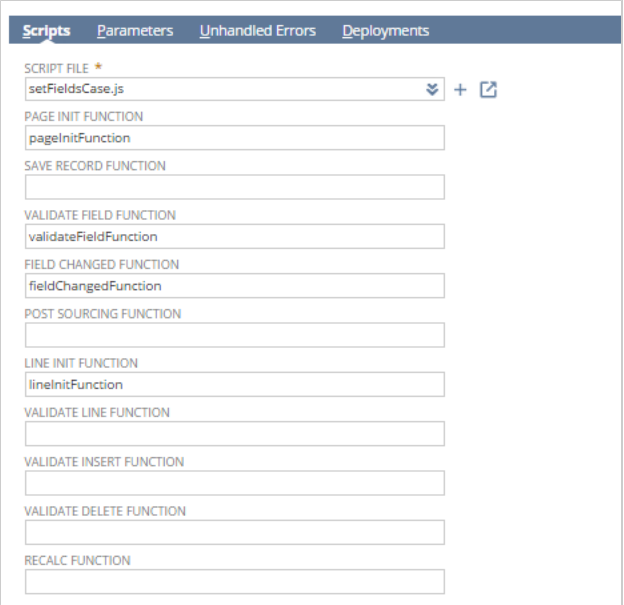
- **If defining a User Event script**, you can execute one function per operation type. For example, you can have a before load function and an after submit function defined within the same script execution. These functions must exist in either the library script file or the SuiteScript file associated with the script record.

> **Note:** For details on the before load, before submit, and after submit operations, see User Event beforeLoad Operations and User Event beforeSubmit and afterSubmit Operations.



- **If defining a record-level Client script**, type the names of the functions you want executed when the script runs. As the following figure shows, enter the function name in the field next to the client event that will trigger the function. Note that your functions must exist in either the

library script file or the SuiteScript file associated with the script record. You have the option of calling up to eight functions from within the same script file.



- **If defining a bundle installation script**, type the names of the functions you want executed before the bundle is installed, after the bundle is installed, before the bundle is updated, or after the bundle is updated. Enter the function name in the field next to the bundle deployment event that will trigger the function. Note that these functions must exist in the SuiteScript file associated with the script record. If these functions call functions in other script files, these files should be listed as library files.

> **Note:** SuiteBundler is still supported, but it will not be updated with any new features.
>
> To take advantage of new features for packaging and distributing customizations, you can use the Copy to Account and SuiteCloud Development (SDF) features instead of SuiteBundler.
>
> Copy to Account is an administrator tool that you can use to copy custom objects between your accounts. The tool can copy one object at a time, including dependencies and data. For more information, see the help topic Copy to Account Overview.
>
> SuiteCloud Development Framework is a development framework that you can use to create SuiteApps from an integrated development environment (IDE) on your local computer. For more information, see the help topic SuiteCloud Development Framework Overview.

10. On the **Parameters** tab, define possible parameters (custom fields) to pass to the functions specified in the previous step.
11. On the **Unhandled Errors** subtab, define which individual(s) will be notified if script errors occur.

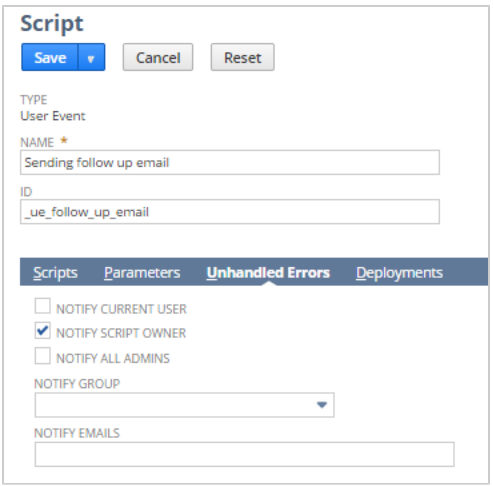
    Three types of error notifications are sent:

- An initial email is sent about the first occurrence of an error within an hour.
- An email with aggregated error information for every hour is sent. (The error counter is reset when this email is sent.)
- An email is sent about the first 100 errors that have occurred after the error counter is set to 0.

For example, an error is thrown 130 times within an hour. An initial email is sent. After the 100th occurrence, another email is sent. Since there are an additional 30 occurrences within the same hour, a final summary email is sent at the end of the hour. During the second hour, if there are only 50 occurrences of the error, only one summary email is sent at the end of that hour.

Note: By default the **Notify Script Owner** box is selected.

- (Optional) Select **Notify All Admins** if all admins should be notified.
- (Optional) Select the groups that should be notified. Only existing groups are available in the **Groups** notification dropdown list. To define new groups, go to Lists > Relationships > Groups > New.
- (Optional) Enter the email address of anyone who should be notified. You can also enter a comma-separated list of email addresses.



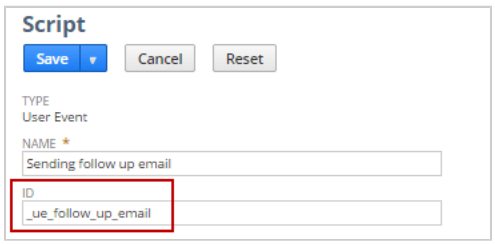
12. From the **Save** button:
    1. If you want to save the script record, but you are not ready to deploy the script, select the **Deployments** tab, clear the **Deployed** box, and click **Save**.

       Important: Scripts do not execute until they are deployed.
    2. If you want to save the script record and deploy the script, but you are not yet ready to define the script's runtime/deployment behaviors, click **Save**.
    3. If you want to save the script record and automatically open the Script Deployment page, click **Save and Deploy**. Use the Script Deployment page to define runtime behaviors such as when the script will run and which accounts the script will run in.
13. Now that you have created a Script record for your script, go to Step 5: Define Script Deployment.

Note: Although the Script record has a **Deployments** tab where you can define many of the same deployment options found on the Script Deployment page, you should define your deployments on the Script Deployment page. This page provides deployment settings that are not available on the **Deployments** tab of the Script record.

Creating a Custom Script Record ID

All Script records have an ID. Many SuiteScript APIs contain parameters such as **ID** or **scriptID**. Through these parameters you pass the scriptId or internalId of the script record. The scriptId is considered to be a custom ID you create yourself for the Script record. If you do not create your own scriptId, then the system generates an ID for you. In the documentation, the system-generated ID is referred to as the Script record's internalId.



For an example of how Script record IDs are used in a SuiteScript API call, see the help topic nlapiScheduleScript(scriptId, deployId, params).

Note: You can programmatically get the the value of a *scripId* by calling nlobjContext.getScriptId().

If you choose, you can create a custom ID for your **Script** record. If the ID field is left blank on the **Script** record, a system-generated ID is created for you. This is the ID that appears in the ID field after the Script record is saved.

Whether creating a custom ID or accepting a system-generated ID, after the script record is saved, the system automatically adds **customscript** to the front of the ID.

Why Should I Create a Custom ID?

You should use custom IDs if you plan to use the SuiteBundler feature to bundle the script and deploy it into another NetSuite account. Custom IDs reduce the risk of naming conflicts for scripts deployed into other accounts. (For details on bundling scripts, see the help topic SuiteBundler Overview.)

Note: SuiteBundler is still supported, but it will not be updated with any new features.

To take advantage of new features for packaging and distributing customizations, you can use the Copy to Account and SuiteCloud Development (SDF) features instead of SuiteBundler.

Copy to Account is an administrator tool that you can use to copy custom objects between your accounts. The tool can copy one object at a time, including dependencies and data. For more information, see the help topic Copy to Account Overview.

SuiteCloud Development Framework is a development framework that you can use to create SuiteApps from an integrated development environment (IDE) on your local computer. For more information, see the help topic SuiteCloud Development Framework Overview.

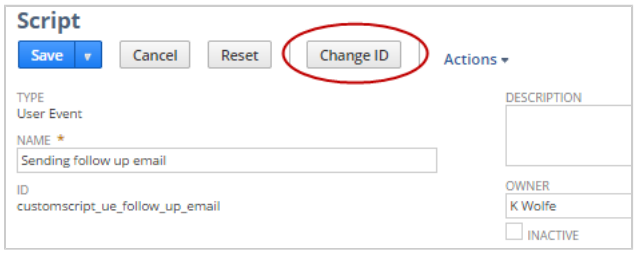
When creating a custom ID, you should insert an underscore (_) before the ID to enhance readability. For example, a custom script ID called _employeeupdates will appear as customscript_employeeupdates after the Script record is saved. Similarly, a custom deployment ID will appear as **customdeploy_employeeupdates** after the Script Deployment page is saved.

Important: Custom IDs must be in **lowercase** and contain no spaces. Also, custom IDs cannot exceed 30 characters in length. These 30 characters do not include the **customscript** or **customdeploy** prefixes that are automatically appended to the ID.

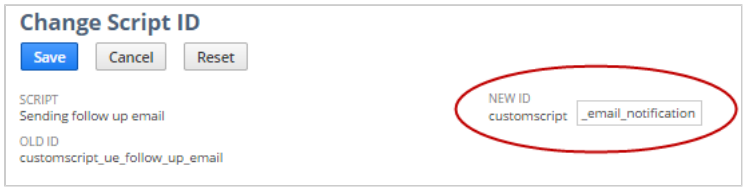
Can I Edit an ID?

Although not preferred, you can edit both custom and system-generated IDs after the Script record or script deployment is saved. To edit an ID, click the Change ID button that appears on both script record and script deployment pages AFTER each has already been saved.

The following figure shows the Change ID button on a Script Deployment page after the deployment has been saved.



After clicking the Change ID button, the Change Script ID page appears. This page shows the old ID and provides a field for creating a new ID.



Important: After you change a script record or script deployment ID, you **MUST** update all references to that ID in your code files.

Step 5: Define Script Deployment

After you have created a Script record for your SuiteScript file, you must then deploy the script into NetSuite. A script's deployment definitions, as set on the Script Deployment page, affect its runtime behaviors when it is released into NetSuite.

Some of these deployment definitions include:

- When the script will be executed
- Audience and role restrictions for the script
- Script log levels
- Deployment-specific parameter defaults
- Specific records the script will run against
- Execution and localization contexts for the script

Note that Script Deployment pages look different for each script type. For example, the Script Deployment page for a user event script will not include an area for you to define the script's deployment schedule. The Script Deployment page for a scheduled script, however, will include an area for this. Deployment pages for Suitelets will include a field for setting whether the Suitelet executes on a GET or POST request. The deployment page for a global client script will not include such a field.

Because Script Deployment pages vary depending on the script type, see Steps for Defining a Script Deployment for general steps that are applicable to most script types. See the help topic Setting Runtime Options for information about setting more advanced runtime options. In many cases, these more advanced options are specific to a particular script type.

Important Things to Note:

- You cannot edit a Script Deployment record during the time that the script associated with the deployment is running in NetSuite.
- Multiple deployments can be applied to the same record. These deployments are executed in the order specified in the UI. If an error occurs in one deployment, subsequent deployed scripts may NOT be executed. When troubleshooting, verify you are executing only one script per record type.

Steps for Defining a Script Deployment

For an overview of the Script Deployment record, be sure to see Step 5: Define Script Deployment. This section describes why a Script Deployment record is required for each script.

To define a script deployment:

1. When you save your Script record, you can immediately create a Script Deployment record by selecting **Save and Deploy** from the Script record **Save** button.

   If you want to update a deployment that already exists, go to Customization > Scripting > Script Deployments and select **Edit** next to the deployment.
2. On the Script Deployment page:
   - For Suitelet, Scheduled, and Portlet scripts, in the **Title** field, provide a name for the deployment.
   - For User Event and Client scripts, in the **Applies To** field, select the record the script will run against. In the **Applies To** field you can also select **All Records** to deploy the script to

all records that officially support SuiteScript. (For a list of these records, see the help topic SuiteScript Supported Records.)

3. In the **ID** field, if desired, enter a custom scriptId for the deployment. If you do not create a scriptId, a system-generated internalId is created for you.

   For information about whether to create a custom ID, see Creating a Custom Script Deployment ID.

4. (Optional) Clear the **Deployed** box if you do not want to deploy the script. Otherwise, accept the default. A script will not run in NetSuite until the **Deployed** box is selected.
5. In the **Status** field, set the script deployment status. See the help topic Setting Script Deployment Status.
6. (Optional) In the **Event Type** dropdown list, specify an event type for the script execution. See the help topic Setting Script Execution Event Type from the UI.
7. (Optional) In the **Log Level** field, specify which log messages will appear on the **Execution Log** tab after the script is executed. See the help topic Setting Script Execution Log Levels.
8. In the **Execute as Role** field, select whether you want the script to execute using Administrator priviledges, regardless of the permissions of the currently logged in user. See the help topic Executing Scripts Using a Specific Role.
9. On the **Audience** tab, specify the audiences for the script. See the help topic Defining Script Audience.
10. On the **Context Filtering** tab, specify the execution contexts and localization contexts for the script. See the help topics Execution Contexts and Record Localization Context.
11. On the **Links** tab (for Suitelets only), if you want to launch your Suitelet from the UI, create a menu link for the Suitelet. See Running a Suitelet in NetSuite.
12. (Optional) On the **Execution Log** tab, create custom views for all script logging details. See Creating Script Execution Logs.
13. Click **Save**.

    Note that for portlet scripts, you must enable the portlet to display on your dashboard (see Displaying Portlet Scripts on the Dashboard).

Creating a Custom Script Deployment ID

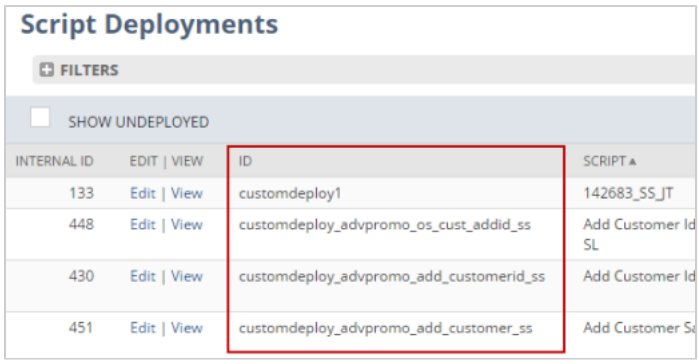
Script deployment IDs are necessary for SuiteScript development. Many SuiteScript API calls contain parameters such as *ID, scriptID*, and *deployID* that reference the IDs on the **Script Deployment** page.

These parameters allow you to pass the values of an *internalId* (a system-generated ID) or a *scriptId* (a custom ID that you provide). For an example of how script record and script deployment IDs are used in a SuiteScript API call, see the help topic nlapiScheduleScript(scriptId, deployId, params).

If you choose, you can create a custom ID for your script deployment. If the ID field is left blank on the Script Deployment page, a system-generated ID is created for you. This is the ID that appears in the ID field after the Script Deployment page is saved.

Whether creating a custom ID or accepting a system-generated ID, after the script deployment is saved, the system automatically adds **customdeploy** to the front of the ID.

The following figure shows a list of script deployments (Setup > Customization > Script Deployments). Note that there is a combination of custom IDs (for example, customdeploy_campaign_assistant) and system-generated deployment IDs (for example customdeploy1). Although **customdeploy1** is the ID for many script deployments, be aware that deployment IDs are unique only within a specific script definition.



If you are unsure whether to create your own custom ID or accept a system-generated ID, see Why Should I Create a Custom ID? for more information.

Also see Can I Edit an ID? for information about editing IDs.

Viewing Script Deployments

Note: The content in this help topic pertains to all versions of SuiteScript. Be aware that currently it may only include links or examples for SuiteScript 1.0.

There are several ways to view your script deployments:

- Go directly to the script deployment by clicking Customization > Scripting > Script Deployments.
- View deployed scripts by clicking View Deployments in the upper-right corner of the Script record.
- Click the Deployments tab on a Script record to see the deployments specific to that Script record. Next, click on a specific deployment to go to the deployment record.

  Remember: In each specific deployment record you can define default parameter values for **that** deployment. Note that first you must create the script parameter before you can define its value on a Script Deployment record.

  For more information, see the help topic Creating Script Parameters Overview in the NetSuite Help Center. Also see the help topic Setting Script Parameter Preferences for information that is specific to setting script parameter values on the Script Deployment record.
- View a list of records that have scripts associated with them at Customization > Scripting > Scripted Records. For complete details, see the help topic The Scripted Records Page in the NetSuite Help Center.

  By default, the Scripted Records list displays only those records that have at least one script associated with them.

Viewing Script and Deployment System Notes

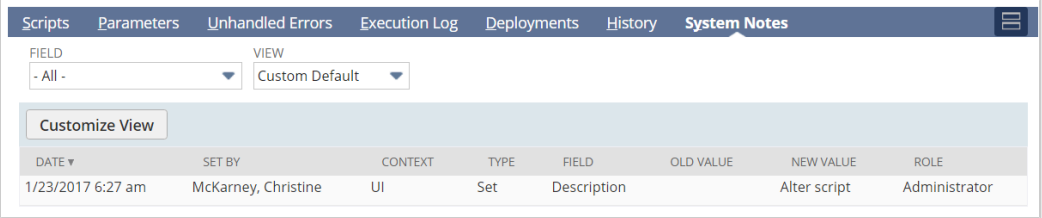
Note: This topic applies to System Notes only. For information about viewing System Notes v2, see the help topic Viewing System Notes v2.

To view the activity of your script and deployment, use the System Notes subtab.

To view script and deployment system notes:

1. Choose an option:
   - Go to Customization > Scripting > Scripts. Click **Edit** beside the script that you want to view.
   - Go to Customization > Scripting > Script Deployments. Click **Edit** beside the deployment that you want to view.
2. Click the **System Notes** subtab.

   Note: Previously, script and deployment information was listed on the History subtab, but that subtab is no longer updated. New script and deployment activity is captured on the System Notes subtab.



The system note for a change on a script or deployment record captures the following information:

- Date when the change was made
- Who made the change
- Context for the change (for example, UI)
- Type of change, for example, Edit
- Field changed
- Old value
- New value
- Role of the user who made the change (included as of 2017.2 — you have to customize the view to see the Role column)

For more information about System Notes, see the help topic System Notes Overview.

Creating Script Execution Logs

During script execution, a detailed script execution log is generated when either an unexpected error occurs or the nlapiLogExecution method is called.

For example, the following Suitelet code generates an execution log that indicates the request type of the Suitelet:

```
nlapiLogExecution('DEBUG', 'Suitelet Details', 'Suitelet method = ' + request.getMethod());
```

For more information about the nlapiLogExecution function, see the help topic nlapiLogExecution(type, title, details).

There are two ways to view the script execution logs:

- The **Script Execution Logs** list that can be accessed through Customization > Scripting > Script Execution Logs.

  The list of script execution logs is an enhanced repository that stores all log details for 30 days. The filter options on this page allow you to search for specific logs. See the help topic Viewing a List of Script Execution Logs.
- The **Execution Log** tab that appears on Script pages, Script Deployment pages, and the SuiteScript Debugger.

  The execution log tab displays logs for a specific script, but these logs are not guaranteed to persist for 30 days. These logs are searchable, and you can customize views to find specific logs. See the help topic Using the Script Execution Log Tab.

SuiteScript 1.0 Working with Records

The following topics are covered in this section:

Note: For a list of SuiteScript supported records, see the help topic SuiteScript Supported Records in the NetSuite Help Center.

SuiteScript 1.0 Working with Records

The SuiteScript API includes several Record APIs that interact with the entire NetSuite record object. When you work with Record APIs, you are doing things such as creating, deleting, copying, or loading all elements of a record.

Whether you are working with **standard** NetSuite records (for example, Sale Order, Invoice, Customer, Vendor) or **custom** records you have created using SuiteBuilder, you will use all the same Record APIs to interact with the record object.

Important: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). The NetSuite UI should only be accessed using SuiteScript APIs.

Note: For a list of SuiteScript supported records, see the help topic SuiteScript Supported Records in the NetSuite Help Center.

SuiteScript 1.0 How Records are Processed in Scripting

When using SuiteScript to interact with records, NetSuite will process your data and execute all core NetSuite business logic after the data is submitted.

As a SuiteScript developer, note the following when working with records:

- You can write your SuiteScript code without having to "reverse engineer" NetSuite logic. When you submit the record, the validation, sourcing, and recalculation logic that automatically occurs in the UI will also occur when you **submit** a record in SuiteScript. This is considered the **standard** mode of processing.
- When scripting in **standard** mode (as opposed to **dynamic** mode), you do not have to write your code in a way that references fields in the order they are referenced in the UI. (To learn more about **dynamic** mode scripting, see SuiteScript 1.0 Working with Records in Dynamic Mode.)
- When scripting in **standard** mode (as opposed to **dynamic** mode), you will have to submit and then reload a record to know the values for calculated fields. For example, you will need to submit and then

reload a record to know how much an item will cost after NetSuite applies tax and shipping amounts to the data submitted.

SuiteScript 1.0 Working with Records in Dynamic Mode

When creating, copying, loading, or transforming records in SuiteScript, you have the choice of working with records in dynamic mode. When scripting in dynamic mode, you are working with a record in a way that emulates the behaviors of the UI. For example, in the UI, if you create a new sales order, select a custom form, and specify a customer, various field values throughout the new record are sourced (automatically populated, validated, or calculated in real-time).

When you programmatically create a new sales order in dynamic mode, all of the same sourcing behaviors that occur in the UI also occur in your script as each line is executed. You can obtain sourced, validated, and calculated field values in real-time without first having to submit the record.

When working with records in standard mode, you must submit and then load the record to obtain these same values. In standard mode, all business logic is executed only after the data is submitted.

Note: Calls to the APIs nlapiGetNewRecord() and nlapiGetOldRecord() always return a record in standard mode. They never return a record in dynamic mode.

SuiteScript 1.0 How do I enable dynamic mode?

You use the optional **initializeValues** parameter in the following APIs to control whether a record is processed in dynamic mode.

- nlapiCopyRecord(type, id, initializeValues)
- nlapiCreateRecord(type, initializeValues)
- nlapiLoadRecord(type, id, initializeValues)

In nlapiTransformRecord(type, id, transformType, transformValues), use the **transformValues** parameter to control whether one record will be transformed into another in dynamic mode.

Important: If you provide no value for the initializeValues parameter, records will copy, create, load, and transform in the standard mode of execution, where sourcing, validation, and recalculations occur only after a record is submitted.

Important: Calls to the APIs nlapiGetNewRecord() and nlapiGetOldRecord() always return a record in standard mode. They never return a record in dynamic mode.

Example

The initializeValues parameter is an Object that can contain an array of name/value pairs of defaults to be used during record initialization. To initialize a record in dynamic mode, you set the **recordmode** initialization type to **dynamic**. For example:

- var record = nlapiCopyRecord('salesorder', 55, {recordmode: 'dynamic'});
- var record = nlapiCreateRecord('salesorder', {recordmode: 'dynamic'});

- var record = nlapiLoadRecord('salesorder', 111, {recordmode: 'dynamic'});
- var transformRecord = nlapiTransformRecord('salesorder', 111, 'itemfulfillment', {recordmode: 'dynamic'});

Note: For a list of additional initialization types that can be specified for the initializeValues parameter, see SuiteScript 1.0 Record Initialization Defaults in the NetSuite Help Center. Note that you do not need to run scripts in dynamic mode to use these other initialization types.

SuiteScript 1.0 Is dynamic mode better than standard mode?

Yes and no. There are obvious benefits to getting / setting field values in real-time. When working with records in dynamic mode, you do not need to submit the record for all business logic to be executed. In dynamic mode you can get field values and write logic around these values without having to submit a record and wonder what will be returned when the record is reloaded.

Note, however, when scripting in dynamic mode, the order in which you set field values matters. For some developers, this aspect might feel constraining. It is likely that scripting in dynamic mode will require you to refer back to the UI often. For example, on an invoice in the UI, you would not set the Terms field before setting the Customer field (see figure). The reason is that as soon as you set the Customer field, the value of Terms will be overridden. On an invoice, the value of Terms is sourced from the terms specified on the Customer record. The same behavior will happen in dynamic scripting. In your scripts, if you do not correctly set field values in the order that they are sourced in the UI, some of the values you set could be overridden.

In non-dynamic mode, you do not have to worry about the order in which you set field values. For some developers, this fact alone will make scripting in non-dynamic mode preferable.

SuiteScript 1.0 Can I change existing code to run in dynamic mode?

Yes. If you decide to convert an existing script to one that runs dynamically, you will need to do the following:

1. Set the new initializeValues parameter to {recordmode: 'dynamic'} - for example:

   var recDynamic = nlapiCreateRecord('salesorder', **{recordmode: 'dynamic'}**);
2. Ensure that the field values you have set in your scripts are set in the correct order.

   As the second example Field Ordering shows, if you do not set your fields in the right order, you may end up overriding certain field values.
3. Update any code that uses either the nlapiSetLineItemValue(...) function or the nlobjRecord.setLineItemValue(...) method.

   Neither of these APIs will execute in dynamic mode. For details, see Working with Sublists in Dynamic Mode and Client SuiteScript in the NetSuite Help Center.

SuiteScript 1.0 Standard vs. Dynamic Mode Code Samples

This section uses code snippets to further demonstrate some of the differences between scripting in standard mode and dynamic mode. The areas covered are field sourcing, field ordering, and field calculation.

Sourcing

Example 1: Standard mode (sourcing does not occur real-time)

This sample shows a script executing in standard (non-dynamic) mode. In standard mode, no real-time sourcing occurs. You cannot get values that you have not already set in your script.

```
// Create a new sales order and set the customer
var record = nlapiCreateRecord('salesorder');
record.setFieldValue('entity', 343);

// Try to get the value of salesrep. Note that null is returned because the value
// of salesrep is not sourced as you step through the code
var sr = record.getFieldValue('salesrep');
```

Example 2: Dynamic mode (sourcing occurs as each line executes)

This sample shows the same script, but running in dynamic mode. In this script, the value of salesrep is sourced after the entity is set. Ultimately you can write fewer lines of code, as you are able to get many field values without first having to set them.

```
// Create a new sales order and set the customer
var record = nlapiCreateRecord('salesorder', {recordmode: 'dynamic'});
record.setFieldValue('entity', 343);
// Get the value of salesrep. Note that John Smith will be returned. The value of the salesrep
// field is sourced from the salesrep value specified on the customer record
var sr = record.getFieldValue('salesrep');
```

Field Ordering

Example 1: Standard mode (field ordering does not matter)

This sample shows that the order in which you set values does not matter in standard mode. In this sample, you can set a sales rep before setting the entity, even though in the UI there is a sourcing relationship between these two fields. In the UI, after you set the entity, the value of salesrep is automatically sourced. In standard mode, field sourcing relationships are not respected.

```
// Create a sales order. First set salesrep, then set the customer (entity). When the
// record is submitted, salesrep remains as 88, the internal ID for sales rep Bud Johnson.
var record = nlapiCreateRecord('salesorder');
record.setFieldValue('salesrep', 88);
record.setFieldValue('entity', 343);
```

Example 2: Dynamic mode (field ordering matters)

In dynamic mode, if you write the same script, the value of salesrep will be overridden when the next line of code is executed. In this example, when you submit the record the value of salesrep will change from 88 to 333 (the value of the salesrep specified on the customer record).

```
var record = nlapiCreateRecord('salesorder', {recordmode: 'dynamic'});
record.setFieldValue('salesrep', 88);
record.setFieldValue('entity', 343);
```

In dynamic mode, if you want salesrep to remain as 88, you must write:

```
var record = nlapiCreateRecord('salesorder', {recordmode: 'dynamic'});
```

```
record.setFieldValue('entity', 343);
record.setFieldValue('salesrep', 88);
```

Example 3: Dynamic mode (field ordering matters)

For transaction records in dynamic mode, the entity must be set prior to setting the subsidiary.

```
var transRec = nlapiCreateRecord('invoice', {recordmode: 'dynamic'});
transRec.setFieldValue('entity', Customer);
transRec.setFieldValue('subsidiary', USASubsidiary);
nlapiSubmitRecord(transRec);
```

Field Calculation

Example 1: Standard mode (field totals are not calculated in real-time)

In this beforeSubmit user event script, if a line is added to a sales order, the totals are not recalculated until you submit the line. The following sample shows a new line being added in a beforeSubmit script, however, the total is not recalculated.

```
function beforesubmit(type)
{
var record = nlapiGetNewRecord();
record.selectNewLineItem('item');
record.setCurrentLineItemValue('item', 'item', 441);
record.setCurrentLineItemValue('item', 'quantity', '2');
record.commitLineItem('item');
}
```

Example 2: Dynamic mode (field totals are calculated real-time)

In dynamic mode, you can add a line to a sublist (the Items sublist in this example) and not worry about recalculating the amount and subtotal fields. Even before the record is submitted, you can get accurate data in a beforeSubmit event.

```
function beforesubmit(type)
{
var record = nlapiGetNewRecord({recordmode: 'dynamic'});
record.selectNewLineItem('item');
record.setCurrentLineItemValue('item', 'item', 441);
record.setCurrentLineItemValue('item', 'quantity', '2');
record.commitLineItem('item');
}
```

SuiteScript 1.0 Client Scripting and Dynamic Mode

Generally speaking, the concept of dynamic mode does not apply to client scripting. The only exception is on the pageInit client event. Scripting in a “current record context” is always in dynamic mode.

Note that client remote object scripting does not support dynamic scripting. In a client script, if you attempt to copy, create, load, or transform a “remote object” (an object on the NetSuite server), you will not be able to work with the record in dynamic mode. (See also Client Remote Object Scripts.)

The following is an example of a client remote object script. On the saveRecord client event, an estimate record is created. If you attempt to use a client script to create the record in dynamic mode, an error is thrown.

For example:

```
function onSave()
{
var rec = nlapiCreateRecord('estimate', {recordmode: 'dynamic'}); // this will not work
 rec.setFieldValue('entity','846');
rec.insertLineItem('item',1);
rec.setLineItemValue('item','item', 1, '30');
rec.setLineItemValue('item','quantity', 1, '500');
var id = nlapiSubmitRecord(rec, true);
return true;
}
```

Working with Subrecords in SuiteScript

The following topics are covered in this section:

Note: For a list of SuiteScript supported records and subrecords, see the help topic SuiteScript Supported Records in the NetSuite Help Center.

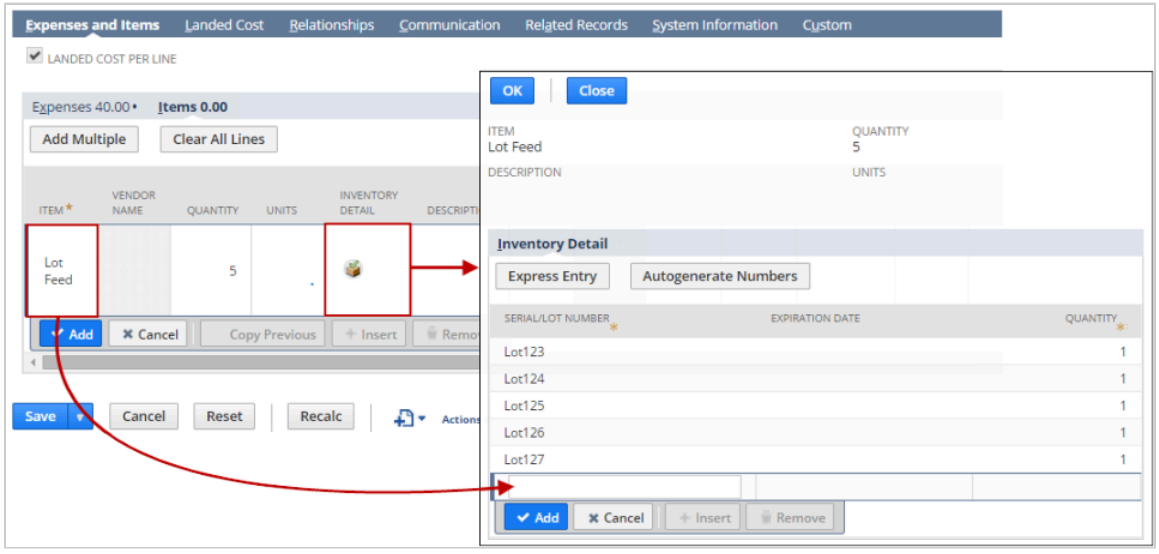
What is a Subrecord?

A subrecord includes many of the same elements of a standard NetSuite record (body fields, sublists and sublist fields, and so on). However, subrecords must be created, edited, removed, or viewed from within the context of a standard (parent) record.

The purpose of a subrecord is to hold key related data about the parent record. For example, a parent record would be a Serialized Inventory Item record. This record defines a type of item. A subrecord would be an Inventory Detail subrecord. This is a subrecord that contains all data related to where the item might be stored in a warehouse. In this way, the subrecord contains data related to the item, but not data that directly defines the item. Without the parent record, the subrecord would serve no purpose.

The following figure shows an Inventory Detail subrecord. Its parent is a Bill record. In this figure the Inventory Detail subrecord is accessed through the Inventory Details sublist field. The Inventory Detail subrecord contains the inventory details for the item called the Lot Feed item.

In this case the parent record is still the Bill record, even though the subrecord tracks inventory details related to the Lot Feed item. Ultimately it is the Bill record that must be saved before the subrecord (pertaining to an item on the Bill) is committed to the database.



Creating Subrecord Custom Entry Forms

You can create custom entry forms for subrecords by going to Customization > Forms > Entry Forms. A currently supported subrecord type is Inventory Detail, which is associated with the Advanced Bin / Numbered Inventory Management feature. In the Custom Entry Forms list, you can select Customize next to Inventory Detail to create a custom form for this subrecord type.

Note that when you create a custom form for Inventory Detail, you can use the Actions tab to add new buttons to the custom form. When clicked, these buttons will execute client SuiteScript. However, you cannot customize the buttons that currently exist on the Inventory Detail record. These buttons are required; without them you cannot save this subrecord to its parent record.

Also note that the Store Form with Record preference is not currently supported for custom subrecord forms. You can, however, set the customized subrecord form as Preferred.

Additionally, like any other custom form, you can attach client scripts to the Custom Forms tab.

Using the SuiteScript API with Subrecords

The SuiteScript API includes several Subrecord APIs to interact with the subrecord object (nlobjSubrecord).

Important: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). The NetSuite UI should only be accessed using SuiteScript APIs.

Using SuiteScript you can create and access subrecords through a **body field** on a parent record. (See Creating and Accessing Subrecords from a Body Field for details.) You can also create and access subrecords through a **sublist field** on a parent record. (See Creating and Accessing Subrecords from a Sublist Field for details.)

To set values on sublists that appear on subrecords, you will use some of the same Sublist APIs used to set values on sublists appearing on parent records. See Setting Values on Subrecord Sublists for details.

To save a subrecord, you must follow the pattern outlined in the section Saving Subrecords Using SuiteScript.

Creating and Accessing Subrecords from a Body Field

If you want to create a subrecord to hold data related to the parent, you can do so from a **body field** on the parent. When working with subrecords from a body field on the parent, you will use the following APIs if you are working with the parent record in a “current record” context, such as in a user event script or a client script:

- nlapiCreateSubrecord(fldname)
- nlapiEditSubrecord(fldname)
- nlapiRemoveSubrecord(fldname)
- nlapiViewSubrecord(fldname)

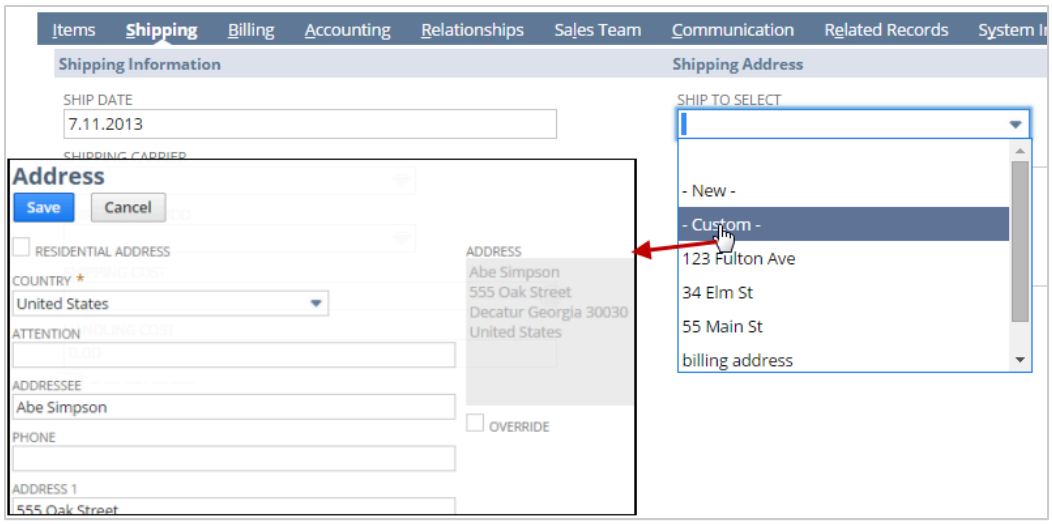
Note: nlapiCreateSubrecord(fldname) and nlapiEditSubrecord(fldname) are not supported in client scripts deployed on the parent record.

If you are loading the parent record using SuiteScript, you will use these methods on the nlobjRecord object to create and access a subrecord:

- createSubrecord(fldname)
- editSubrecord(fldname)
- removeSubrecord(fldname)
- viewSubrecord(fldname)

The following figure shows the Ship To Select (shippingaddress) body field on the Sales Order parent record. To create a custom shipping address subrecord on the parent, you will do so from this body field. After creating the subrecord, you can then edit, remove, or view the subrecord through the same body field on the parent record.

Note: For additional information about creating custom shipping addresses, see Scripting Billing and Shipping Addresses.



Note that after creating or editing a subrecord, you must save both the subrecord and the parent record for the changes to be committed to the database. See Saving Subrecords Using SuiteScript for more information.

For code samples showing how "body field" subrecord APIs are used, see Using SuiteScript with Advanced Bin / Numbered Inventory Management or Scripting Billing and Shipping Addresses.

Creating and Accessing Subrecords from a Sublist Field

If you want to create a subrecord to hold data for a record in a sublist, you can do so from a sublist field.

When working with subrecords from a **sublist field** on the parent record, you will use these APIs if you are working with the parent record in a "current record" context, such as in a user event script or a client script:

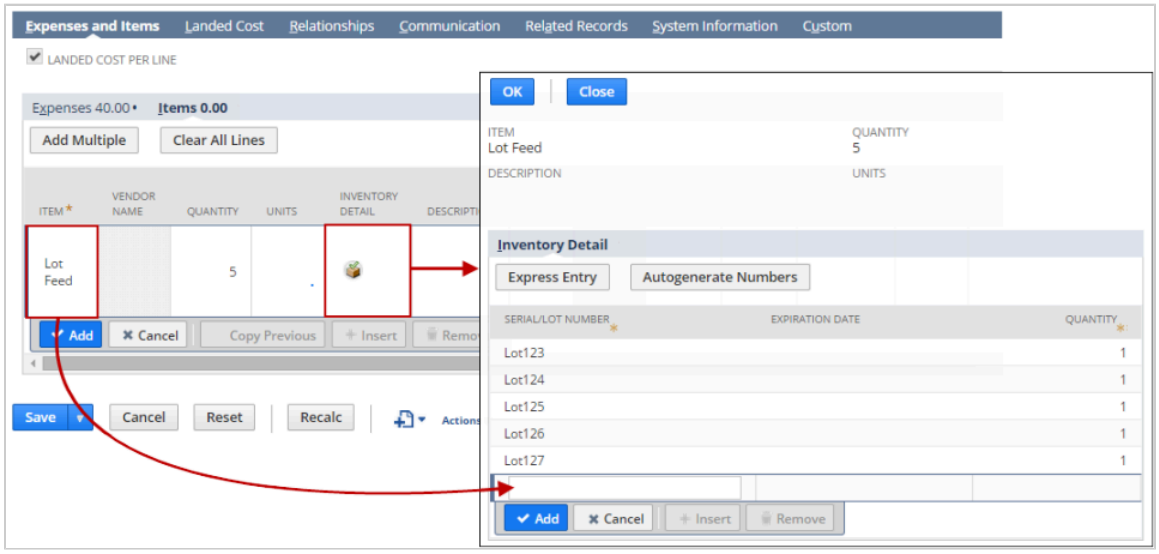
- nlapiCreateCurrentLineItemSubrecord(sublist, fldname)
- nlapiEditCurrentLineItemSubrecord(sublist, fldname)
- nlapiRemoveCurrentLineItemSubrecord(sublist, fldname)
- nlapiViewCurrentLineItemSubrecord(sublist, fldname)
- nlapiViewLineItemSubrecord(sublist, fldname, linenum)

Note: nlapiCreateCurrentLineItemSubrecord() and nlapiEditCurrentLineItemSubrecord() are not currently supported in client scripts.

If you are loading the parent record using SuiteScript, and you want to create/access a subrecord from a sublist, you will use these methods on the nlobjRecord object:

- createCurrentLineItemSubrecord(sublist, fldname)
- editCurrentLineItemSubrecord(sublist, fldname)
- removeCurrentLineItemSubrecord(sublist, fldname)
- viewCurrentLineItemSubrecord(sublist, fldname)
- viewLineItemSubrecord(sublist, fldname, linenum)

This figure shows that the Inventory Detail subrecord is being edited on the Items sublist.

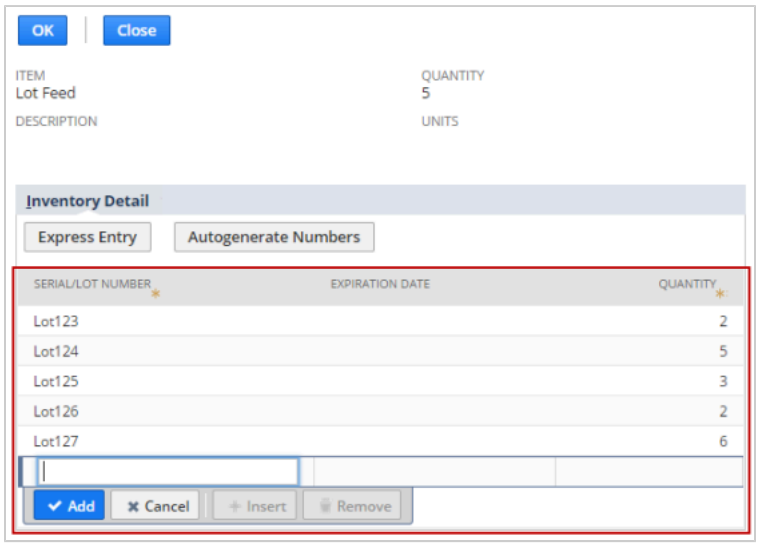


For code samples showing how "sublist field" subrecord APIs are used, see Using SuiteScript with Advanced Bin / Numbered Inventory Management.

Setting Values on Subrecord Sublists

When working with sublists on subrecords (see figure), you will use the following Sublist APIs on the nlobjRecord object:

- selectNewLineItem(group) - use if creating a new sublist line
- selectLineItem(group, linenum) - use if selecting an existing line on the sublist
- setCurrentLineItemValue(group, name, value) - use to set the values on a line
- commitLineItem(group, ignoreRecalc) - use to commit the line



Important: The nlapiSetLineItemValue(...) and nlobjRecord.setLineItemValue(...) APIs are NOT supported when scripting a subrecord's sublist.

The following sample shows how to use Sublist APIs to set values on a subrecord sublist.

```
var qtytobuild = 2;
    var obj = nlapiCreateRecord('assemblybuild', {recordmode:'dynamic'});
    obj.setFieldValue('subsidiary', 3 );
    obj.setFieldValue('item', 174);
    obj.setFieldValue('quantity', qtytobuild);
    obj.setFieldValue('location', 2);

    var bodySubRecord = obj.createSubrecord('inventorydetail');
    var ctr;
    for(ctr = 1; ctr <= qtytobuild ; ctr ++)
    {

   //Here we are selecting a new line on the Inventory Assignment sublist on the subrecord
   bodySubRecord.selectNewLineItem('inventoryassignment');
   bodySubRecord.setCurrentLineItemValue('inventoryassignment', 'newinventorynumber',
      'amsh_' + ctr);
        bodySubRecord.setCurrentLineItemValue('inventoryassignment', 'quantity', 1);
        bodySubRecord.setCurrentLineItemValue('inventoryassignment', 'binnumber', 3);
        bodySubRecord.commitLineItem('inventoryassignment');
    }
    bodySubRecord.commit();

   //Here we are selecting and editing an existing line on the Components sublist
    //on the parent record. Note that when working with the Assembly Build record only,
    //the internal ID for the Inventory Details field on the Components sublist is
   // 'componentinventorydetail'. This is because the Assembly Build record already contains
   //an Inventory Details (inventorydetails) body field.
    obj.selectLineItem('component', 1);
    obj.setCurrentLineItemValue('component', 'quantity', qtytobuild);
    var compSubRecord = obj.createCurrentLineItemSubrecord('component',
     'componentinventorydetail');

  //Here we are selecting and editing a new line on the Inventory Assignment sublist on
 //the subrecord.
       compSubRecord.selectNewLineItem('inventoryassignment');
        compSubRecord.setCurrentLineItemValue('inventoryassignment', 'binnumber', 3);
        compSubRecord.setCurrentLineItemValue('inventoryassignment', 'quantity', 2);
        compSubRecord.commitLineItem('inventoryassignment');
        compSubRecord.commit();

   obj.commitLineItem('component');
   var id = nlapiSubmitRecord(obj);
```

```
    obj = nlapiLoadRecord('assemblybuild', id);
    var subrecord = obj.viewSubrecord('inventorydetail');
    subrecord.selectLineItem('inventoryassignment', 1);

    var str;

    str = subrecord.getCurrentLineItemValue('inventoryassignment', 'newinventorynumber');
    if (str!= 2)
    {

    }
```

For additional code samples showing how to use Sublist APIs in the context of a subrecord, see Using SuiteScript with Advanced Bin / Numbered Inventory Management.

Saving Subrecords Using SuiteScript

To save a subrecord to a parent record you will call nlobjSubrecord.commit(). You must then save the subrecord's parent record using nlapiSubmitRecord(record, doSourcing, ignoreMandatoryFields). If you do not commit both the subrecord and the parent record, all changes to the subrecord are lost.

In the following sample an Inventory Detail subrecord is edited from the 'inventorydetail' field on the Items sublist. Next, values are set on the 'inventoryassignment' sublist. This is the sublist on the Inventory Detail subrecord. After this sublist is edited, you must call commitLineItem(...) to commit the changes to this sublist.

Next, you call commit() on the nlobjSubrecord object to commit the subrecord to the parent record. After that, you must call commitLineItem(...) again, but this time on the Items sublist of the parent record. This is necessary because, ultimately what you are doing in this script is updating the Items sublist.

Finally, you must call nlapiSubmitRecord(...) on the Purchase Order record. This is the parent record and must be saved for all changes in the script to be committed to the database.

```
var record2= nlapiLoadRecord('purchaseorder', id, {recordmode: 'dynamic'});
record2.selectLineItem('item', 1);
record2.setCurrentLineItemValue('item', 'quantity', 2);

var subrecord2= record2.editCurrentLineItemSubrecord('item', 'inventorydetail');
subrecord2.selectLineItem('inventoryassignment', 1);
subrecord2.setCurrentLineItemValue('inventoryassignment', 'inventorynumber', 'working123');
subrecord2.selectNewLineItem('inventoryassignment');
subrecord2.setCurrentLineItemValue('inventoryassignment', 'inventorynumber', '2ndlineinventorynumber');
subrecord2.setCurrentLineItemValue('inventoryassignment', 'quantity', '1');
subrecord2.commitLineItem('inventoryassignment');

subrecord2.commit();

record2.commitLineItem('item');

var id = nlapiSubmitRecord(record2);
```

Guidelines for Working with Subrecords in SuiteScript

The following are guidelines you must following when working with subrecords.

- In SuiteScript, you must first create or load a parent record before you can create/access a subrecord. You can create/load the parent record in either standard mode or dynamic mode.
- You cannot create or edit a subrecord in a beforeLoad user event script. You must use a pageInit client script if you want to create/edit a subrecord before the end user has access to the page.

- If you attempt to edit or view a subrecord that does not exist, *null* will be returned.
- In a client script attached or deployed to the parent record, you cannot create or edit a subrecord; you can only view or delete subrecords.
- There is no automatic client-side validation on a subrecord when a field is changed on the parent record. For example, if a user changes the quantity of an item on an item line, there is no detection of a quantity mismatch between the item line and its Inventory Detail. Note, however, validation can be implemented programmatically using a validateLine() call.
- To save a subrecord, you must commit both the subrecord, the line the subrecords appears on (if accessing a subrecord through a sublist), and the parent record. See Saving Subrecords Using SuiteScript for complete details.
- If you call one of the Subrecord APIs on a non-subrecord field, an error is thrown.
- The following sublist and body field APIs are not supported on subrecords:
  - nlapiGetLineItemValue(type, fldname, linenum)
  - nlapiGetLineItemText(type, fldnam, linenum)
  - nlapiFindLineItemValue(type, fldnam, val)
  - nlapiGetCurrentLineItemText(type, fldnam)
  - nlapiGetCurrentLineItemValue(type, fldnam)
  - nlapiGetFieldValue()
  - nlapiGetFieldText()
- When using the Assembly Build record as a parent record, be aware that this record has two inventorydetail fields: one on the body of the record and the other as a field on the Components sublist. When creating/assessing a subrecord from the body field, use **inventorydetail** as the internal ID for the fldname parameter. When creating/accessing a subrecord from the sublist field on the Components sublist, use **componentinventorydetail** as the internal ID for the fldname parameter. To see an example, see the code sample provided in Setting Values on Subrecord Sublists.

Working with Specific Subrecords in SuiteScript

- Using SuiteScript with Advanced Bin / Numbered Inventory Management
- Using SuiteScript with Address Subrecords

Using SuiteScript with Advanced Bin / Numbered Inventory Management

When you write scripts with the Advanced Bin / Numbered Inventory Management feature enabled, your scripts must reference not only a main (parent) record or transaction, but also the Inventory Details "subrecord." In the UI the Inventory Details subrecord appears as a pop-up when you click the Inventory Details body field or sublist field. In SuiteScript, this pop-up is considered a subrecord object (nlobjSubrecord), which is created and accessed through its own set of Subrecord APIs.

See the following topics for details specific to using SuiteScript with the Advanced Bin / Numbered Inventory Management feature:

- SuiteScript and Advanced Bin Management – Overview
- Scripting the Inventory Detail Subrecord
- Sample Scripts for Advanced Bin / Numbered Inventory Management

See these topics for general information about working with subrecords:

- Working with Subrecords in SuiteScript
- Guidelines for Working with Subrecords in SuiteScript

Warning: If you are currently using SuiteScript with the **basic** Bin Management feature, your scripts will no longer work after you enable the Advanced Bin / Numbered Inventory Management feature. This is especially true if you have written client scripts, which will have to be completely rewritten as server scripts. See Updating Your Scripts After Enabling Advanced Bin / Numbered Inventory Management for more information.

SuiteScript and Advanced Bin Management – Overview

The following figure draws a comparison between how the advanced bin management feature appears in the UI and how that translates into SuiteScript.

Note: Even with the Advanced Bin / Numbered Inventory Management feature enabled, not all items will require an Inventory Details subrecord. (See the help topic Advanced Bin / Numbered Inventory Management for details on which items will use Inventory Detail subrecords.)

The figure below shows a subrecord being accessed from a sublist field. (Note that subrecords can also be created and accessed from a body field on the parent record. See Creating and Accessing Subrecords from a Body Field for more information.)

The numbers below further explain the numbers in the figure.

1. **Bill** : This is a parent record. In both the UI and in SuiteScript you must have a parent record before you can create or access a subrecord. Without the parent, the subrecord has no relevance.

   In SuiteScript, you can create/load the parent record in either standard mode or dynamic mode.
2. **Items sublist** : In the case of this figure, a subrecord is being created for the Lot Bin Item referenced on the Items sublist.

   Important: Note that the parent is still considered to be the Bill record, even though the subrecord is being created for the Lot Bin Item. Ultimately it is the Bill record that must be saved before any changes to the Items sublist or the Inventory Details subrecord are committed to the database.
3. **Inventory Details sublist field** : As you enter information for the Lot Bin Item, in the UI you click the Inventory Details icon to create a new Inventory Details subrecord for this item.

   In SuiteScript, if you are creating a subrecord from the Inventory Details sublist field, you will call either nlapiCreateCurrentLineItemSubrecord(sublist, fldname) or nlobjRecord.createCurrentLineItemSubrecord(sublist, fldname), depending on the nature of your script. The sublist is 'item' and the fldname is 'inventorydetail'.
4. **Inventory Detail subrecord** : This is a subrecord containing the inventory details for the Lot Bin Item.
5. **Inventory Assignment sublist** : This is the sublist that appears on the Inventory Details subrecord. Although there is no UI label for the Inventory Assignment sublist, this is the sublist you will reference to add and edit new sublist lines.

   In SuiteScript, you create and edit lines on the Inventory Assignment sublist using the APIs described in Setting Values on Subrecord Sublists.
6. **OK button:** In the UI you click the OK button to save the subrecord (note, however, the subrecord is not yet saved on the server).

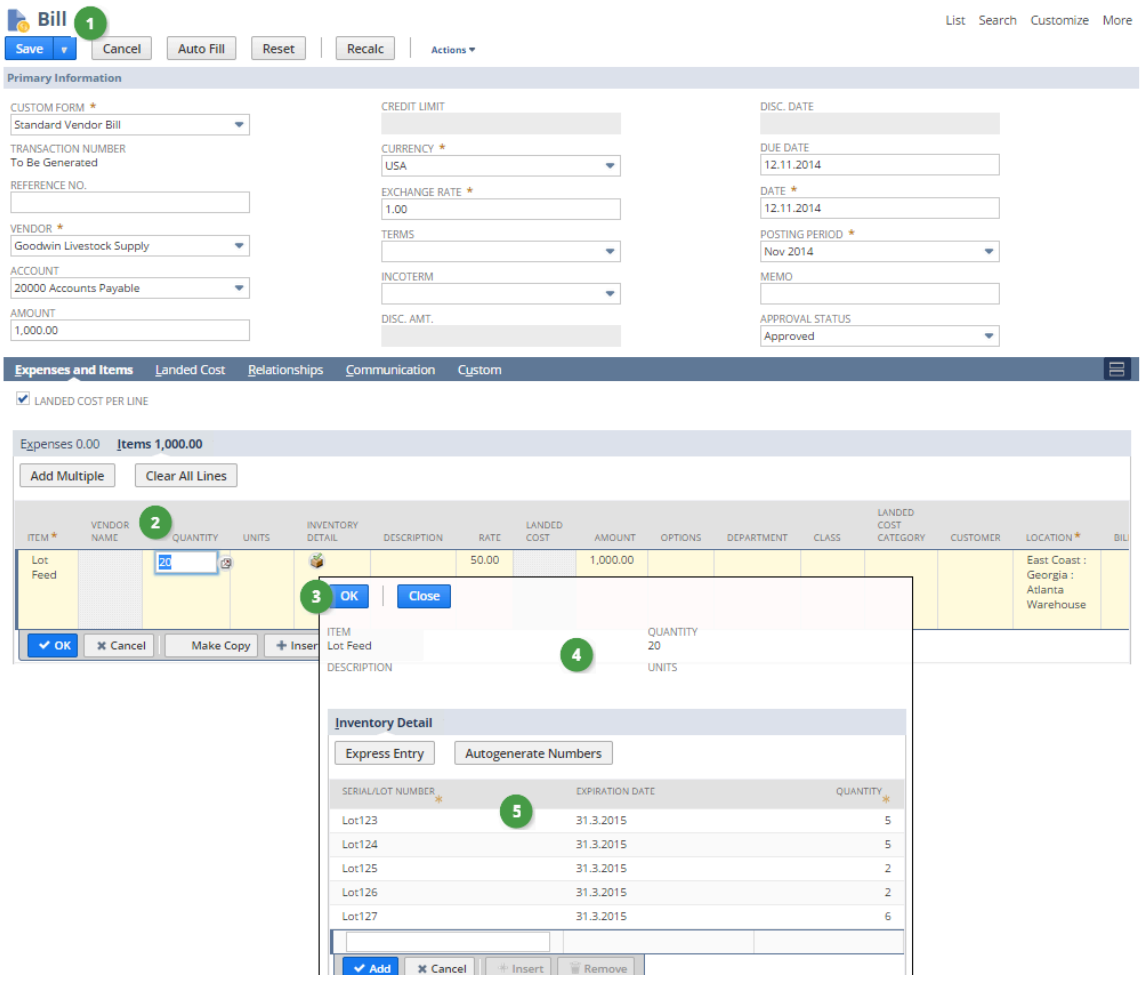
   In SuiteScript, to save a subrecord you must call nlobjSubrecord.commit() on the subrecord, nlobjRecord.commitLineItem() - - if you have created a subrecord on a sublist line.

7. **Add button on sublist:** In the UI you must click the Add button on a sublist to commit your changes to the line.

   In SuiteScript, you will call nlobjRecord.commitLineItem() - - if you have created a subrecord on a sublist line.

8. **Save button on parent record:** In the UI, you will click the Save button on the parent record to commit **all** changes to the server.

   In SuiteScript, you will call nlapiSubmitRecord(...) on the parent record. See Saving Subrecords Using SuiteScript for complete details.



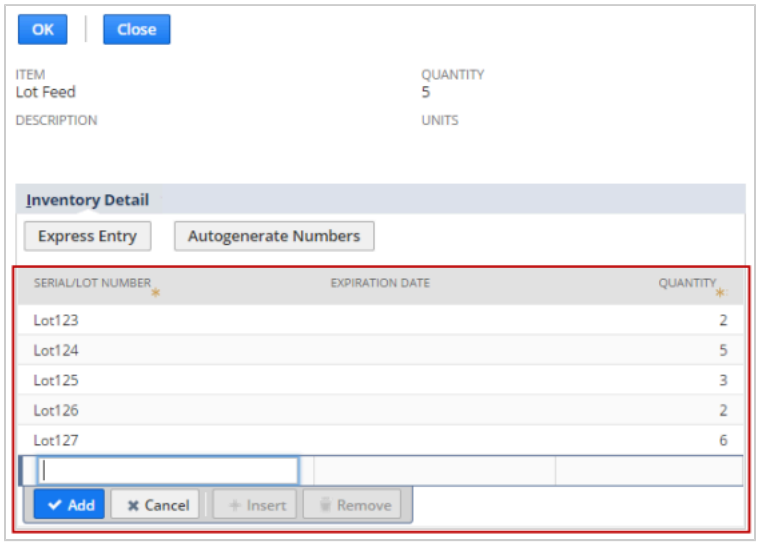
Scripting the Inventory Detail Subrecord

Like a standard (parent) record, the Inventory Details subrecord contains body fields, a sublist, and sublist fields. You can create and access the Inventory Details subrecord from a body field on the parent record or from a sublist field. See Creating and Accessing Subrecords from a Body Field and Creating and Accessing Subrecords from a Sublist Field for details.

Note: Currently you cannot create or edit subrecords using Client SuiteScript.

The sublist that appears on the Inventory Detail subrecord is referred to as the Inventory Assignment sublist, even though it has no UI label. In your scripts, use the ID `inventoryassignment` to reference this

sublist. In the figure below, the `inventoryassignment` sublist is used to assign serial/lot numbers and which serial/lot belongs to which bin.



To set values on the Inventory Assignment sublist, you will use some of the same Sublist APIs used to set values on other sublist in the system. See Setting Values on Subrecord Sublists for details.

To save a subrecord, you must follow the pattern outlined in the section Saving Subrecords Using SuiteScript.

Internal IDs for the Inventory Details Subrecord

Use the following internal IDs when writing SuiteScript against the Inventory Details subrecord:

Subrecord Elements	Internal IDs	Notes
Inventory Details subrecord	inventorydetail	This is the internal ID for the Inventory Details subrecord. When using any of the Subrecord APIs, the value you set for the fldname parameter is **inventorydetail**. It is through this field that you will create a subrecord. **Important:** When using the Assembly Build record as a parent record, be aware that this record has two inventorydetail fields: one on the body of the record and the other as a field on the Components sublist. When creating/assessing a subrecord from the body field, use **inventorydetail** as the internal ID for the fldname parameter. When creating/accessing a subrecord from the sublist field on the Components sublist, use **componentinventorydetail** as the internal ID for the fldname parameter. To see an example, see the code sample provided in Setting Values on Subrecord Sublists.
Body fields on the Inventory Details subrecord	item	
	location	
	tolocation	

Subrecord Elements	Internal IDs	Notes
	itemdescription	
	quantity	
	baseunitquantity	
	unit	
	totalquantity	
Inventory Assignment sublist	inventoryassignment	This is the internal ID for the Inventory Assignment sublist that appears on the Inventory Details subrecord. Note that the Inventory Assignment as no UI label.
Sublist fields on the Inventory Assignment sublist	receiptinventorynumber	This field is for entering text of serial/lot number for create. Use the receiptinventorynumber internal ID in both the inventory detail of an item receipt and inventory adjustment.
	issueinventorynumber	This is the select field where users pick an issueinventorynumber out of inventory.
	binnumber	
	tobinnumber	
	expirationdate	
	quantity	
	quantityavailable	

Sample Scripts for Advanced Bin / Numbered Inventory Management

The following samples are provided in this section:

- Creating an Inventory Detail Subrecord
- Editing an Inventory Detail Subrecord
- Removing an Inventory Detail Subrecord from a Sublist Line
- Canceling an Inventory Detail Subrecord
- Viewing an Inventory Detail Subrecord
- Updating Your Scripts After Enabling Advanced Bin / Numbered Inventory Management

As you begin writing your own scripts, you should review Guidelines for Working with Subrecords in SuiteScript. This section highlights many of the rules that are enforced when scripting with subrecords.

Creating an Inventory Detail Subrecord

This sample shows how to create a subrecord from a body field and from a sublist field. The subrecord created on the body field is an Inventory Detail subrecord pertaining to the Assembly Build (parent) record.

The subrecord created on the sublist field (using nlobjRecord.createCurrentLineItemSubrecord(sublist, fldname)) is an Inventory Detail subrecord. This subrecord pertains to the first component on the Components sublist (of the Assembly Build parent record).

Notice that to set values on the sublist for the Inventory Detail (the Inventory Assignment sublist), you will use many of the same APIs you use to work with sublists on a parent record.

```
var qtytobuild = 2;
    var obj = nlapiCreateRecord('assemblybuild', {recordmode:'dynamic'});
    obj.setFieldValue('subsidiary', 3 );
    obj.setFieldValue('item', 174);
    obj.setFieldValue('quantity', qtytobuild);
    obj.setFieldValue('location', 2);

    var bodySubRecord = obj.createSubrecord('inventorydetail');
    var ctr;
    for(ctr = 1; ctr <= qtytobuild ; ctr ++)
    {
        bodySubRecord.selectNewLineItem('inventoryassignment');
        bodySubRecord.setCurrentLineItemValue('inventoryassignment', 'issueinventorynumber',
      'amsh_' + ctr);
        bodySubRecord.setCurrentLineItemValue('inventoryassignment', 'quantity', 1);
        bodySubRecord.setCurrentLineItemValue('inventoryassignment', 'binnumber', 3);
        bodySubRecord.commitLineItem('inventoryassignment');
    }
    bodySubRecord.commit();

    obj.selectLineItem('component', 1);
    obj.setCurrentLineItemValue('component', 'quantity', qtytobuild);
    var compSubRecord = obj.createCurrentLineItemSubrecord('component',
     'componentinventorydetail');

        compSubRecord.selectNewLineItem('inventoryassignment');
        compSubRecord.setCurrentLineItemValue('inventoryassignment', 'binnumber', 3);
        compSubRecord.setCurrentLineItemValue('inventoryassignment', 'quantity', 2);
        compSubRecord.commitLineItem('inventoryassignment');
        compSubRecord.commit();

    obj.commitLineItem('component');
    var id = nlapiSubmitRecord(obj);
```

Important: To save a subrecord you must call commit() on the subrecord, commitLineItem()-- if you have created a subrecord on a sublist line - - AND nlapiSubmitRecord(...) on the parent record. See Saving Subrecords Using SuiteScript for more details.

Editing an Inventory Detail Subrecord

To edit a subrecord you must first load the parent record. In the sample below the parent record (a Purchase Order) is loaded in dynamic mode. When working with subrecords, you can load parent records in dynamic mode or in standard mode. However, the subrecord itself must be scripted using "dynamic" subrecord APIs. These are the APIs that have "current" in the function or method signature.

Note: If you attempt to edit or view a subrecord that does not exist, null is returned.

```
var record2= nlapiLoadRecord('purchaseorder', id, {recordmode: 'dynamic'});
record2.selectLineItem('item', 1);
record2.setCurrentLineItemValue('item', 'quantity', 2);

var subrecord2= record2.editCurrentLineItemSubrecord('item', 'inventorydetail');
subrecord2.selectLineItem('inventoryassignment', 1);
subrecord2.setCurrentLineItemValue('inventoryassignment', 'receiptinventorynumber', 'working123');
subrecord2.selectNewLineItem('inventoryassignment');
subrecord2.setCurrentLineItemValue('inventoryassignment', 'receiptinventorynumber',
   '2ndlineinventorynumber');
subrecord2.setCurrentLineItemValue('inventoryassignment', 'quantity', '1');
subrecord2.commitLineItem('inventoryassignment');

subrecord2.commit();
```

```
record2.commitLineItem('item');

var id = nlapiSubmitRecord(record2);
```

Removing an Inventory Detail Subrecord from a Sublist Line

The following sample shows how to remove a subrecord from a sublist line with removeCurrentLineItemSubrecord.

```
var purchaseOrder = nlapiLoadRecord('purchaseorder', 1792, {recordmode: 'dynamic'});
var i=1;
var totalLine = purchaseOrder.getLineItemCount('item');

for(i; i<=totalLine; i++)
{
   purchaseOrder.selectLineItem('item', i);
   var invDetailSubrecord = purchaseOrder.viewCurrentLineItemSubrecord('item',
     'inventorydetail');
   if(invDetailSubrecord != null)
   {
               purchaseOrder.removeCurrentLineItemSubrecord('item', 'inventorydetail');
               purchaseOrder.commitLineItem('item');
   }
}
nlapiSubmitRecord(purchaseOrder);
```

Note that the nlapiRemoveSubrecord(fldname) and nlobjRecord.removeSubrecord(fldname) APIs are for removing subrecords from a body field on the parent record. Assembly Build and Assembly Unbuild are the only two parent record types that support the creation of a subrecord on a body field. Therefore, these APIs would only be useful in the context of these two record types. Be aware though that even in the UI, NetSuite business logic prevents users from removing subrecords from these parents when the subrecords are created from a body field. This means that in SuiteScript, if you attempt to call either of the removeSubrecord body field APIs, and then you call nlapiSubmitRecord on the parent, a user error will be thrown. This is in adherence to NetSuite business logic.

If you want to use either of the removeSubrecord body field APIs, it will probably be in the context of creating, and then removing your subrecord all in the same code, based on your particular use case.

Canceling an Inventory Detail Subrecord

The following sample shows how to cancel the submission of a subrecord.

```
var purchaseOrder=nlapiCreateRecord('purchaseorder', {recordmode: 'dynamic'});
purchaseOrder.setFieldValue('entity', 38);
purchaseOrder.selectNewLineItem('item');
purchaseOrder.setCurrentLineItemValue('item', 'item', 909 );
purchaseOrder.setCurrentLineItemValue('item', 'quantity', 1);

var invDetailSubrecord = purchaseOrder.createCurrentLineItemSubrecord('item', 'inventorydetail');
invDetailSubrecord.selectNewLineItem('inventoryassignment');
invDetailSubrecord.setCurrentLineItemValue('inventoryassignment', 'receiptinventorynumber', 'EIOJNF98');

invDetailSubrecord.setCurrentLineItemValue('inventoryassignment', 'quantity', 1);
invDetailSubrecord.commitLineItem('inventoryassignment');
invDetailSubrecord.cancel(); //undo this subrecord operation

purchaseOrder.commitLineItem('item'); // no subrecord is saved with this line.
var test = nlapiViewLineItemSubrecord('item', 'inventorydetail', 1);

nlapiLogExecution('DEBUG', 'subrecord should be null, and it is: ' +test);
nlapiSubmitRecord(purchaseOrder);
```

Viewing an Inventory Detail Subrecord

The following samples show how to use different versions of the subrecord "view subrecord" APIs.

Example 1

This sample shows how to return the read-only details of the subrecord that appears on the first line, being the current line, of the Items sublist.

The sample also shows how get the read-only details of the subrecord associated with the second line on the sublist.

```
var purchaseOrder=nlapiLoadRecord('purchaseorder', 1793, {recordmode: 'dynamic'});
purchaseOrder.selectLineItem('item', 1);
var invDetailSubrecord = purchaseOrder.viewCurrentLineItemSubrecord('item', 'inventorydetail');
invDetailSubrecord.selectLineItem('inventoryassignment', 1);

nlapiLogExecution('DEBUG', 'inventory number: ' +
   invDetailSubrecord.getCurrentLineItemValue('inventoryassignment',
   'receiptinventorynumber'));

var invDetailOnLine2 = purchaseOrder.viewLineItemSubrecord('item', 'inventorydetail', 2);
invDetailOnLine2.selectLineItem('inventoryassignment', 1);

nlapiLogExecution('DEBUG', 'inventory number: ' +
   invDetailOnLine2.getCurrentLineItemValue('inventoryassignment',
   'receiptinventorynumber') );
```

Example 2

This sample shows how to use the view API to access a subrecord associated with a body field.

```
var record3 = nlapiLoadRecord('assemblybuild', id, {recordmode: 'dynamic'});
var subrecord3 = record3.viewSubrecord('inventorydetail');
subrecord3.selectLineItem('inventoryassignment', 1);

nlapiLogExecution('DEBUG', 'inven: ' + subrecord3.getCurrentLineItemValue('inventoryassignment', 'issueinventorynumber'));
```

Updating Your Scripts After Enabling Advanced Bin / Numbered Inventory Management

The first script shows what a typical script might look like with the Advanced Bin / Numbered Inventory Management feature turned off (not enabled). Notice that in this script you are calling the setCurrentLineItemValue(...) API to set inventory and serial number details for the item.

When scripting with the advanced bin management feature enabled, these lines of code will break. Instead, you must create subrecords to hold all inventory detail data.

With Advanced Bin / Numbered Inventory Management OFF

```
    var obj = nlapiCreateRecord('inventoryadjustment');
    obj.setFieldValue('subsidiary', 3);  //UK
    obj.setFieldValue('account', 173 );
    obj.setFieldValue('department', 2);
    obj.setFieldValue('class', 2);
    obj.setFieldValue('memo', 'Testing 123');
    obj.setFieldValue('adjlocation' , 2);

    obj.selectNewLineItem('inventory');
    obj.setCurrentLineItemValue('inventory', 'item', 170);
```

```
    obj.setCurrentLineItemValue('inventory', 'location', 2);
    obj.setCurrentLineItemValue('inventory', 'adjustqtyby', 1);


//The next lines will be break when adv. bin management is turned on.
   obj.setCurrentLineItemValue('inventory', 'serialnumbers', 'testserial');
   obj.setCurrentLineItemValue('inventory', 'binnumbers', 'bin1');
   obj.commitLineItem('inventory');

   var id = nlapiSubmitRecord(obj);
```

With Advanced Bin / Numbered Inventory Management ON

The following shows the changes you would have to make to your script to account for the new subrecord object model.

```

    var obj = nlapiCreateRecord('inventoryadjustment', {recordmode:'dynamic'});

    obj.setFieldValue('subsidiary', 3);  //UK
    obj.setFieldValue('account', 173);
    obj.setFieldValue('department', 2);
    obj.setFieldValue('class', 2);
    obj.setFieldValue('memo', 'Testing 123');
    obj.setFieldValue('adjlocation', 2);

    obj.selectNewLineItem('inventory');
    obj.setCurrentLineItemValue('inventory', 'item', 170);
    obj.setCurrentLineItemValue('inventory', 'location', 2);
    obj.setCurrentLineItemValue('inventory', 'adjustqtyby', 1);


// the setCurrentLineItemValue API used in the first example must now be removed,
// and a subrecord must be created to hold the data you want

     var subrecord = obj.createCurrentLineItemSubrecord('inventory', 'inventorydetail');

     subrecord.selectNewLineItem('inventoryassignment');
     subrecord.setCurrentLineItemValue('inventoryassignment', 'receiptinventorynumber',
  'testserial');
     subrecord.setCurrentLineItemValue('inventoryassignment', 'quantity', 1);
     subrecord.setCurrentLineItemValue('inventoryassignment', 'binnumber', 'bin1');
     subrecord.commitLineItem('inventoryassignment');
    subrecord.commit();

    obj.commitLineItem('inventory');

    var id = nlapiSubmitRecord(obj);
```

Using SuiteScript with Address Subrecords

See the following topics for details specific to using SuiteScript with the Address Customization feature:

- SuiteScript and Address Subrecords – Overview
- Scripting the Address Subrecord
- Sample Scripts for Address Subrecords
- Scripting Billing and Shipping Addresses

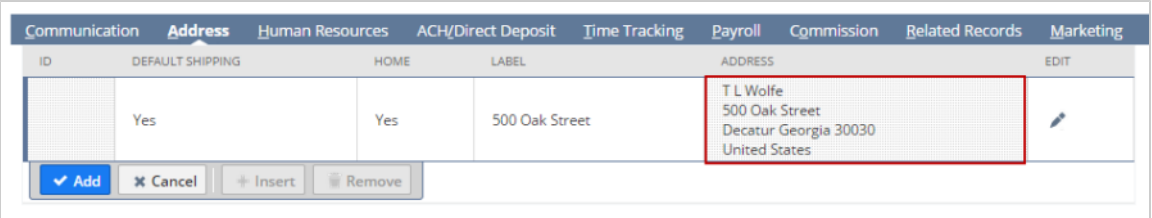
See these topics for general information about working with subrecords:

- Working with Subrecords in SuiteScript
- Guidelines for Working with Subrecords in SuiteScript

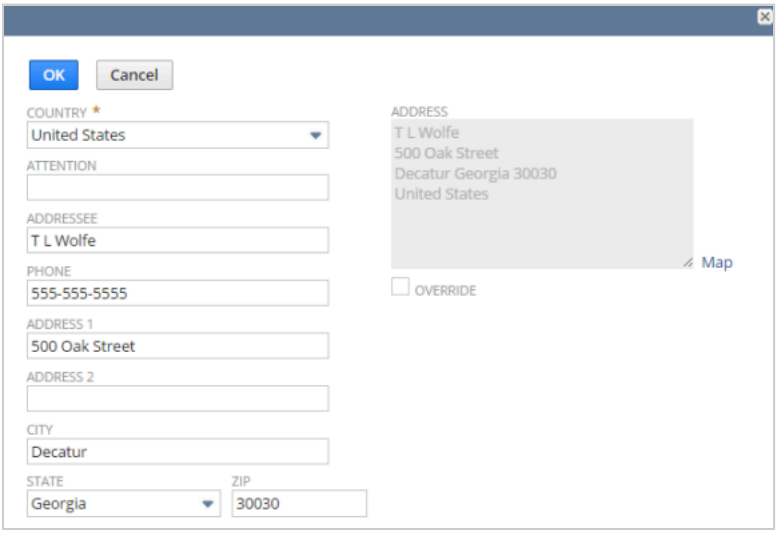
SuiteScript and Address Subrecords – Overview

The Address Customization feature consolidates individual address fields into an address subrecord. Within SuiteBuilder, you can create custom address forms (templates) for different countries. These forms determine the fields available on the address subrecord (for example, a UK address has different fields than a US address). When creating a new address, you create a new sublist item for it. You then choose the address form you want to use by selecting the country associated with it. See the help topic Working with Addresses on Transactions for additional information.

The address subrecord is accessed from a sublist field on the parent record. In the following screenshot, the sublist field that contains the address subrecord is outlined in red.



Fields on the address sublist are not part of the address subrecord. To access the address subrecord fields, you must open the subrecord. Within the UI, you access the address subrecord by clicking the pencil icon in the Edit sublist field. The address subrecord is shown below.



Scripting the Address Subrecord

Address subrecords are accessed from a sublist field on the parent record. When scripting address subrecords, use the following APIs if you are working with the parent record in a "current record" context, such as in a user event script or a client script:

- nlapiCreateCurrentLineItemSubrecord(sublist, fldname)
- nlapiEditCurrentLineItemSubrecord(sublist, fldname)
- nlapiRemoveCurrentLineItemSubrecord(sublist, fldname)
- nlapiViewCurrentLineItemSubrecord(sublist, fldname)
- nlapiViewLineItemSubrecord(sublist, fldname, linenum)

Note: nlapiCreateCurrentLineItemSubrecord and nlapiEditCurrentLineItemSubrecord are not currently supported in client scripts.

If you are loading the parent record using SuiteScript, and you want to create/access a subrecord from a sublist, use these methods on the nlobjRecord object:

- createCurrentLineItemSubrecord(sublist, fldname)
- editCurrentLineItemSubrecord(sublist, fldname)
- removeCurrentLineItemSubrecord(sublist, fldname)
- viewCurrentLineItemSubrecord(sublist, fldname)
- viewLineItemSubrecord(sublist, fldname, linenum)

Access the sublist and subrecord with the following internal ID names:

- Sublist – 'addressbook'
- Address Subrecord – 'addressbookaddress'

There are two state fields available in addresses:

- `state` is a text entry field that is not validated
- `dropdownstate` is a select field that can is used for U.S., Canadian, and Australian states or provinces

Supported Script Deployments

- Server-side scripts cannot deploy on subrecords. Server-side scripts can only access subrecords through the parent record. To accomplish this, a server-side script must be deployed on the parent record. Server-side scripts deployed on the parent record can read, create, edit, and delete address subrecords with the Subrecord APIs.
- Client scripts deployed on the parent record can read and delete address subrecords with the Subrecord APIs.
- Client scripts deployed on the address subrecord can read and edit address subrecords with the standard APIs.
- Client scripts deployed on any subrecord can run on the server side.

Client Scripts Attached to the Address Subrecord

Client scripts deployed on any subrecord can run on the server side in addition to the client side. Be aware that this is expected behavior. For logic in a client script attached to an address subrecord to execute only one time, wrap the logic in an `if` statement that immediately exits the script on the server side. For example:

```
if (typeof document!=='undefined') {
// client script logic
}
```

For more information, see the help topic Client Scripts.

Important Items to Note Before Scripting on Address Subrecords

- One of the most important things to remember when scripting address subrecords is that the country field determines which address form is used. If your script runs in dynamic mode, you must set the country field first.

- Validate Field, Field Changed, and Post Sourcing events on address fields will not fire in client scripts deployed to Entity or Item Fulfillment records or in custom code on forms for these record types. Instead, add this code to the Custom Code subtab of the address form for the record type.
- You cannot use nlapiGetLineItemValue and nlapiSetLineItemValue to access address fields in dynamic mode. Use nlapiGetCurrentLineItemValue or nlapSetCurrentLineItemValue instead
- You cannot use nlobjRecord.getFieldText, nlobjRecord.setFieldText, nlapiGetFieldText and nlapiSetFieldText when scripting subrecords. Use nlobjRecord.getFieldValue, nlobjRecord.SetFieldValue, nlapiGetFieldValue, or nlapiSetFieldValue instead.
- You cannot use nlobjRecord.getLineItemField, nlobjRecord.getField, and nlapiGetField to get address field metadata. You must access the subrecord with the subrecord APIs to get this information.
- You cannot use the subrecord APIs to access address fields on the Company Information page. Access these fields with nlapiLoadConfiguration the same way you would access non-subrecord fields. See the help topic nlapiLoadConfiguration(type) for an example.
- If you allow third-party input of address information, you need to translate third-party state input to validated NetSuite state input. For example, if the user enters a state of California, it must be converted to CA.

Sample Scripts for Address Subrecords

Creating an Address Subrecord for a New Employee

```
function createEmployee()
{
    var record = nlapiCreateRecord('employee', {recordmode: 'dynamic'});
    record.setFieldValue('companyname','Lead Company 123');
    record.setFieldValue('firstname', 'Lead Company');
    record.setFieldValue('lastname', '123');
    record.setFieldValue('subsidiary','1'); //PARENT COMPANY

    //Add first line to sublist
    record.selectNewLineItem('addressbook');
    record.setCurrentLineItemValue('addressbook', 'defaultshipping', 'T');  //This field is not a subrecord field.
    record.setCurrentLineItemValue('addressbook', 'defaultbilling', 'T');   //This field is not a subrecord field.
    record.setCurrentLineItemValue('addressbook', 'label', 'First Address Label');  //This field is not a subrecord field.
    record.setCurrentLineItemValue('addressbook', 'isresidential', 'F');    //This field is not a subrecord field.

    //create address subrecord
    var subrecord = record.createCurrentLineItemSubrecord('addressbook', 'addressbookaddress');

    //set subrecord fields
    subrecord.setFieldValue('country', 'US'); //Country must be set before setting the other address fields
    subrecord.setFieldValue('attention', 'John Taylor');
    subrecord.setFieldValue('addressee', 'NetSuite Inc.');
    subrecord.setFieldValue('addrphone', '(123)456-7890');
    subrecord.setFieldValue('addr1', '2955 Campus Drive');
    subrecord.setFieldValue('addr2', 'Suite - 100');
    subrecord.setFieldValue('city', 'San Mateo');
    subrecord.setFieldValue('dropdownstate', 'CA');
        // if the address is not in U.S., Canada, or Australia, use
        // state instead of dropdownstate. For example,
        // subrecord.setFieldValue('state', 'BY');
        // for Bavaria, Germany
    subrecord.setFieldValue('zip', '94403');

    //commit subrecord and line item
    subrecord.commit();
    record.commitLineItem('addressbook');

    //submit record
    var x = nlapiSubmitRecord(record);
}
```

Accessing Address Subrecord fields on an Existing Customer Record

```
var record = nlapiLoadRecord('customer', 143,{recordmode: 'dynamic'});

record.selectLineItem('addressbook', 2);

var subrecord = record.viewCurrentLineItemSubrecord('addressbook', 'addressbookaddress');

var country = subrecord.getFieldValue('country');
var attention = subrecord.getFieldValue('attention');
```

Editing an Address Subrecord on an Existing Customer Record

```
var record = nlapiLoadRecord('customer', 143,{recordmode: 'dynamic'});

record.selectLineItem('addressbook', 2);

var subrecord = record.editCurrentLineItemSubrecord('addressbook', 'addressbookaddress');

subrecord.setFieldValue('attention', 'Accounts Payable');

subrecord.commit();
record.commitLineItem('addressbook');

var x = nlapiSubmitRecord(record);
```

Removing an Address Subrecord on an Employee Record

```
var record = nlapiLoadRecord('employee', 234, {recordmode: 'dynamic'});

record.selectLineItem('addressbook', 3);
record.removeCurrentLineItemSubrecord('addressbook', 'addressbookaddress');
record.commitLineItem('addressbook');

var x = nlapiSubmitRecord(record);
```

Scripting Billing and Shipping Addresses

You can use SuiteScript to create a new address subrecord for an entity. Subsequently, you can use that address as the billing or shipping address on transactions.

When working with a transaction in the UI, you can select one of the customer's existing addresses by using either the **Shipping** or **Billing** subtab. Each subtab includes a select field from which you can select an existing address or create a new one.

Compared with other addresses, the primary difference with transaction billing and shipping addresses is that they can be sourced from other records. For this reason, when creating a transaction, you can select an existing address for shipping or billing by referring to the address's internal ID, which is stored with the entity record. The following sample scripts illustrate this approach.

Another approach is to create a new address that will be associated only with the transaction (and not sourced from or saved to the customer record). That technique is not covered in this topic.

Note: For additional information about scripting subrecords from a body field, see Creating and Accessing Subrecords from a Body Field. For additional script examples of subrecords accessed from a body field, see Using SuiteScript with Advanced Bin / Numbered Inventory Management.

Creating a New Address

The following user event script creates a new address subrecord on an entity.

```
function addAddress () {

    // Identify a customer.

    var customerId = 1163;

    // Load the customer record.

    var record = nlapiLoadRecord('customer', customerId, {recordmode: 'dynamic'});

    // Create a new address for the customer

    var addrSubrecord = record.createCurrentLineItemSubrecord('addressbook', 'addressbookaddress');

    // Set the appropriate address subrecord fields.

    addrSubrecord.setFieldValue('country', 'US');
    addrSubrecord.setFieldValue('isresidential', 'F');
    addrSubrecord.setFieldValue('attention', 'Billing Address');
    addrSubrecord.setFieldValue('addressee', 'NetSuite Inc.');
    addrSubrecord.setFieldValue('addrphone', '(123)456-7890');
    addrSubrecord.setFieldValue('addr1', '2955 Campus Drive');
    addrSubrecord.setFieldValue('addr2', 'Suite - 100');
    addrSubrecord.setFieldValue('city', 'San Mateo');
    addrSubrecord.setFieldValue('state', 'CA');
    addrSubrecord.setFieldValue('zip', '94403');

    // Commit the new address subrecord.

    addrSubrecord.commit();
    record.commitLineItem('addressbook');

    // Update the customer record.

    nlapiSubmitRecord(record);


}
```

Creating a New Custom Shipping Address

The following user event script searches for a specific address subrecord on an entity and then assigns that address to a sales order record.

```
function createSalesOrder() {

    var customerid = 1163;
    var itemid = 100;
    var addressInternalId = '';

    // Load the customer record.

    var readrecord = nlapiLoadRecord('customer', customerid);


    // Get the internal ID of the address subrecord you want to use.

    for(var x = 1; x <= readrecord.getLineItemCount('addressbook'); x++) {
             if (readrecord.getLineItemValue('addressbook', 'addressee', x) === 'NetSuite Inc.') {
                addressInternalId = readrecord.getLineItemValue('addressbook', 'internalid', x)
                break;
             }
    }

```

```
    // Create a new sales order record.

    var record = nlapiCreateRecord('salesorder');


    // Set the customer (entity) field.

    record.setFieldValue('entity', customerid);


    // For debugging purposes, set the memo field equal to the internal Id of the address.

    record.setFieldValue('memo', addressInternalId);


    // Set the billaddresslist value to that of the desired address.

    record.setFieldValue('billaddresslist', addressInternalId);


    // Create a new item for the sales order.

    record.selectNewLineItem('item');


    // Set the appropriate fields for the item

    record.setCurrentLineItemValue('item', 'item', itemid);
    record.setCurrentLineItemValue('item', 'quantity', 1);
    record.setCurrentLineItemValue('item', 'location', 6);
    record.setCurrentLineItemValue('item', 'amount', '190.89');


    // Commit the new item for the sales order.

    record.commitLineItem('item');


    // Update the sales order.

    nlapiSubmitRecord(record);

}
```

Legacy fields like `shipcity` are only available for backward compatibility with server-side scripts that were created before address customization. These legacy fields should not be used if possible because they have limitations.

Note: Your solution might not work as expected if you use legacy fields with the SuiteTax feature, for example, onChange scripts on address fields are triggered when the record is submitted.

Use code similar to the following to work with subrecords in server-side scripts:

```
var address = rec.createSubrecord('shippingaddress');
address.setFieldValue('city');
rec.commitSubrecord('shippingaddress');
```

Using Specific Fields from an Address Subrecord on a Transaction Record

The following user event script searches for a specific address subrecord on an entity and then assigns that address to a sales order record.

```
function createSalesOrder()
```

```
{
    var customerid = 1163;
    var itemid = 100;
    var addressInternalId = '';

    // Load the customer record.
    var readrecord = nlapiLoadRecord('customer', customerid);

    // Get the internal ID of the address subrecord you want to use.
    for(var x = 1; x <= readrecord.getLineItemCount('addressbook'); x++) {
        if (readrecord.getLineItemValue('addressbook', 'addressee', x) === 'First Software') {
            addressInternalId = readrecord.getLineItemValue('addressbook', 'internalid', x);
                break;
          }
    }

    // Create a new sales order record.
    var record = nlapiCreateRecord('salesorder');

    // Set the customer (entity) field.
    record.setFieldValue('entity', customerid);

    // For debugging, set the memo field equal to the internal Id of the address.
    record.setFieldValue('memo', addressInternalId);

    // Create a new item for the sales order.
    record.selectNewLineItem('item');

    // Set the appropriate fields for the item
    record.setCurrentLineItemValue('item', 'item', itemid);
    record.setCurrentLineItemValue('item', 'quantity', 1);
    record.setCurrentLineItemValue('item', 'amount', '190.89');

    // Commit the new item for the sales order.
    record.commitLineItem('item');

    // Update the sales order.
    nlapiSubmitRecord(record);
}
```

Defining Address by Item Line

To define a custom address by item line, you enter custom code similar to the following.

```
var tran = nlapiCreateRecord('salesorder', {recordmode: 'dynamic'});
tran.setFieldValue('entity', '220');
tran.setFieldValue('ismultishipto', 'T');

tran.selectLineItem('item', 1);
tran.setCurrentLineItemValue('item', 'item', '144');
tran.setCurrentLineItemValue('item', 'quantity', '2');

//Custom address definition
var subrecord = tran.createCurrentLineItemSubrecord('item', 'shippingaddress');
subrecord.setFieldValue('country', 'US');
subrecord.setFieldValue('addr1', 'Test 1');

//Custom address label is write-only. Address will be stored with this label, but when address is loaded again, it won't be available for read using SuiteScript (it will be visible in UI as usual). This is because label is not a part of address and so it won't be loaded on address subrecord load.
//When custom address is edited, new custom address is always created. When custom address is edited after transaction was saved and loaded again, label will be null as it won't be copied over to new custom address. In such case, user has to define address label again.
subrecord.setFieldValue('label', 'My label');
subrecord.setFieldValue('zip', '94403');
subrecord.setFieldValue('state', 'CA');
subrecord.commit();
tran.commitLineItem('item');

tran.selectNewLineItem('item');
tran.setCurrentLineItemValue('item', 'item', '144');
tran.setCurrentLineItemValue('item', 'quantity', '3');

```

```
//User can use new custom address on multiple item lines by setting its address book key to item line 'shipaddress' select. In this case we set custom address used on item line 1
tran.setCurrentLineItemValue('item', 'shipaddress', tran.getLineItemValue('item', 'shipaddress', 1));
tran.commitLineItem('item');

nlapiSubmitRecord(tran);
```

Working with Fields

The following topics are covered in this section. If you are new to SuiteScript, they should be read in order:

- Working with Fields Overview
- Referencing Fields in SuiteScript
- Working with Custom Fields in SuiteScript

Working with Fields Overview

The SuiteScript API includes several Field APIs you can use to set and get values for built-in NetSuite **standard** fields, as well as for **custom** fields. Standard fields are those that are included with NetSuite. Custom fields are those that have been created by NetSuite users to customize their accounts. Custom fields are created using SuiteBuilder point-and-click customization tools.

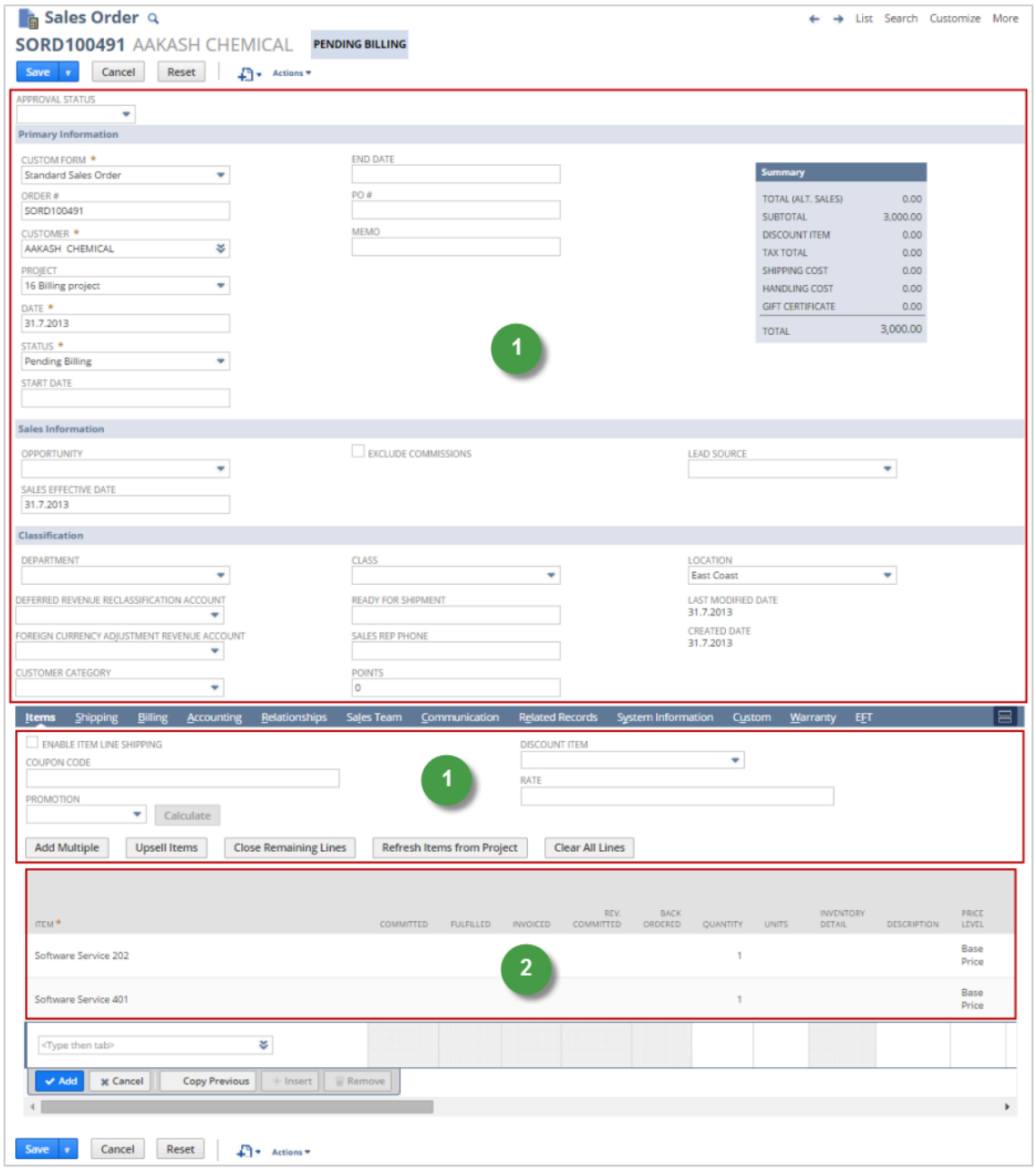
Note: For information about working with nlobjField objects that you can add dynamically to NetSuite records at runtime, see the help topic nlobjField in the NetSuite Help Center. These are the only type of fields you can programmatically add to a record. There are no SuiteScript APIs available for creating custom fields that are akin to the kinds of custom field created using SuiteBuilder point-and-click functionality.

Important: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). The NetSuite UI should only be accessed using SuiteScript APIs.

The following figure shows a combination of body and sublist fields. The body sections of a record include the top (header) portion and non-sublist fields that sometimes appear on the top area of a subtab. Body fields that appear under a subtab should not be confused with sublist fields. Each line on a sublist is referred to as a line item; the fields on each line item are sublist fields. Sublist fields are accessed using Sublist APIs.

On the figure below:

1. **Body fields** - can be a mix of standard and custom fields.
2. **Sublist fields** - fields on a sublist. See Working with Subtabs and Sublists for more information.



Referencing Fields in SuiteScript

Many SuiteScript APIs allow you to get, set, or search for the value of a particular field. Whether you are referencing a standard field or a custom field, when you reference the field in SuiteScript, you will use the field's internal ID. To obtain field internal IDs, see How do I find a field's internal ID? in the NetSuite Help Center.

Important: Be aware that not every field that appears in your NetSuite account officially supports SuiteScript. To make sure you write scripts that include only supported, officially tested NetSuite fields, refer to the SuiteScript Records Browser to verify a field's official support. See the help topic Working with the SuiteScript Records Browser for more details.

Getting Field Values in SuiteScript

If you are using SuiteScript to process record data in **standard** mode (as opposed to **dynamic** mode), be aware of the following when using "getter" APIs to get the value of a field:

Note: If you are not familiar with standard mode and dynamic mode scripting, see SuiteScript 1.0 Working with Records in the NetSuite Help Center.

To check if a field has a non-empty value, you should write code which checks for null and empty when using any of the following APIs:

- nlapiGetFieldValue(fldnam)
- nlapiGetLineItemValue(type, fldnam, linenum)
- nlobjRecord.getFieldValue(name)
- nlobjRecord.getLineItemValue(group, name, linenum)

For guidance and examples of syntax for scripting with fields, click on one of the links above to view details for each specific API.

Important: Note that this inconsistency in field return values does NOT exist when scripting records in **dynamic** mode.

The following snippets provide examples of how you might want to write your code to catch any null vs. empty string return value inconsistencies:

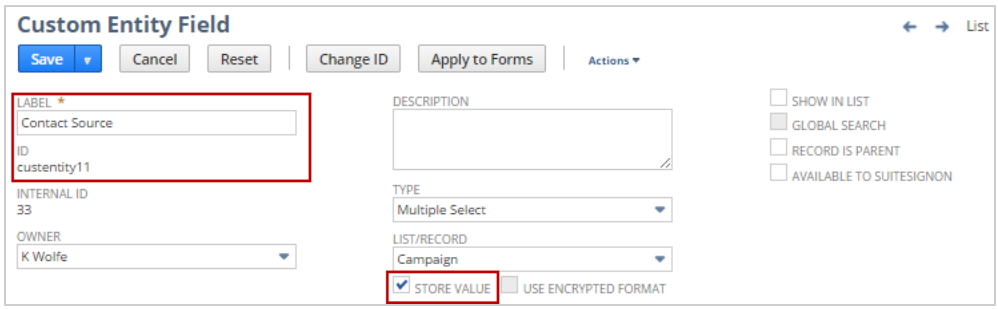
```
if (value) {
    // Handle the case where value is not empty
}
```

```
if (!value) {
    // Handle the case where value is empty (or null)
}
```

Working with Custom Fields in SuiteScript

You can use SuiteScript APIs to get, set, and search the values of custom fields that have been created using SuiteBuilder. Note, however, you can only set the value of custom fields that have a **stored value**. This follows the behavior of the UI.

The following figure shows a custom entity field. The field's UI label is Contact Source and its internal ID is custentity11. In this figure, the **Store Value** box is selected, which means that you can use SuiteScript to get and set the value of this custom entity field.



When a custom field does not contain a stored value (the **Store Value** box is not selected), you *can* reference this field in a SuiteScript search to return the current value of the field. However, non-stored custom fields are considered to have dynamic values, so in a search, the value of a non-stored custom field might be 10 one day and 12 the next day when the same search is executed.

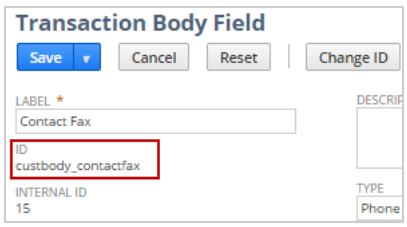
Note: If you are not familiar with creating custom fields in NetSuite, see the help topic Custom Fields in the NetSuite Help Center.

Providing Internal IDs for Custom Fields

If you are using SuiteBuilder to create a custom field, and you plan to reference the field in your scripts, you should create an internal ID that includes an underscore (_) after the custom field's prefix. You should then add a meaningful name after the underscore. This will enhance readability in your SuiteScript code.

For example, if you are using SuiteBuilder to create a custom transaction body field with the UI label Contact Fax, the field's internal ID should be something equivalent to **_contactfax**. Note that you do not need to write the custom field's prefix in the ID field (see figure below). After the custom field definition is saved, the prefix for that custom field type is automatically added to the ID. When the custom transaction body field (below) is saved, its internal ID will appear as **custbody_contactfax**. This is the ID you will reference in your scripts.

Important: Do not use **_send** when you specify an internal ID for a custom field. The suffix **_send** is reserved and may cause an issue with successful deployment of your script.



Understanding Custom Field Prefixes

As a reference, the following table provides the prefixes for each custom field type. You do not need to type these prefixes when you assign an internal ID to a custom field. This table is provided only for convenience to SuiteScript developers who may be working with different custom field types and are not sure how to identify the field type using the prefix.

Custom field type	Custom field prefix
Entity field	custentity

Custom field type	Custom field prefix
Item field	custitem
CRM field	custevent
Transaction body field	custbody
Transaction column field	custcol
Transaction item options	custcol
Item number fields	custitemnumber
Other custom fields	custrecord

Working with Subtabs and Sublists

Subtabs and Sublists Overview

When using SuiteScript on subtabs and sublists, you should be aware of the following:

1. The distinction between subtabs and sublists (see Subtabs and Sublists - What's the Difference?)
2. Sublist types (see Sublist Types)
3. Adding subtabs with SuiteScript (Adding Subtabs with SuiteScript)
4. Adding sublists with SuiteScript (Adding Sublists with SuiteScript)
5. Manipulating sublist with SuiteScript (Working with Sublist Line Items)
6. Sublist scripting when a record is in dynamic mode (Working with Sublists in Dynamic Mode and Client SuiteScript)

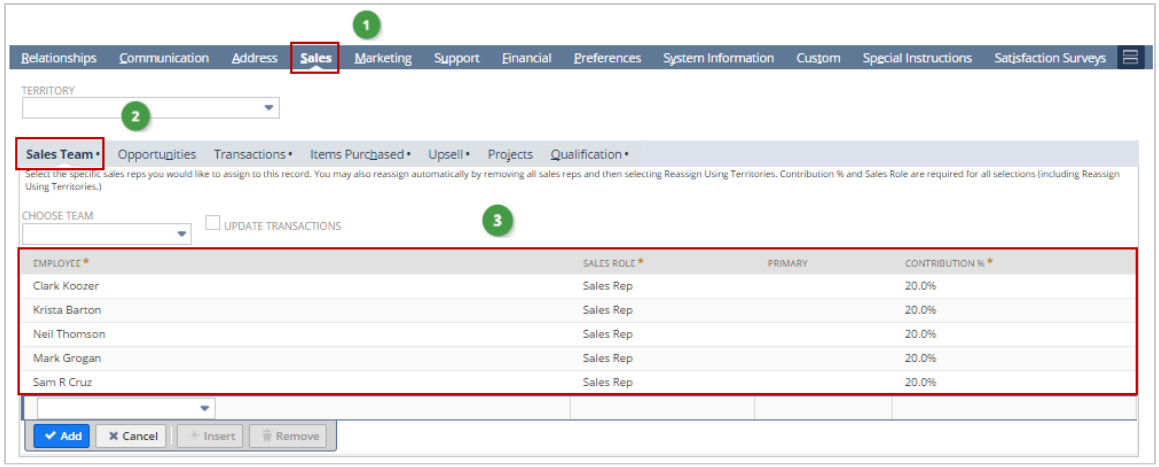
Important: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). The NetSuite UI should only be accessed using SuiteScript APIs.

Note: For a list of all sublists that support SuiteScript, see the SuiteScript Records Browser. To see all sublist-related APIs, see the help topic Sublist APIs.

Subtabs and Sublists - What's the Difference?

Subtabs and sublists both look like tabs in the UI (see figure). However, functionally they serve different purposes. See these sections to learn about the differences between subtabs and sublists:

- What is a Subtab?
- What is a Sublist?

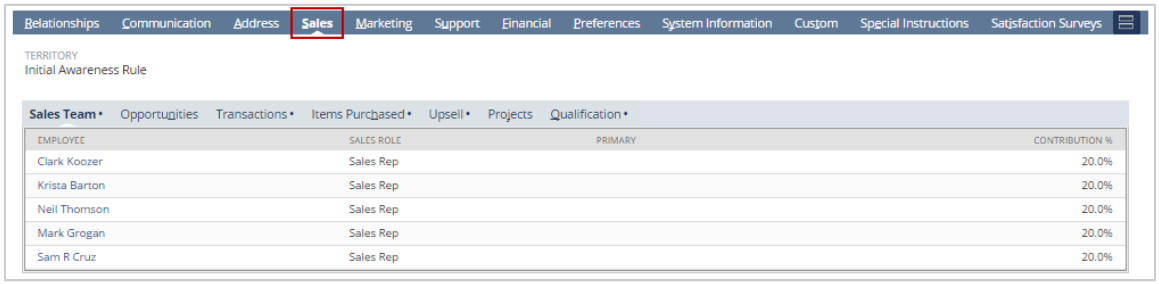


1	Parent Subtab
2	Child Subtab
3	Sublist

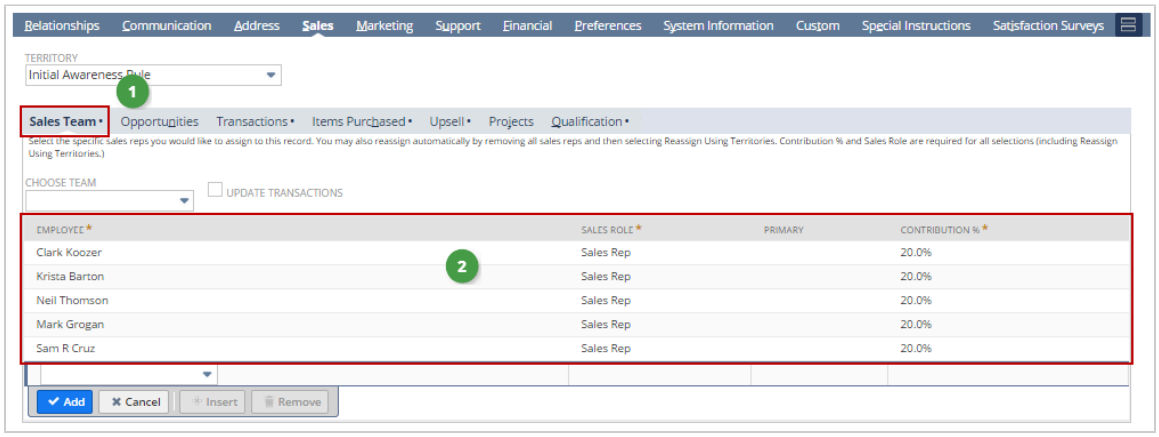
What is a Subtab?

Subtabs contain body fields, other subtabs, and sublists. Unlike sublists, subtabs do not contain references to other records. Subtabs are used mainly for organizational purposes.

The figure below shows the Sales subtab on a Customer record. Notice that the Sales tab contains body fields that hold data specific to the Customer. The primary purpose of the Sales subtab is to organize all of the sales-related sublists (Sales Team, Opportunities, Transactions, and so on).



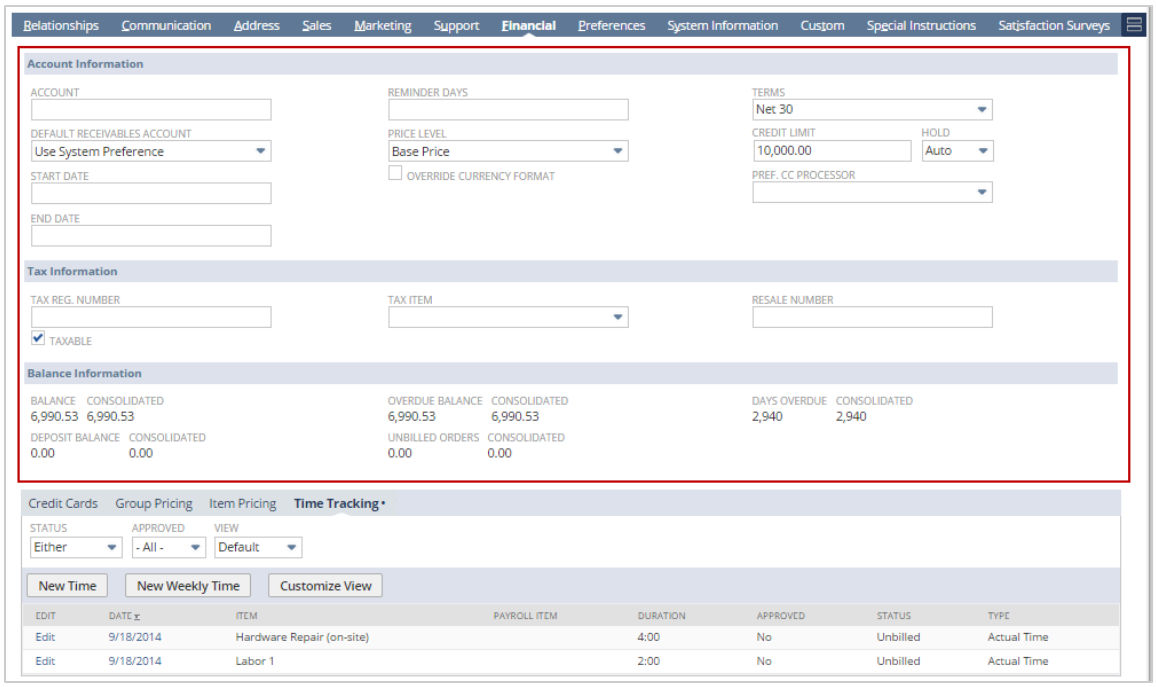
To compare what you see on the Sales subtab, the Sales Team sublist contains data that link to other records—in this case, the employee records for the sales people associated with this customer (see figure).



1	Child Subtab
2	Sublist

The next figure shows the Financial subtab, also on the Customer record. Notice that the information about this subtab is additional field-level information related to this particular customer. None of the information applies to or references data that exists on another record.

In SuiteScript you can access fields that appear on a subtab using Field APIs. Field APIs are also used on regular body fields that appear on the top portion of records.



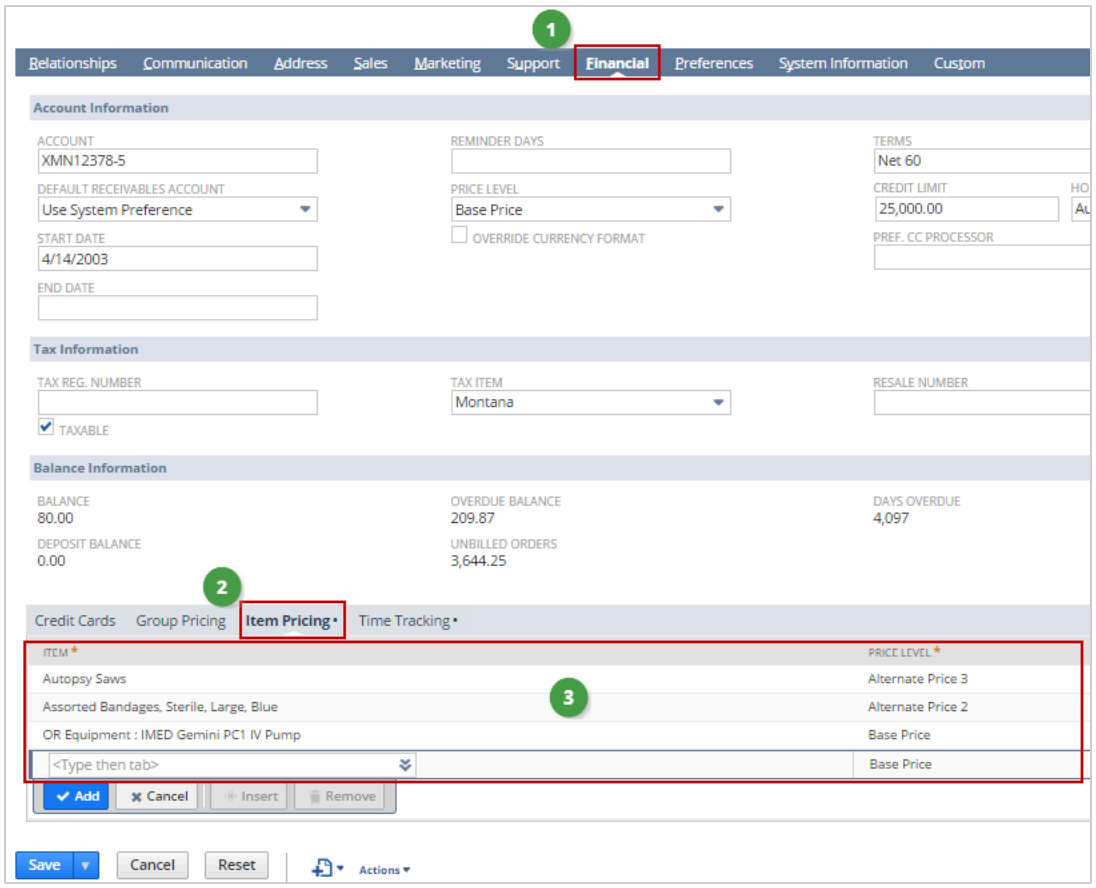
What is a Sublist?

Sublists contain a list of references to other records. Note that the list of record references are referred to as **line items**. Within NetSuite there are four types of sublists: editor, inline editor, list, and static list (see Sublist Types for details on each type).

Important: Static list sublists do not support SuiteScript. For a list of all editor, inline editor, and list sublists that support SuiteScript.

The following figure shows the Pricing Sublist / Pricing Matrix on the Customer record. This is an **inline editor** sublist that appears on a subtab, in this case the Financial subtab. Whereas the field-level data captured on the Financial subtab applies specifically to this customer, the data on the Item Pricing sublist references data contained on other records.

In the UI, you can add/insert/remove lines items to this sublist using the Add, Insert, and Remove buttons. In SuiteScript, you can perform the same actions using Sublist APIs such as nlapiInsertLineItem(type, line) and nlapiRemoveLineItem(type, line).



1	Parent Subtab
2	Child Subtab
3	Sublist

Sublist Types

There are four types of sublists in NetSuite:

- Editor Sublists
- Inline Editor Sublists
- List Sublists
- Static List Sublists

Important: Static list sublists do not support SuiteScript. Scripts written against static list sublists will either not run or will return a system error. All other sublist types support both client and server SuiteScript.

Note: If you are building your own custom form and are adding a sublist object to that form through nlobjForm.addSubList(name, type, label, tab), you can set the sublist **type** to any of the four sublist types. You can then write scripts against your custom sublist. Note that sorting (in the UI) is not supported on static sublists created using the addSubList(...) method if the row count exceeds 25.

Editor Sublists

The editor sublist allows users to insert/edit/remove lines dynamically prior to submitting the form. On an editor sublist, editing sublists lines (referred to as line items) is done in fields directly above the line items. In the UI, changes you make when you add/edit/remove lines are not committed to the database until you save the entire record. Similarly, in SuiteScript add/edit/remove functions provided in Sublist APIs are not persisted in the NetSuite database until the change is committed to the NetSuite database.

When writing client scripts, you must call nlapiCommitLineItem(type) after each sublist line change. Otherwise your changes will not be committed to NetSuite.

When writing server scripts, you must call nlobjRecord.commitLineItem(group, ignoreRecalc) to commit sublist updates. Note that you must do this in addition to calling nlapiSubmitRecord(record, doSourcing, ignoreMandatoryFields), which commits the entire record object to the database.

Note: In SuiteScript, the first sublist line item is numbered 1, not 0.

Inline Editor Sublists

Inline editor sublists are similar to Editor Sublists in these ways:

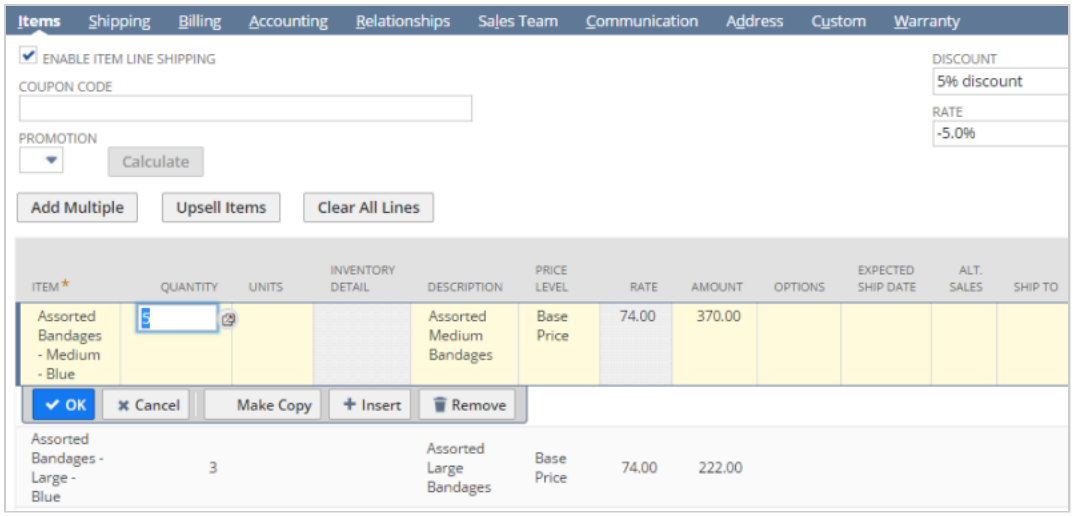
- you can add/edit/remove lines dynamically prior to submitting the form
- you can add/edit/remove lines using the UI or SuiteScript
- when writing client scripts, you must call nlapiCommitLineItem(type) after each sublist line change. Otherwise your changes will not be committed to NetSuite.
- When writing server scripts, you must call nlobjRecord.commitLineItem(group, ignoreRecalc) to commit sublist updates. Note that you must do this in addition to calling nlapiSubmitRecord(record, doSourcing, ignoreMandatoryFields), which commits the entire record object to the database.

The **only** difference between an inline editor sublist and an editor sublist is UI appearance. Inline editor sublists do not contain a line item edit area directly above the line items. The line items on an inline editor sublist are edited "inline" directly on the lines on which they appear.

The following figure shows the Items sublist on the Estimate record. The field-level data that appears directly above the line items are not used for adding, editing, or removing line items that appear below it.

In SuiteScript, fields above the line items are accessed using Field APIs. Sublist line items are accessed using Sublist APIs.

Note: In SuiteScript, the first sublist line item is numbered 1, not 0.



List Sublists

Unlike Editor Sublists and Inline Editor Sublists, **list** sublists are not dynamic. The number of line items are fixed and cannot be removed or added on-the-fly though UI customziation or through SuiteScript.

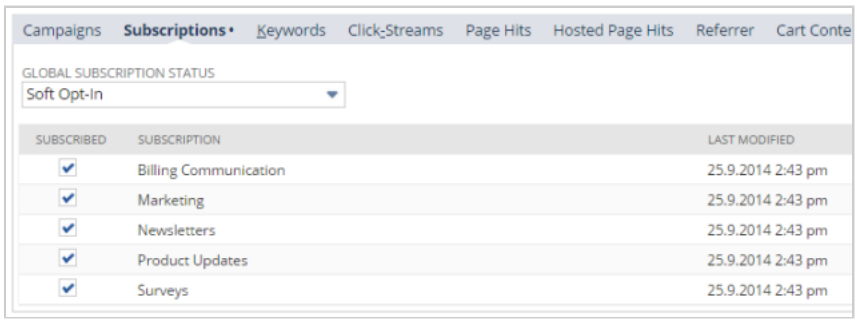
Changes you make to existing line items on list sublists are submitted along with the main record and do not take effect until after the record has been saved. Note that even though you cannot add or remove lines on a list sublist, you can use the UI to change values or SuiteScript to get/set values on lines that currently exist.

In SuiteScript you would not use Sublist APIs such as nlapiSelectNewLineItem(type), nlapiInsertLineItem(type, line), or nlapiRemoveLineItem(type, line) to add or remove line items. Neither will you use the nlapiCommitLineItem(type) or nlapiRefreshLineItems(type) APIs in the context of a list sublist.

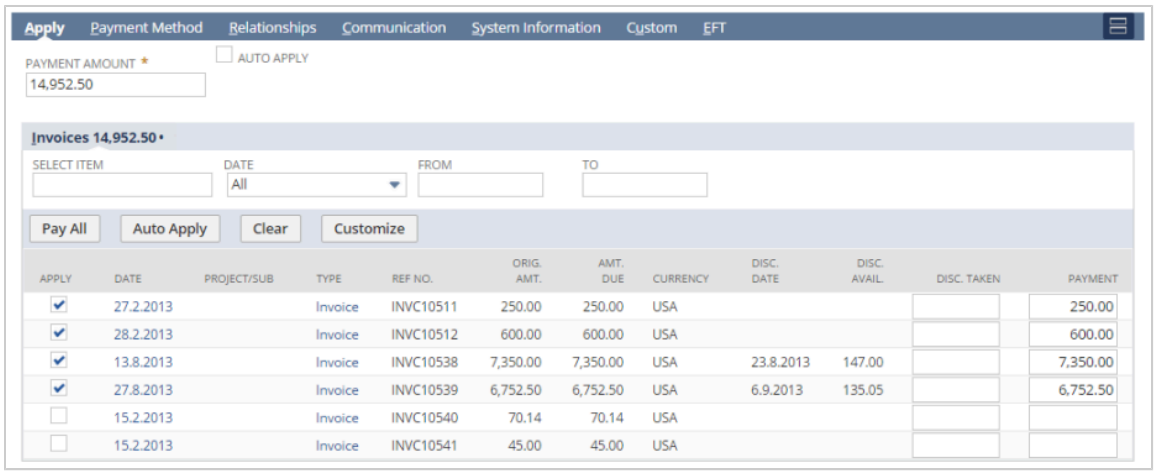
Also note that in SuiteScript, client lineInit and validateLine functions will not execute, since they have no context in a list sublist. (For information about client event functions, see Client Event Types.)

The following figure shows the Subscriptions child sublist on the Customer record. Although you cannot add/remove lines, you can edit the lines that are there (in this case, you can select or de-select the check boxes).

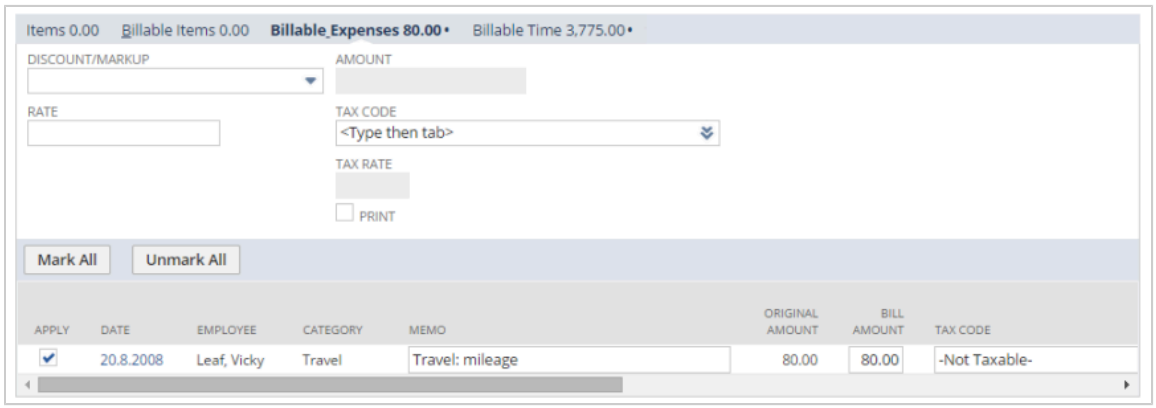
Note: In SuiteScript, the first sublist line item is numbered 1, not 0.



The next figure shows the Apply sublist on the Accept Customer Payments record. Similar to the Subscriptions sublist, you can manipulate the line items that appear, but you cannot dynamically add or remove additional lines.



The last figure provides another example of a list sublist—the Billable Expenses sublist on the Invoice record. Again, you can only manipulate the line item data provided. You cannot dynamically add or remove items. Any changes you make to the data on this sublist will not be committed to the database until you call nlapiSubmitRecord(record, doSourcing, ignoreMandatoryFields) in your script.

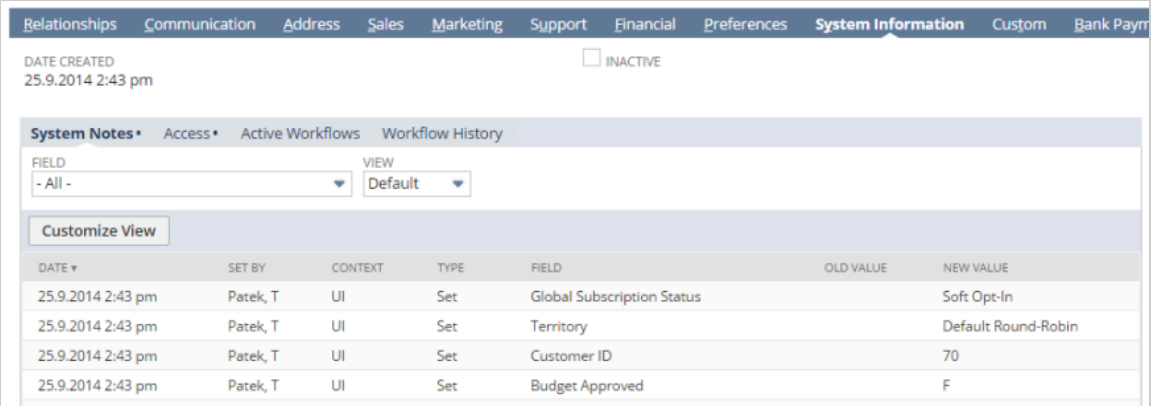


Static List Sublists

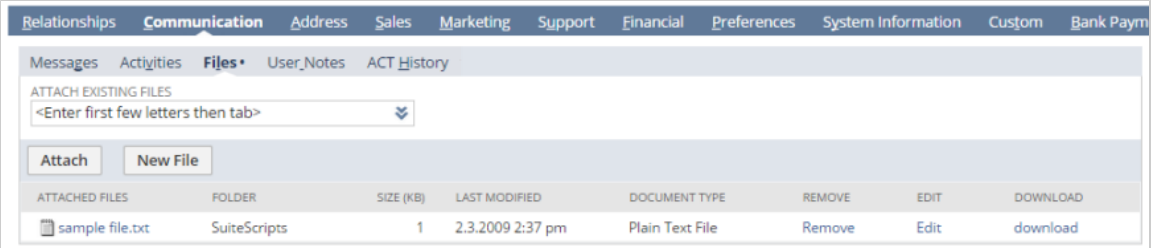
Important: SuiteScript is not currently supported on static list sublists.

Static list sublists, also referred to as read-only sublists, contain static data. These sublists are typically used for displaying associated records/data rather than child records/data. Technically, this means that the data on a static list sublist is not a part of the record (and therefore not accessible to SuiteScript), and is not submitted with the record when the record is saved.

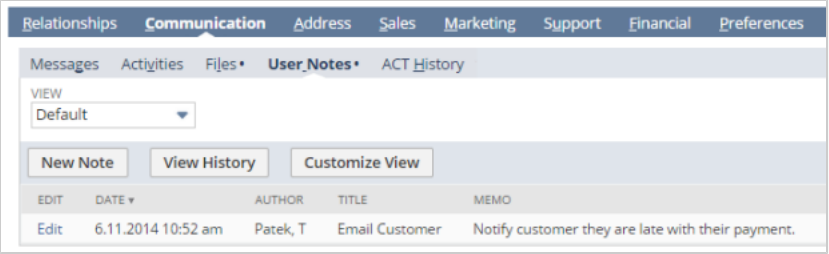
The following figure shows the System Notes sublist, which is accessed through the System Information subtab on many records. Note that all data in the System Notes sublist is read-only and is not even settable through the UI.



The next figure shows the Files sublist, which is accessed from the Communications subtab on many records. In this case you can attach/detach a file to this sublist, but the file is maintained entirely as a separate document. The data in this document (in this case a .txt file), is not considered to be part of Customer record, which can be manipulated through the UI or through SuiteScript.



The last figure shows the User Notes sublist, also accessed through the Communication subtab. Although you can add a new Note to this sublist, the data you define on the Note record is not available to this Customer record. Therefore, the User Notes sublist is considered to hold static/read-only data.



Note: In some cases you *can* use search joins in SuiteScript to search the data on a static list sublist (for example, data related to notes, contacts, messages, or files that appear on a particular record). In the previous example, you could use the *file* search join to search for all files associated with this particular Customer record.

Adding Subtabs with SuiteScript

You can add subtabs to custom forms through UI point-and-click customization and through SuiteScript. In scripting, you must use either of the following two nlobjForm methods, depending on your use case:

- addTab(name, label) — to add a top-level tab
- addSubTab(name, label, tab) — to create a nested subtab

> **Important:** You must define **two** subtabs for subtab UI labels to appear. If you define only one subtab in your script, the UI label you provide for the subtab doesn't appear in the UI.

Both methods return an nlobjTab object, through which you can further define the properties of your tab.

> **Note:** To add subtabs using UI customization, in the NetSuite Help Center, see the help topics Adding Subtabs to a Custom Record and Configuring Subtabs for Custom Entry and Transaction Forms.

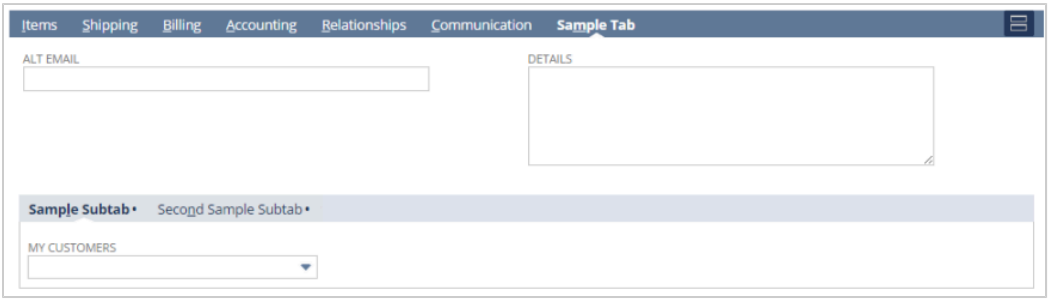
Example

The following example shows how to use SuiteScript to add subtabs to a custom NetSuite form. This script is a beforeLoad user event script that is deployed to the Sales Order. Note that if you add only one subtab, the UI label you define for the subtab **will not appear** in the UI. You must define two subtabs for subtab UI labels to appear.

When you are adding UI Objects to an existing form, be sure to prefix the internal IDs for all elements with **custpage**, for example 'custpage_sample_tab' and 'custpage_field_email' (see sample). In the sample below, the nlobjTab and nlobjField UI objects are being added to a custom transaction form on a Sales Order. (See the help topic Custom Transaction Forms in the NetSuite Help Center if you are not familiar with this form type.)

Also note that element internal IDs must be in all lowercase.

```
//Define the user event beforeLoad function
function tabsToSalesOrder(type, form)
{
//Define the values of the beforeLoad type argument
if (type == 'create')
    {
    //Add a new tab to the form
    var sampleTab = form.addTab('custpage_sample_tab', 'Sample Tab');

    //Add a field to the new tab
    var newFieldEmail = form.addField('custpage_field_email', 'email', 'Alt Email', null,
      'custpage_sample_tab');

    //Add a second field to the new tab
    var newFieldText = form.addField('custpage_field_text', 'textarea', 'Details', null,
      'custpage_sample_tab');

    //Add a subtab to the first tab
    var sampleSubTab = form.addSubTab('custpage_sample_subtab', 'Sample Subtab',
      'custpage_sample_tab');

    //Add a select field to the subtab
    var newSubField = form.addField('custpage_sample_field', 'select', 'My Customers', 'customer',
      'custpage_sample_subtab');

    //Add a second subtab to the first tab
    var sampleSubTab = form.addSubTab('custpage_sample_subtab2', 'Second Sample Subtab',
    'custpage_sample_tab');

    //Add a field to the second subtab
    var newSubField = form.addField('custpage_sample_field2', 'select', 'My Employees', 'employee',
      'custpage_sample_subtab2');
    }
}
```



Adding Sublists with SuiteScript

You can add sublists to custom forms through UI point-and-click customization and through SuiteScript. In scripting, you must use the nlobjForm.addSubList(name, type, label, tab) method to add a sublist. This method returns an nlobjSubList object, through which you can further define the properties of your sublist.

Important: The internal ID for all custom sublists, subtabs, and fields must be prefixed with custpage. The rest of the ID name must be in lowercase.

When adding a sublist through scripting you must define:

1. The custom sublist internal ID.

   Example : ' **custpage** _contacts'
2. The sublist type you are defining.

   Example : 'editor', 'inlineeditor', 'list', or 'staticlist'
3. The UI label for the sublist.

   Example : 'Custom Contacts'
4. The subtab on which the sublist will appear.

   Example : 'general' or 'custpage_mynewsubtab'

Important: To add sublists through point-and-click customization, see the help topic Custom Sublists in the NetSuite Help Center. When adding a sublist through UI customization, you are essentially adding the data from a saved search, the results of which are only associated with the record. This is the equivalent of a static list sublist. The results are not considered to be part of the record.

Example 1

This sample shows how to create a custom sublist and run a search every time the form is loaded, edited, or viewed. This is a beforeLoad user event script.

```
function beforeLoadSublist(type, form)
{
    if (type=='edit' || 'view')
    {
        //add a sublist to the form. Specify an internal ID for the sublist,
```

```
        //a sublist type, sublist UI label, and the tab the sublist will appear on
        var contacts = form.addSubList('custpage_contacts', 'staticlist', 'Custom Contacts', 'general');

        //add fields to the sublist
        contacts.addField('entityid', 'text', 'Name');
        contacts.addField('phone', 'phone', 'Phone');
        contacts.addField('email', 'email', 'Email');

        // perform a Contact record search. Set search filters and return columns for
        // the Contact search
        var contactdata = nlapiSearchRecord('contact', null, new
                          nlobjSearchFilter('company', null, 'anyOf', nlapiGetRecordId()),
                            [new nlobjSearchColumn('entityid'), new nlobjSearchColumn('phone'),
                            new nlobjSearchColumn('email')])

        // display the search results on the Custom Contact sublist
        contacts.setLineItemValues(contactdata)
}
}
```

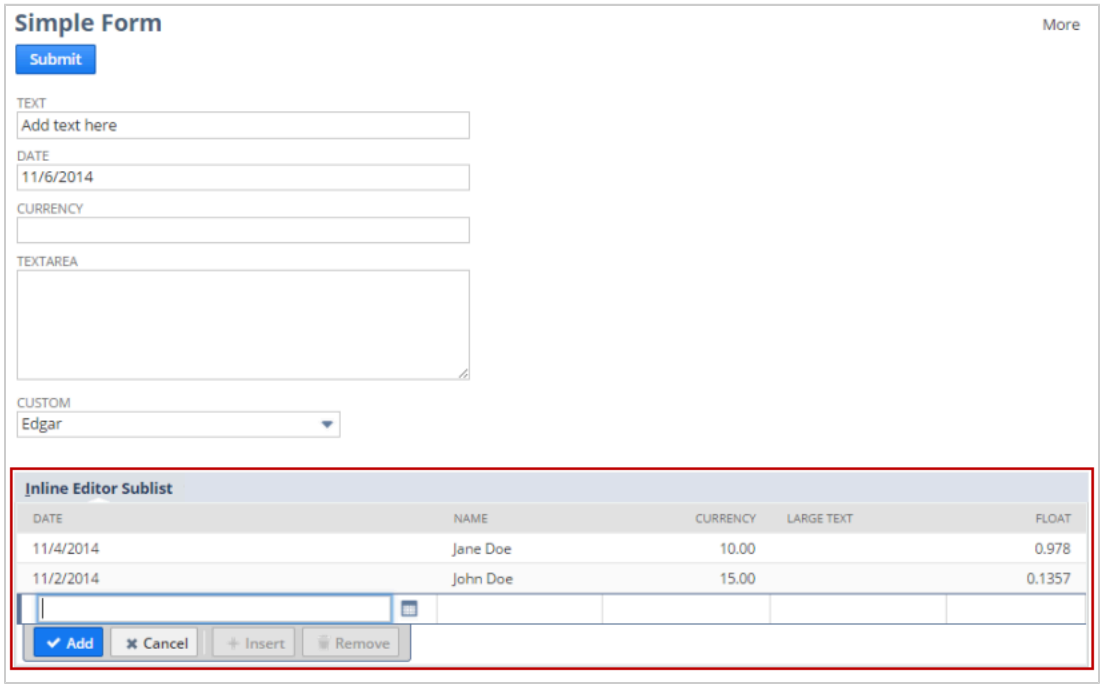
Example 2

The following example shows how to add an inline editor sublist to a Suitelet by instantiating the nlobForm object (using nlapiCreateForm(title, hideNavbar)) and then calling nlobjForm.addSubList(name, type, label, tab). Note that because you are not adding field or sublist elements to an existing NetSuite form, you do not need to prefix the element internal IDs with **custpage**.

Script:

```
function createSuiteletWithSublist(request, response)
{
   if (request.getMethod() == 'GET' )
     {
     // create the form
     var form = nlapiCreateForm('Simple Form');

     // add fields to the form
     var field = form.addField('textfield','text', 'Text');
     field.setLayoutType('normal','startcol')
     form.addField('datefield','date', 'Date');
     form.addField('currencyfield','currency', 'Currency');
     form.addField('textareafield','textarea', 'Textarea');

     // add a select field and then add the select options that will appear in the dropdown list
     var select = form.addField('selectfield','select','Custom');
     select.addSelectOption('','');
     select.addSelectOption('a','Albert');
     select.addSelectOption('b','Baron');
     select.addSelectOption('c','Chris');
     select.addSelectOption('d','Drake');
     select.addSelectOption('e','Edgar');

     // add a sublist to the form
     var sublist = form.addSubList('sublist','inlineeditor','Inline Editor Sublist', 'tab1');

     // add fields to the sublist
     sublist.addField('sublist1','date', 'Date');
     sublist.addField('sublist2','text', 'Name');
     sublist.addField('sublist3','currency', 'Currency');
     sublist.addField('sublist4','textarea', 'Large Text');
     sublist.addField('sublist5','float', 'Float');

     // make the Name field unique. Users cannot provide the same value for the Name field.
     sublist.setUniqueField('sublist2');

     form.addSubmitButton('Submit');
```

```
38
39      response.writePage( form );
40      }
41  }
```



Note: The nlapiRefreshLineItems(type) API can be used to refresh static list sublists that have been added using nlobjSubList. This API implements the behavior of the Refresh button on the UI.

Working with Sublist Line Items

NetSuite provides several Sublist APIs to manipulate sublist line items. You can use these APIs to add or remove line items, update multiple line items when a body field is changed, or automate the population of line items when certain conditions exist on the form.

When scripting with sublists, you should know if you are scripting an Editor Sublists, Inline Editor Sublists, or List Sublists sublist. Because List Sublists sublists are not dynamic, you cannot add/remove lines. You can only get/set values that currently exist on the sublist.

Whether you are scripting an editor, inline editor, or list sublist, generally you will specify one or more of the following in the sublist API:

1. The sublist internal ID

   Example : 'salesteam' (appears in the UI as the **Sales Team** sublist)

2. The sublist line item (field) ID.

   Example : 'isprimary' (appears in the UI as **Primary**)

3. The sublist line number — doing so enables you to specify *where* on the sublist you want to change, add, or remove a line. Note that first line on a sublist is **1** (not 0).

4. The value of the line item.

Example : The value can be defined directly in the API or it can be a passed in value that was defined elsewhere in the script.

Example

```
nlapiSetLineItemValue('salesteam', 'isprimary', 2, 'T');
```

Important: When you edit a sublist line with SuiteScript, it triggers an internal validation of the sublist line. If the line validation fails, the script also fails. For example, if your script edits a closed catch up period, the validation fails and prevents SuiteScript from editing the closed catch up period.

Adding and Removing Line Items

To add/remove sublist line items, follow the general guidelines provided below. The approach you follow depends on whether you are writing a client script to attach to a record, or a server script that loads a record from the database. (In this context, scripts that are considered to be **server scripts** are Suitelets, user event scripts, and scheduled scripts. Scripts considered to be **client scripts** are form- and record-level client scripts.)

Important: This section does not apply to List Sublists. List sublists contain information that cannot be dynamically added or removed in either the UI or in SuiteScript. For information about getting/setting existing values on a list sublist, see Getting and Setting Line Item Values.

Client Scripts

1. (Optionally) Call nlapiGetLineItemCount(type) to get the number of lines in the sublist. Alternatively you can call nlapiGetCurrentLineItemIndex(type) to return the number of the currently select line item.
2. Call either nlapiInsertLineItem(type, line) or nlapiRemoveLineItem(type, line) to add/remove a line. In the **line** argument you will specify the line number of the line you want to add/remove.
3. If adding a line:
    1. Call nlapiSelectNewLineItem(type) to select and insert a new line (as you would in the UI).
    2. Call nlapiSetCurrentLineItemValue(type, fldnam, value, firefieldchanged, synchronous) to set the value of the line.
4. Call nlapiCommitLineItem(type) to commit/save the changes to the sublist.
5. Perform steps 3 and 4 as many times as necessary to add all line items.

Server Scripts

1. Load the record object — for example using nlapiLoadRecord(type, id, initializeValues).
2. (Optionally) Call nlobjRecord.getLineItemCount(group) to get the number of lines in the sublist.
3. Call either nlobjRecord.insertLineItem(group, linenum, ignoreRecalc) or nlobjRecord.removeLineItem(group, linenum, ignoreRecalc) to add/remove a line. In the **group** argument specify by line number where to add/remove the line. Line numbering begins with **1**, not 0.

4. If adding a line:
    1. Call nlobjRecord.selectNewLineItem(group) to select and insert a new line (as you would in the UI).
    2. Call nlobjRecord.setCurrentLineItemValue(group, name, value) to set the value of the line.
5. Call nlobjRecord.commitLineItem(group, ignoreRecalc) to commit/save the changes to the sublist.
6. Submit the record using nlapiSubmitRecord(record, doSourcing, ignoreMandatoryFields).

Example 1 (Server Script)

This sample shows how to create a new Vendor Bill record and then add items to the Item sublist and expenses to the Expenses sublist. Note that because you are adding new lines to each sublist, you must call the nlobjRecord.selectNewLineItem() method. You then set all values for the new lines using the nlobjRecord.setCurrentLineItemValue() method. When you are finished adding values to each line in the sublist, you must commit each line to the database. You will call the nlobjRecord.commitLineItem() method to commit each line.

```
var record = nlapiCreateRecord('vendorbill');
record.setFieldValue('entity', 196);
record.setFieldValue('department', 3);
record.selectNewLineItem('item');
record.setCurrentLineItemValue('item','item',380);
record.setCurrentLineItemValue('item', 'location', 102);
record.setCurrentLineItemValue('item', 'amount', '2');
record.setCurrentLineItemValue('item', 'customer',294);
record.setCurrentLineItemValue('item','isbillable','T');
record.commitLineItem('item');

record.selectNewLineItem('expense');
record.setCurrentLineItemValue('expense','category',3);
record.setCurrentLineItemValue('expense', 'account', 11);
record.setCurrentLineItemValue('expense', 'amount','10');
record.setCurrentLineItemValue('expense','customer',294);
record.setCurrentLineItemValue('expense','isbillable','T');
record.commitLineItem('expense');

var id = nlapiSubmitRecord(record, true);
```

This sample shows how to add a line to a sublist. When the record is saved, the updates to the sublist are committed to the database.

```
//Load a sales order. 187 is the internal ID of the sales order
var rec = nlapiLoadRecord('salesorder', 187);

//Insert a new line at the start of Item sublist
rec.insertLineItem('item', 1);

//Set the value of quantity to 10 on the first line of the sublist
rec.setLineItemValue('item', 'quantity', 1, 10);

//Set the value of currency to 1 (the internal ID for US dollar) on the first line of the sublist
rec.setLineItemValue('item', 'currency', 1, 1);

//Submit the record to commit the sublist changes to the database
var id = nlapiSubmitRecord(rec, true);
```

Example 2 (Server Script)

This sample shows how to use nlobjRecord.getLineItemCount(group), which is used to determine the number of lines in a sublist. In this sample, a line item is added to the end of the Items sublist. When the record is saved, the updates to the sublist are committed to the database.

```
//Get the new record
```

```
var rec = nlapiGetNewRecord();

//Determine the number of lines on the Item sublist
var intCount = rec.getLineItemCount('item');

//Insert a line after the line that already exists
rec.insertLineItem('item', intCount + 1);

//Set the value of the line item
rec.setCurrentLineItemValue('item', 'quantity', intCount + 1, 10);

// Commit the sublist line changes
rec.commitLineItem('item');

// Submit the record to commit all change to the database
var id = nlapiSubmitRecord(rec, true);
```

Example 3 (Client Script)

This sample shows how to add a line item to a transaction using a client script. Be aware that in client scripting you must always use nlapiCommitLineItem(type) to commit any line item changes to the sublist.

In this example you first insert the line and then commit the line. If you set the item field using nlapiSetCurrentLineItemValue(type, fldnam, value, firefieldchanged, synchronous), you cannot call nlapiCommitLineItem until the server call for the item information has completed. The only way to know that the server call is complete is to create a post-sourcing function that sets a flag.

For example, suppose you want to insert a shipping line when a user clicks a button. You can attach a function such as insertShippingRate() to that button, which adds an item named “Shipping”, sets its rate, and then commits the line.

```
function insertShippingRate()
{
   nlapiSelectNewLineItem('item');

   /* important so that you know that the script was called from insertShippingRate() */
   nlapiSetCurrentLineItemValue('item', 'custcolinsertshippingrate', true);
   nlapiSetCurrentLineItemText('item', 'item', 'Shipping');
}
function doPostSourcing(type, fldname)
{
   if ( type == 'item' && fldname == 'item' && nlapiGetCurrentLineItemValue
   ('item', 'custcolinsertshippingrate') == true )
   {
      nlapiSetCurrentLineItemValue('item', 'custcolinsertshippingrate', false);
      nlapiSetCurrentLineItemValue('item', 'rate', '7.50');
      nlapiCommitLineItem('item');
   }
}
```

Getting and Setting Line Item Values

You can use both client and server scripts to get/set line item values. The set/get guidelines provided here can be used on Editor Sublists, Inline Editor Sublists, and List Sublists sublists.

Example 1

The following sample includes several Sublist APIs. This sample copies sales reps from the Sales Team sublist of one sales order to another sales order, ignoring those on the Sales Team sublist who are not sales reps.

```
// Copy all the reps from the original order to the adjusting order
```

```
    var iRep = 1;
    var reps = originalSo.getLineItemCount('salesteam');

     for (var rep = 1; rep <= reps; rep++)
     {
         // If the role is not sales rep, ignore it
         if (originalSo.getLineItemValue('salesteam', 'salesrole', rep) != '-2')
             continue;
         var reppct = originalSo.getLineItemValue('salesteam', 'contribution', rep);
         if (reppct != '0.0%')
         {
             var repId = originalSo.getLineItemValue('salesteam', 'employee', rep);
             // keep the percent the same
             if (reppct.substring(reppct.length-1) == '%')
             {
                 //remove the percent sign % from the end
                 reppct = reppct.substring(0, reppct.length-1);
             }
             so.insertLineItem('salesteam', iRep);
             so.setCurrentLineItemValue('salesteam', 'contribution', iRep, reppct);
             so.setCurrentLineItemValue('salesteam', 'employee', iRep, repId);

      // copy the role
      so.setCurrentLineItemValue('salesteam','salesrole',iRep,originalSo.getLineItemValue
         ('salesteam','salesrole', rep));

    // If primary rep on original order make it primary on the new sales order
             var primary = originalSo.getLineItemValue('salesteam', 'isprimary', rep);
             so.setCurrentLineItemValue('salesteam', 'isprimary', iRep, primary);
             iRep++

     so.commitLineItem('salesteam');
         }

     }
     // save the new order and return the ID
     var soId = nlapiSubmitRecord(so, true);
```

Example 2

The following sample is a validateLine client script which uses nlapiGetCurrentLineItemValue(type, fldnam) to prevent the addition of Sales Order item lines with an amount greater than 10000.

```
function validateLine(group)
{
   var newType = nlapiGetRecordType();
   if ( newType == 'salesorder' && group == 'item' && parseFloat(nlapiGetCurrentLineItemValue('item','amount')) > 10000 )
   {
      alert('You cannot add an item with amount greater than 10000.')
      return false;
   }
   return true;
}
```

Working with Item Groups in a Sublist

NetSuite item groups are stocked and sold as single units, even though they consist of several individual items. Item groups are used to sell vendor-specific objective evidence (VSOE) item group bundles, which can contain both taxable and nontaxable items.

You can use SuiteScript to interact with item groups in the same way you use the UI. Item Group type items are added to transactions as other line items are. In the case of an Item Group item, the item group expands to its member items. Item groups can optionally include start and end lines. The SuiteScript behavior emulates the behavior of how you would add an item group in the UI.

Example

In this example, an Item Group item is added to a transaction, and the tax code propagates to the members of the group.

```
var rec = nlapiCreateRecord( 'cashsale' );
rec.setFieldValue( 'entity', '76' );   //set the customer
rec.selectNewLineItem( 'item' );
rec.setCurrentLineItemValue( 'item', 'item', '66');  //item group item
rec.setCurrentLineItemValue( 'item', 'quantity', 1);
rec.setCurrentLineItemValue( 'item', 'taxcode', -7 );  //set to non-taxable
rec.commitLineItem( 'item' );
var id = nlapiSubmitRecord( rec );   //on submit the item group expands
```

Working with Sublists in Dynamic Mode and Client SuiteScript

When you copy, create, load, or transform a record in dynamic mode, you are also interacting with all of the record's elements in dynamic mode; this includes a record's sublists.

Note: When using client SuiteScript on a sublist, you must also script in a way that emulates the behaviors of the UI. Consequently, an API such as nlapiSetLineItemValue(type, fldnam, linenum, value) will generally not be supported in client scripts. Read the rest of this section for more details.

Note: If you are unfamiliar with the concept of dynamic scripting, see SuiteScript 1.0 Working with Records in Dynamic Mode for details.

When scripting against a sublist that is in dynamic mode, the following APIs will **NOT** work when adding a line or changing the values of an existing line:

- nlapiSetLineItemValue(type, fldnam, linenum, value) - used when scripting in a "current record" context, for example in user event scripts.
- nlobjRecord.setLineItemValue(group, name, linenum, value) - used when scripting the nlobjRecord object itself, as it exists on the server.

These APIs will not work in dynamic mode or in client SuiteScript because they have no UI correlation. One of the primary components of dynamic and client scripting is that they emulate the behaviors of the UI.

When users interact with sublists in the UI, they first select the sublist they want to work with, then they select the line they want to add or change, and finally they click the Add button to commit the line to the database. When you are scripting a sublist in dynamic mode or in client SuiteScript, calling nlapiSetLineItemValue(type, fldnam, linenum, value) does not provide enough context for the script to execute. Instead, you will follow one of these two patterns when adding or changing a line:

To add a new line:

1. nlapiSelectNewLineItem(type) - to specify the sublist you want to work with.
2. nlapiSetCurrentLineItemValue(type, fldnam, value, firefieldchanged, synchronous) - to set values on the current line.

3. nlapiCommitLineItem(type) - to commit the line to the database.

Example:

This sample creates a new sales order in dynamic mode, and then adds two new items to the Items sublist.

```
var record = nlapiCreateRecord('salesorder', {recordmode: 'dynamic'});

// add the first item
record.selectNewLineItem('item');
record.setCurrentLineItemValue('item', 'item', 556);
record.setCurrentLineItemValue('item', 'quantity', 2);
record.commitLineItem('item');

// add the second item
record.selectNewLineItem('item');
record.setCurrentLineItemValue('item', 'item', 380);
record.setCurrentLineItemValue('item', 'quantity', '2');
record.setCurrentLineItemValue('item', 'amount', '0.1');
record.commitLineItem('item');
```

To change values on an existing line:

1. nlapiSelectLineItem(type, linenum) - to specify the sublist you want to work with and the existing line you want to change.
2. nlapiSetCurrentLineItemValue(type, fldnam, value, firefieldchanged, synchronous) - to set values on the current line.
3. nlapiCommitLineItem(type) - to commit the line to the database.

Example:

This sample loads a sales order in dynamic mode, and then modifies a line that already exists.

```
var record = nlapiLoadRecord('salesorder', 55, {recordmode: 'dynamic'});

// modify an existing line
record.selectLineItem('item', 1);
record.setCurrentLineItemValue('item', 'item', 556);
record.setCurrentLineItemValue('item', 'quantity', 2);
record.commitLineItem('item');
```

- Example:

  This sample loads a sales order in dynamic mode, and then inserts a new item on line one. The item that was previously on line one moves to line two.

```
var record = nlapiLoadRecord('salesorder', 1966, {recordmode: 'dynamic'});
record.insertLineItem('item', 1);
record.setCurrentLineItemValue('item','item', 98);
record.commitLineItem('item');
```

Sublist Errors

You can only set line items that are valid. If you attempt to set a line that does not exist, you will receive an “Invalid Sublist Operation” out-of-bounds error. The exception to this is on Suitelets. Because Suitelets contain only your data, you will not receive a NetSuite error.

Example

Working with Sublists in Standard Mode and Client SuiteScript

In standard mode, sublist values are not automatically sourced when you select a record in SuiteScript.

Do the following steps to source values when you select a record:

1. Submit the record with a new placeholder sublist line item.
2. Reload the record.
3. Remove the line item.
4. Set appropriate line item values.
5. Submit the record. After it is submitted, the record is pre-selected on the form for the next new transaction.
6. Reload the record to verify that values are automatically sourced.

Example

The following example shows how to update the Payment sublist of a Deposit record in standard mode:

```
var rec = nlapiCreateRecord('deposit');

// Set the account.
rec.setFieldValue('account', 123);
// In Standard mode, setFieldValue does not source values in the Payment sublist.

// Insert a placeholder line to enable record submission.
rec.selectNewLineItem('other');
rec.setCurrentLineItemValue('other', 'account', 456);
rec.setCurrentLineItemValue('other', 'amount', 0);
rec.commitLineItem('other');
var id = nlapiSubmitRecord(rec);

// Load the record.
var fin = nlapiLoadRecord('deposit', id);
// The Payment sublist is now populated.

// Remove the placeholder line.
fin.removeLineItem('other', 1);

// Apply values from the Payment sublist.
fin.setLineItemValue('payment','deposit','1','T');

// Submit the finalized record.
var id = nlapiSubmitRecord(fin);
```

Working with Online Forms

Only the APIs listed in the following table are supported on online forms.

Important: These are also the only APIs supported on externally available Suitelets (Suitelets set to Available Without Login on the Script Deployment page). For more information about externally available Suitelets, see SuiteScript and Externally Available Suitelets.

SuiteScript APIs available on online forms and externally available Suitelets

- nlapiAddDays(d, days)
- nlapiAddMonths(d, months)
- nlapiCancelLineItem(type)
- nlapiDateToString(d, format)
- nlapiDisableField(fldnam, val)
- nlapiDisableLineItemField(type, fldnam, val)
- nlapiEncrypt(s, algorithm, key)
- nlapiEscapeXML(text)
- nlapiFormatCurrency(str)
- nlapiGetCurrentLineItemIndex(type)
- nlapiGetCurrentLineItemText(type, fldnam)
- nlapiGetCurrentLineItemValue(type, fldnam)
- nlapiGetFieldText(fldnam)
- nlapiGetLineItemText(type, fldnam, linenum)
- nlapiIsLineItemChanged(type)
- nlapiRefreshLineItems(type)
- nlapiRemoveLineItemOption(type, fldnam, value)
- nlapiRemoveSelectOption(fldnam, value)
- nlapiSelectLineItem(type, linenum)
- nlapiSelectNewLineItem(type)
- nlapiGetLineItemCount(type)
- nlapiGetFieldValue(fldnam)
- nlapiSetFieldValue(fldnam, value, firefieldchanged, synchronous)
- nlapiGetLineItemValue(type, fldnam, linenum)
- nlapiSelectNode(node, xpath)
- nlapiSelectNodes(node, xpath)
- nlapiSelectValue(node, xpath)
- nlapiSelectValues(node, path)
- nlapiStringToDate(str, format)
- nlapiStringToXML(text)
- nlapiXMLToString(xml)
- nlapiSetLineItemValue(type, fldnam, linenum, value)
- nlapiInsertLineItem(type, line)
- nlapiRemoveLineItem(type, line)
- nlapiGetRecordType()
- nlapiGetRecordId()
- nlapiGetRole()
- nlapiGetUser()

Important: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). The NetSuite UI should only be accessed using SuiteScript APIs.

Why are only certain APIs supported on online forms?

For security reasons, many SuiteScript APIs are not supported on online forms or externally available (Available Without Login) Suitelets. Online forms and externally available Suitelets are used for generating stateless pages that access or manipulate account information that is not considered to be confidential. Therefore, scripts running on these pages cannot be used to access information about the server because that would require a valid NetSuite session (through user authentication).

Note that server-side SuiteScript execution for such pages (for example, user events and Suitelet page generation or backend code) have no such restrictions.

Note: nlapiGetRole() always returns -31 (the online form user role) when used in this context. nlapiGetUser() returns -4, the return value for an entity if a user cannot be properly identified by NetSuite. This occurs when the user has not authenticated to NetSuite, for example, when using externally available (Available without Login) Suitelets or online forms.

The APIs listed in the previous section all operate on the current page and will run as expected without a valid NetSuite session. Note that both types of pages (online forms and externally available Suitelets) are hosted on a NetSuite domain called `<accountID>.extforms.netsuite.com`. Having a separate domain for online forms and externally available Suitelets prevents secure NetSuite sessions established on `<accountID>.app.netsuite.com` from carrying over to these pages.

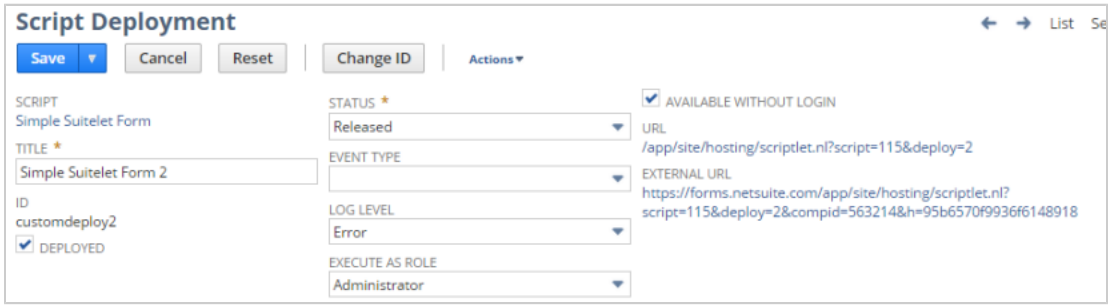
NetSuite supports TLS 1.2 encryption for `<accountID>.extforms.netsuite.com`, `<accountID>.app.netsuite.com`, and other NetSuite domains. Only requests sent using TLS encryption are granted access.

The following figure uses a Suitelet Script Deployment page to show the two domains types. In this case, the Available Without Login preference is selected. When this Suitelet is called, it will be called from the `<accountID>.extforms.netsuite.com` domain. So long as only the APIs listed in the table SuiteScript APIs available on online forms and externally available Suitelets have been used (in addition to any UI Objects), this externally available Suitelet will load and run as intended.

If the Available Without Login preference is not set, the Suitelet will be called from the login domain `<accountID>.app.netsuite.com`

For more information about NetSuite domains, see the help topic Understanding NetSuite URLs.

Note: Although it is not shown on the Script Deployment record, the internal URL is prepended with https://<accountID>.app.netsuite.com.



Inline Editing and SuiteScript

The following topics are covered in this section:

Inline Editing and SuiteScript Overview

In the NetSuite UI, inline editing lets you edit fields directly from a record list or from a set of search results. (See the help topic Using Inline Editing in the NetSuite Help Center for general information about inline editing not related to SuiteScript.)

In SuiteScript, the equivalent of inline editing is changing the value of a field without loading and submitting the entire record the field appears on. This is done using nlapiSubmitField(type, id, fields, values, doSourcing). See Inline Editing Using nlapiSubmitField for details.

Be aware that in SuiteScript, inline editing and mass updating are considered to be event types that can trigger user event scripts. When users inline edit a field in the UI, or when they perform a mass update, these two event contexts can trigger the execution of a user event script if the script's context type has been set to **xedit**. See Inline Editing (xedit) as a User Event Type and User Event Scripts.

Important: When using SuiteScript to inline edit a field on a record, note the following:

- In the UI and in SuiteScript, you can only perform inline editing on **body fields**. You cannot inline edit sublist fields. If you do not know the distinction between body and sublist fields, see Working with Fields Overview in the NetSuite Help Center.
- In SuiteScript, you cannot inline edit select fields. In other words, you cannot call nlapiSubmitField on a select field.
- In the NetSuite UI, users cannot set fields that are not inline editable. SuiteScript, however, **does** let you set non inline editable fields using nlapiSubmitField, but this is NOT the intended use for this API. See Consequences of Using nlapiSubmitField on Non Inline Editable Fields to learn about the increased governance cost of using this API on non inline editable fields.
- You can use the nlapiSubmitField function to inline edit inline-editable body fields on **SuiteScript-supported** records only. Do not use nlapiSubmitField (or any other SuiteScript API) on a record that does not officially support SuiteScript. For a list of records that officially support SuiteScript, see the help topic SuiteScript Supported Records in the NetSuite Help Center.
- If you want to perform inline editing through the UI or through SuiteScript, you must first enable the Inline Edit feature in your account. Enable this feature by going to Setup > Company > Enable Features. In the Data Management section, click the Inline Editing check box.

Why Inline Edit in SuiteScript?

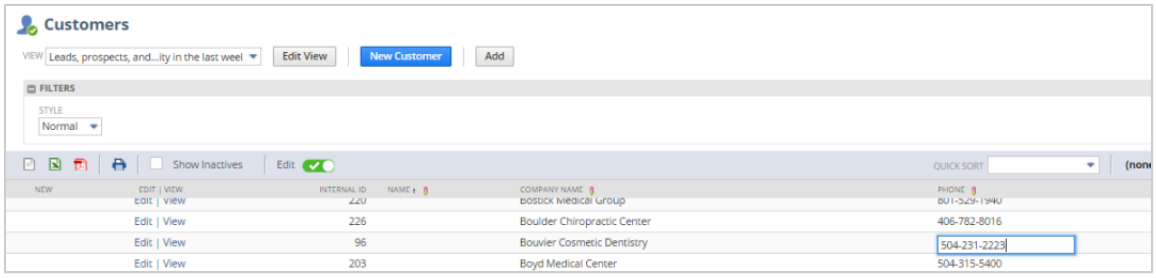
To change values on a record you can either load and submit the entire record, or you can call nlapiSubmitField(type, id, fields, values, doSourcing) on a specified body field or fields. Calling the nlapiSubmitField function, which is the programmatic equivalent of inline editing, requires less database processing since the entire record object is not being loaded to make a single field update. Note that in the UI and in SuiteScript, not all fields are inline editable.

Important: In the NetSuite UI, users cannot set fields that are not inline editable. SuiteScript, however, **does** let you set non inline editable fields using nlapiSubmitField, but this is NOT the intended use for this API. See Consequences of Using nlapiSubmitField on Non Inline Editable Fields to learn about the increased governance cost of using this API on non inline editable fields.

Inline Editing Using nlapiSubmitField

The SuiteScript equivalent of inline editing a single field or multiple fields on a record is calling the nlapiSubmitField(type, id, fields, values, doSourcing) function. After updating a field's value using nlapiSubmitField, you **do not** need to then call nlapiSubmitRecord to commit the change to the database.

The following figure shows that the **Phone** field on a customer record is being inline edited in the UI. To save the change, a user needs to click away from the field.



In SuiteScript, the programmatic equivalent of inline editing the Phone field on customer record 96 is:

```
var updatefield = nlapiSubmitField('customer', '96', 'phone', '504-231-3754');
```

In one call, you can reference a specific record and field, and then set a new value for that field. The entire process consumes only 10 units, which, in many cases, makes updating fields through nlapiSubmitField preferable to loading a record, referencing the field on the record, setting a value for the field, and then submitting the entire record to the database. For example, the following script consumes 30 units to accomplish the same thing as the previous inline editing sample:

```
var rec = nlapiLoadRecord('customer', '96');  //10 units
rec.setFieldValue('phone', '504-231-3754');
var id = nlapiSubmitRecord(rec); //20 units
```

Note that with inline editing in SuiteScript you can update multiple fields on a record, and the unit count remains as **10**. In this example, three fields are updated, however, there is still only one call to nlapiSubmitField. For example:

```
var fields = new Array();
var values = new Array();
fields[0]='phone';
```

```
values[0] = "800-555-1234";
fields[1] = 'url';
values[1] = "www.goodtimeswithsuitescript.com";
fields[2] = 'billpay';
values[2] = "T";
var updatefields = nlapiSubmitField('customer', '149', fields, values);
```

Important: If you are initiating a scheduled script from a user event script, and the user event type is set to xedit, no call to nlapiSubmitField within that scheduled script will save the field specified in nlapiSubmitField.

Important: In the NetSuite UI, users cannot set fields that are not inline editable. SuiteScript, however, **does** let you set non inline editable fields using nlapiSubmitField, but this is NOT the intended use for this API. See Consequences of Using nlapiSubmitField on Non Inline Editable Fields to learn about the increased governance cost of using this API on non inline editable fields.

Consequences of Using nlapiSubmitField on Non Inline Editable Fields

In the NetSuite UI, only certain fields are inline editable. These are fields that have no secondary relationship to other fields. When users update a field that is inline editable, only the data for *that* field is updated; there is no cascading effect on other data contained in the record.

Although nlapiSubmitField(type, id, fields, values, doSourcing) is the programmatic equivalent of inline editing, it *is* possible to use this API to update fields that are not inline editable in the UI. If a non inline editable field is submitted for update, all the data on the record will be updated appropriately. However, to support this, when a non inline editable field is submitted, the NetSuite backend must load the record, set the field(s), and then submit the record. Completing the "load record, set field, submit record" lifecycle for a record allows all validation logic on the record to execute.

Note: If an array of fields is submitted using nlapiSubmitField(...), and one field in the array is non inline editable, NetSuite also applies the same solution: the record is loaded in the backend, all fields are set, and the record is submitted.

Governance Implications

When you use nlapiSubmitField(type, id, fields, values, doSourcing) as it is intended to be used (to set one or more fields that are inline editable in the UI), the SuiteScript governance cost is **10 units**.

However, when you use nlapiSubmitField(...) to update fields that are NOT inline editable in the UI, the unit cost for nlapiSubmitField(...) is higher. Your script is charged the units it takes to load and submit a record.

For example, the unit cost of nlapiSubmitField(...) to set a non inline editable field on a transaction is:

1. load the record (nlapiLoadRecord) = 10 units
2. set the field = no units
3. submit the record (nlapiSubmitRecord) = 20 units

Total = **30 units**

It is best practice to use nlapiSubmitField(...) as it is intended to be used: to set fields that are inline editable in the UI. To help you know which fields are inline editable, you can refer to the UI.

Inline Editing (xedit) as a User Event Type

To set a user event script to execute in response to an inline edit field change or a mass update, specify **xedit** as the *type* argument in your script. The **xedit** type can be specified in beforeSubmit or afterSubmit user event scripts.

The following sample shows a user event script that will execute when a user inline edits a record, or the record is updated in a mass update. This script shows how to get all fields that were inline edited on the record or during the mass update.

```
function getUpdatedFields(type)
{
   // if the record is inline edited or mass updated, run the script
   if (type == ' xedit ')
   {
      // call nlapiGetNewRecord to get the fields that were inline edited/mass updated
      var fields = nlapiGetNewRecord().getAllFields()

      // loop through the returned fields
      for (var i = 0; i < fields.length; i++)
      {
         if (fields[i] == 'phone')
            nlapiSetFieldValue('phone', nlapiGetFieldValue('phone'))
      }
   }
}
```

Note: User event scripts are not executed upon mass updates of child matrix items from their parent items.

What's the Difference Between xedit and edit User Event Types?

When the user event *type* argument is set to **xedit**, it means that the execution context for the script is inline edit or mass update. In other words, if a user has inline edited a field on a record (or if the record has been part of a mass update), the user event script executes. In contrast, User Event Scripts set to execute when the *type* argument is set to **edit** execute when the record is edited in all other contexts. The script does not execute based on an inline edit or mass update.

Keep in mind that in SuiteScript, inline editing and mass updating are considered to be event types that can trigger user event scripts. When users inline edit a field in the UI, or when they perform a mass update, these two event contexts can trigger the execution of a user event script if the script's context type has been set to xedit. See Inline Editing (xedit) as a User Event Type.

Inline Editing and nlapiGetNewRecord()

In a user event script, if you have set the user event *type* argument to **xedit**, and you are using nlapiGetNewRecord() to return all the newly updated fields, be aware that only the fields which have been updated through an xedit event (inline edited or mass updated) will be returned. In many cases, this is only one or two fields.

In contrast, if the user event *type* argument is set to **edit**, and you call nlapiGetNewRecord() in a beforeSubmit, you will get back all the fields on the record.

For **xedit** user events, you should call nlapiGetNewRecord().getAllFields() to return an array of all the fields being changed in the inline edit, mass update, or nlapiSubmitField() operation.

Note: If you call getFieldValue() on a field that is not in that array, null is returned.

Inline Editing and nlapiGetOldRecord()

Although calling nlapiGetOldRecord() in an inline editing context requires more processing from the NetSuite database (and therefore may add to the user response time), there is less ambiguity when calling this method in an inline editing context than when calling nlapiGetNewRecord().

The following sample shows how nlapiGetOldRecord() is used in a user event script that executes in the context of an inline edit or mass update. This script logs all the revised field IDs in the record prior to being committed to the database. If the phone field is modified, the change is reverted.

```
function getUpdatedFields(type)
{
    // if the record is inline edited or mass updated, run this script
    if (type == 'xedit')
    {
        var recOldEmployee = nlapiGetOldRecord();
        var recUpdEmployee = nlapiGetNewRecord();

        // Get all the field IDs in the record
        var lstEmployeeFields = recOldEmployee.getAllFields();

        // Traverse through all the employee fields
        for (var i = 0; i < lstEmployeeFields.length; i++)
        {
            // If the record has a modified phone field, log the original and revised phone numbers
            if (lstEmployeeFields[i] == 'phone')
            {
                nlapiLogExecution('DEBUG', 'Old Phone #', recOldEmployee.getFieldValue('phone'));
                nlapiLogExecution('DEBUG', 'New Phone #', recUpdEmployee.getFieldValue('phone'));

                // Revert the change
                   nlapiSetFieldValue('phone', recOldEmployee.getFieldValue('phone'));
            }
        }
    }
}
```

SuiteScript 1.0 Searching Overview

Similar to much of the searching functionality available through the NetSuite UI, SuiteScript Search APIs allow you to retrieve real-time data from your account. You can search for a single record by keywords, create saved searches, search for duplicate records, or return a set of records that match filters you define.

The following sections provide details on searching with SuiteScript. If you are new to SuiteScript searches, you should read these topics in order.

- Understanding SuiteScript Search Objects
- Search Samples
- Search APIs
- SuiteScript 1.0 Supported Search Operators, Summary Types, and Date Filters

Understanding SuiteScript Search Objects

Most SuiteScript searches use the following objects:

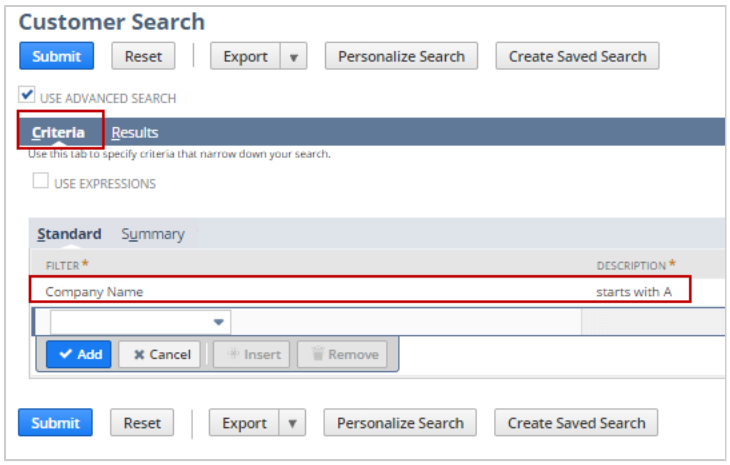
- nlobjSearchFilter: Defines filtering criteria for the search
- nlobjSearchColumn(name, join, summary): Defines search return columns for the search
- nlobjSearchResult: Holds the values of specific search results

After all filters and search columns are defined, the search is executed using the nlapiSearchRecord(type, id, filters, columns) function.

Important: If you are performing a global search or a duplicate record search, you will **not** use the objects listed above or the nlapiSearchRecord(type, id, filters, columns) function. For details on these types of searches, see Searching for Duplicate Records and Performing Global Searches.

Defining Search Filters

The following screenshot shows the UI equivalent of using the nlobjSearchFilter object to define search filters. In the UI, users define search filters by clicking the Criteria tab on the search record (in this case the Customer Search record).



In SuiteScript, the same filter value is specified using the nlobjSearchFilter object:

```
var filters = new Array();
filters[0] = new nlobjSearchFilter('companyname', null, 'startswith', 'A');
```

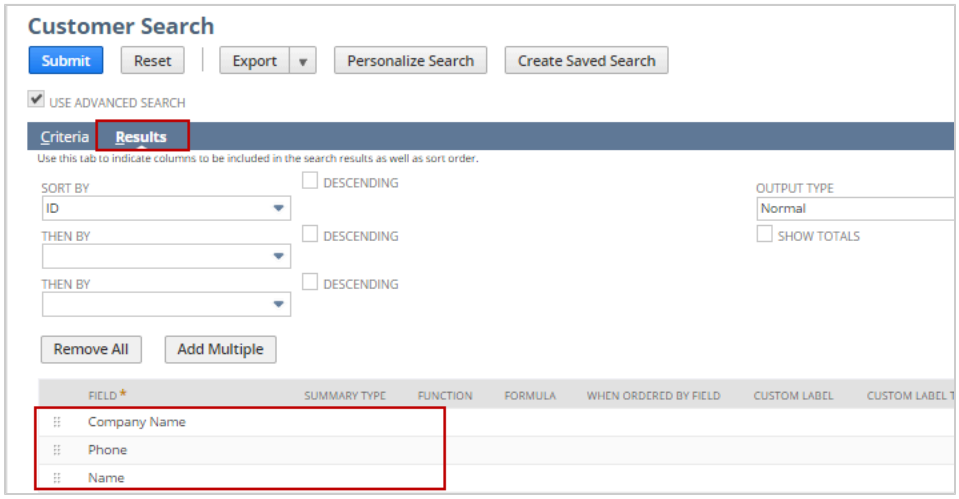
How do I know which search filters I can use in my code?

To determine which search filters are available for a specific record type:

1. Visit the SuiteScript Records Browser.
2. Find the record type you are using in your script.
3. In the documentation for that record type, see the **Search Filters** table. All available search filters for that record type are listed in the table.

Defining Search Columns

The following figure shows the UI equivalent of using the nlobjSearchColumn(name, join, summary) object to define search return columns. In the UI, users define search columns by clicking the Results tab on the search record (in this case the Customer Search record).



In SuiteScript, the same column values are specified using the nlobjSearchColumn(name, join, summary) object:

```
var columns = new Array();
columns[0] = new nlobjSearchColumn('entity');
columns[1] = new nlobjSearchColumn('phone');
columns[2] = new nlobjSearchColumn('companyname');
```

How do I know which search columns I can use in my code?

To figure out which search columns are available for a specific record type:

1. Visit the SuiteScript Records Browser.
2. Find the record type you are using in your script.
3. In the documentation for that record type, see the **Search Columns** table. All available search columns for that record type are listed in the table.

Executing the Search

In the UI, users click the Submit button to execute a search. In SuiteScript, the equivalent of clicking the Submit button is calling the nlapiSearchRecord(type, id, filters, columns) function:

```
// Define search filters
var filters = new Array();
filters[0] = new nlobjSearchFilter('companyname', null, 'startswith', 'A');

// Define search columns
var columns = new Array();
columns[0] = new nlobjSearchColumn('entity');
columns[1] = new nlobjSearchColumn('phone');
columns[2] = new nlobjSearchColumn('companyname');

```

```
// Execute the customer search. You must specify the internal ID of
// the record type you are searching for. In this example, the internal
// ID is 'customer'. You can also pass the values defined in the
// filters and columns arrays to define filters and columns for the
// search.
var searchResults = nlapiSearchRecord('customer', null, filters, columns);
```

Getting Search Return Values

If you want to get specific values returned by the search, you use the nlobjSearchResult object to specify those values:

```
// Execute a search as described in the previous section, and store the
// search results in a variable named searchResults

// Get the value of the Company Name column from the first search result
var theValue = searchResults[0].getValue(columns[2]);
```

Search Samples

The following are samples of SuiteScript searches.

For more general information that describes search objects, see Understanding SuiteScript Search Objects.

Creating Saved Searches Using SuiteScript 1.0

The nlobjSearch object is the primary object used to encapsulate a NetSuite saved search. Note, however, you are **not required** to save the search results returned in this object.

To create a saved search, you will first define all search criteria and then execute the search using nlapiCreateSearch(type, filters, columns). The search will not be saved until you call the nlobjSearch.saveSearch method.

By default, searches returned by nlapiCreateSearch(...) will be private, which follows the saved search model in the UI. To make a saved search public, you must set the nlobjSearch.setIsPublic(type) method to true.

Note: When working with List/Record filters on searches, you must use the numeric internal ID to avoid throwing an error.

Creating a Saved Search Using Search Filter List

```
// Define search filters
var filters = new Array();
filters[0] = new nlobjSearchFilter( 'trandate', null, 'onOrAfter', 'daysAgo90' );
filters[1] = new nlobjSearchFilter( 'projectedamount', null, 'between', 1000, 100000 );
filters[2] = new nlobjSearchFilter( 'salesrep', 'customer', 'anyOf', \-5, null );

// Define return columns
var columns = new Array();
columns[0] = new nlobjSearchColumn( 'salesrep' );
columns[1] = new nlobjSearchColumn( 'expectedclosedate' );
columns[2] = new nlobjSearchColumn( 'entity' );

// Create the saved search
var search = nlapiCreateSearch( 'opportunity', filters, columns );
```

```
var searchId = search.saveSearch('My Opportunities in Last 90 Days', 'customsearch_kr');
```

Creating a Saved Search Using Search Filter Expression

```
//Define search filter expression
var filterExpression =   [ [ 'trandate', 'onOrAfter', 'daysAgo90' ],
                'or',
                [ 'projectedamount', 'between', 1000, 100000 ],
                'or',
                'not', [ 'customer.salesrep', 'anyOf', -5 ] ];

//Define return columns
var columns = new Array();
columns[0] = new nlobjSearchColumn( 'salesrep' );
columns[1] = new nlobjSearchColumn( 'expectedclosedate' );
columns[2] = new nlobjSearchColumn( 'entity' );

//Create the saved search
var search = nlapiCreateSearch( 'opportunity', filterExpression, columns );
var searchId = search.saveSearch('My Opportunities in Last 90 Days', 'customsearch_kr');
```

Using Existing Saved Searches

NetSuite saved searches allow you to create reusable search definitions with many advanced search filters/results display options. Although saved searches must be created in the UI, you can pass the internal ID of the saved search to SuiteScript and re-execute the search on a regular basis. In this manner, you can keep the searches up-to-date for all who might need to access the results.

To re-execute an existing saved search, you will use nlapiSearchRecord(type, id, filters, columns). The filters and columns parameters represent the criteria and results columns that you want to be included when the search is re-executed.

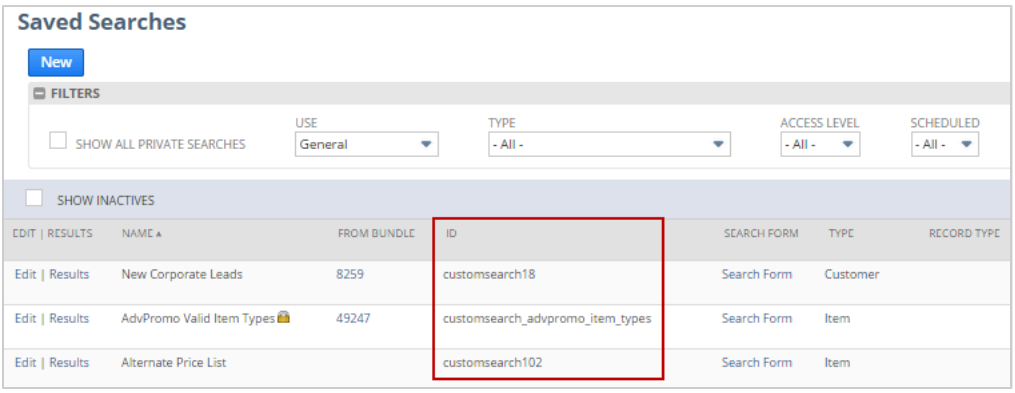
Note: For general information about NetSuite saved searches, see the help topic Saved Searches in the NetSuite Help Center.

When using the `nlapiSearchRecord` function to execute an existing saved search, note the following:

- Only saved searches on record types currently supported by SuiteScript can be executed. For a list of records that support SuiteScript, see the help topic SuiteScript Supported Records in the NetSuite Help Center.
- Saved searches acted on by SuiteScript should be protected. If a saved search is edited after script deployment is complete, the execution of the script could fail. You can add security to a saved search by defining access permissions in the search definition.

  Note: You may want to include the script administrator in an email notification for any time a saved search included in the script is updated. Email notifications can be defined on the Alerts tab of the saved search definition.
- On a Script record page, if you define a saved search as a List/Record script parameter, only saved searches that are public will appear in the List/Record parameter dropdown field. For information about script parameters, see the help topic Creating Script Parameters Overview in the NetSuite Help Center.
- In nlapiSearchRecord(type, id, filters, columns), the value of `id` can be the ID that appears in the **Internal ID** column on the Saved Searches list page (see figure below). Or it can be the value that appears in the **ID** column.

  If you have created a custom scriptId for your saved search, this will appear in the **ID** column (see figure). Note: To access the Saved Searches list page, go to Lists > Search > Saved Searches.



In the following code, a Customer saved search is executed. The ID customsearch57 references a specific saved search.

Note: The second parameter in nlapiSearchRecord(...) is treated as a variable instead of the custom Saved Search ID if it is not within quotes. Also note, if the Internal Id of the saved search is used instead of the Saved Search ID, the Internal Id does not need single quotes since it is evaluated as an integer.

```
function getEmail(firstname, lastname) {
     // Specify the record type and the saved search ID
     var searchresults = nlapiSearchRecord('customer', 'customsearch57', null, null);

     for ( var i = 0; searchresults != null && i < searchresults.length; i++ ) {
          var customerrecord = searchresults[i];

          if (customerrecord.getValue('firstname') == firstname && customerrecord.getValue('lastname') == lastname) {
               return customerrecord.getValue('email');
          }
     }
     return "Customer not found.";
}
```

Filtering a Search

The following samples provide examples for how to set various kinds of filtering criteria in a search. Also provided are samples that show how to filter the results.

Executing an Opportunity Search and Setting Search Filters

```
// Define search filters
var filters = new Array();
filters[0] = new nlobjSearchFilter( 'trandate', null, 'onOrAfter', 'daysAgo90' );
filters[1] = new nlobjSearchFilter( 'projectedamount', null, 'between', 1000, 100000 );
filters[2] = new nlobjSearchFilter( 'salesrep', 'customer', 'anyOf', -5, null );

// Define search columns
var columns = new Array();
columns[0] = new nlobjSearchColumn( 'salesrep' );
columns[1] = new nlobjSearchColumn( 'expectedclosedate' );
columns[2] = new nlobjSearchColumn( 'entity' );
columns[3] = new nlobjSearchColumn( 'projectedamount' );
columns[4] = new nlobjSearchColumn( 'probability' );
columns[5] = new nlobjSearchColumn( 'email', 'customer' );
columns[6] = new nlobjSearchColumn( 'email', 'salesrep' );

```

```
// Execute the search. You must specify the internal ID of the record type.
var searchresults = nlapiSearchRecord( 'opportunity', null, filters, columns );

// Loop through all search results. When the results are returned, use methods
// on the nlobjSearchResult object to get values for specific fields.
for ( var i = 0; searchresults != null && i < searchresults.length; i++ )
{
   var searchresult = searchresults[ i ];
   var record = searchresult.getId( );
   var rectype = searchresult.getRecordType( );
   var salesrep = searchresult.getValue( 'salesrep' );
   var salesrep_display = searchresult.getText( 'salesrep' );
   var salesrep_email = searchresult.getValue( 'email', 'salesrep' );
   var customer = searchresult.getValue( 'entity' );
   var customer_email = searchresult.getValue( 'email', 'customer' );
   var expectedclose = searchresult.getValue( 'expectedclosedate' );
   var projectedamount = searchresult.getValue( 'projectedamount' );
   var probablity = searchresult.getValue( 'probability' );
}
```

Executing an Opportunity Search and Setting Search Filter Expression

```
//Define search filter expression
var filterExpression =   [ [ 'trandate', 'onOrAfter', 'daysAgo90' ],
               'or',
               [ 'projectedamount', 'between', 1000, 100000 ],
               'or',
               'not', ['customer.salesrep', 'anyOf', -5 ]  ] ;

//Define search columns
var columns = new Array();
columns[0] = new nlobjSearchColumn('salesrep');
columns[1] = new nlobjSearchColumn('expectedclosedate');
columns[2] = new nlobjSearchColumn('entity');
columns[3] = new nlobjSearchColumn('projectedamount');
columns[4] = new nlobjSearchColumn('probability');
columns[5] = new nlobjSearchColumn('email', 'customer');
columns[6] = new nlobjSearchColumn('email', 'salesrep');

//Execute the search. You must specify the internal ID of the record type.
var searchresults = nlapiSearchRecord('opportunity', null, filterExpression, columns);

//Loop through all search results. When the results are returned, use methods
//on the nlobjSearchResult object to get values for specific fields.
for (var i = 0; searchresults != null && i < searchresults.length; i++)
{
   var searchresult = searchresults[i];
   var record = searchresult.getId();
   var rectype = searchresult.getRecordType();
   var salesrep = searchresult.getValue('salesrep');
   var salesrep_display = searchresult.getText('salesrep');
   var salesrep_email = searchresult.getValue('email', 'salesrep');
   var customer = searchresult.getValue('entity');
   var customer_email = searchresult.getValue('email', 'customer');
var expectedclose = searchresult.getValue('expectedclosedate');
var projectedamount = searchresult.getValue('projectedamount');
var probability = searchresult.getValue('probability');
}
```

Filtering Based on Box Fields

When filtering search results for box fields, use the **is** operator with **T** or **F** as the filter values. For example, in the following portlet script, all memorized Cash Sale transactions are returned.

```
function testPortlet(portlet) {
portlet.setTitle('Memorized Cash Sales');

var filters = new Array();
```

```
filters[0] = new nlobjSearchFilter('name', null, 'equalTo', '87', null);
filters[1] = new nlobjSearchFilter('memorized', null, 'is', 'T', null);

var columns = new Array();
columns[0] = new nlobjSearchColumn('internalid');
columns[1] = new nlobjSearchColumn('memorized');

var searchresults = nlapiSearchRecord('cashsale', null, filters, columns);
for ( var i = 0; searchresults != null && i < searchresults.length; i++ )
{
var searchResult = searchresults[i];
portlet.addLine(i+": "+searchResult.getValue('internalid')+",
"+searchResult.getValue('memorized'),null,0);
}
}
```

Executing a Customer Search and Filtering the Results

In the following sample, a search for all customer records (leads, prospects, customers) in the system is executed with the maximum limit of 10 results set. Note that in this sample, if you specify customer as the record type, customers, leads, and prospects are returned in the results.

```
function executeSearch()
{
var searchresults = nlapiSearchRecord( 'customer', null, null, null );
   for ( var i = 0; i < Math.min( 10, searchresults.length ); i++)
   {
      var record = nlapiLoadRecord(searchresults[i].getRecordType(), searchresults[i].getId());
   }
}
```

Filtering Based on None of Null Value

To search for a “none of null” value, meaning do not show results without a value for the specified field, use the @NONE@ filter. For example,

```
searchFilters[0] = new nlobjSearchFilter('class', null, 'noneof', '@NONE@');
```

In the following example, only customer records that match the entityid of test1 are returned.

```
function filterCustomers()
{
   var filters = new Array();
   filters[0] = new nlobjSearchFilter( 'entityid', null, 'contains', 'test1', null );
   var searchresults = nlapiSearchRecord('customer', 11, filters, null);
   var emailAddress = '';
   for ( var i = 0; searchresults != null && i < searchresults.length; i++ )
   {
      var searchresult = searchresults[ i ];
   }
}
```

Note: If it is unclear which values you can filter by for a specific filter variable, try performing a search that returns the value of a field so that you can see possible options.

Returning Specific Fields in a Search

You can use the nlobjSearchResult.getValue method to return the values of specific record fields. In the following example, the email fields for records returned from an existing customer saved search are returned. The script ID of the saved search is `customsearch8`.

```
function findCustomerEmails()
{
   var searchresults = nlapiSearchRecord('customer', customsearch8, null,null);
   var emailAddress = '';
   for ( var i = 0; searchresults != null && i < searchresults.length; i++ )
   {
      var searchresult = searchresults[ i ];
      emailAddress += searchresult.getValue( 'email' );
   }
}
```

For another example that shows how to return specific values in a search, see the sample for Searching on Custom Records.

Note: To improve performance, if you only need to access a specific subset of fields, you should limit the returned objects to include only that subset. This can be accomplished using the nlobjSearchFilter and nlobjSearchColumn(name, join, summary) objects.

Searching on Custom Records

Searching on custom records is the same as searching on standard (built-in) records. The following sample shows how to execute a search on a Warranty custom record type. (The internal ID for this record type is customrecord_warranty).

This sample shows how to define search filters and search columns, and then execute the search as you would for any other record type. This sample also shows how to use methods on the nlobjSearchResult object to get the values for the search results.

```
function searchWarranties()
{
// define search filters
var filters = new Array();
filters[0] = new nlobjSearchFilter( 'created', null, 'onOrAfter', 'daysAgo15' );

// return opportunity sales rep, customer custom field, and customer ID
var columns = new Array();
columns[0] = new nlobjSearchColumn( 'name' );
columns[1] = new nlobjSearchColumn( 'owner' );
columns[2] = new nlobjSearchColumn( 'custrecord_customer' );
columns[3] = new nlobjSearchColumn( 'custrecord_resolutiontime' );

// execute the Warrenty search, passing all filters and return columns
var searchresults = nlapiSearchRecord( 'customrecord_warranty', null, filters, columns );

// loop through the results
for ( var i = 0; searchresults != null && i < searchresults.length; i++ )
   {
   // get result values
   var searchresult = searchresults[ i ];
   var record = searchresult.getId( );
   var rectype = searchresult.getRecordType( );
   var name = searchresult.getValue( 'name' );
   var resolution = searchresult.getValue( 'custrecord_resolutiontime' );
   var customer = searchresult.getValue( 'custrecord_customer' );
   var customer_name = searchresult.getText( 'custrecord_customer' );
   }
}
```

This sample shows how you can do a search on custom records using a search filter expression.

```
function searchWarranties()
```

```
{
   //Define search filter expression
   var filterExpression =    ['created', 'onOrAfter', 'daysAgo15'];

   //Define search columns
   var columns = new Array();
   columns[0] = new nlobjSearchColumn('name');
   columns[1] = new nlobjSearchColumn('owner');
   columns[2] = new nlobjSearchColumn('custrecord_customer');
   columns[3] = new nlobjSearchColumn('custrecord_resolutiontime');

   //Execute the Warranty search, passing search filter expression and columns
   var searchresults = nlapiSearchRecord('customrecord_warranty', null, filterExpression, columns);

   //Loop through the results
   for (var i = 0; searchresults != null && i < searchresults.length; i++)
   {
      //Get result values
      var searchresult = searchresults[i];
      var record = searchresult.getId();
      var rectype = searchresult.getRecordType();
      var name = searchresult.getValue('name');
      var resolution = searchresult.getValue('custrecord_resolutiontime');
      var customer = searchresult.getValue('custrecord_customer');
      var customer_name = searchresult.getText('custrecord_customer');
   }
}
```

Searching Custom Lists

The following sample shows how to search a custom list.

```
var col = new Array();
col[0] = new nlobjSearchColumn('name');
col[1] = new nlobjSearchColumn('internalId');
var results = nlapiSearchRecord('customlist25', null, null, col);
   for ( var i = 0; results != null && i < results.length; i++ )
   {
      var res = results[i];
      var listValue = (res.getValue('name'));
      var listID = (res.getValue('internalId'));
      nlapiLogExecution('DEBUG', (listValue + ", " + listID));
   }
```

If you load a custom list record by using `nlapiLoadRecord('customlist','id')`, it will have values in a sublist `customvalue` that are arranged in the order that is defined during the custom list setup. For more information, see the help topic nlapiLoadRecord(type, id, initializeValues).

Executing Joined Searches

This example shows how to set values for a joined search. In this case you are executing an Item search that uses **Customer** and **Currency** (as specified on the Pricing record) as your filtering criteria.

You will define the join to the Pricing record in the nlobjSearchFilter object. You will define search return column values (also joins to the Pricing record) in the nlobjSearchColumn(name, join, summary) object. You will execute the **Item** search using nlapiSearchRecord(type, id, filters, columns).

```
// Create a filters array and define search filters for an Item search
var filters = new Array();

// filter by a specific customer (121) on the Pricing record
```

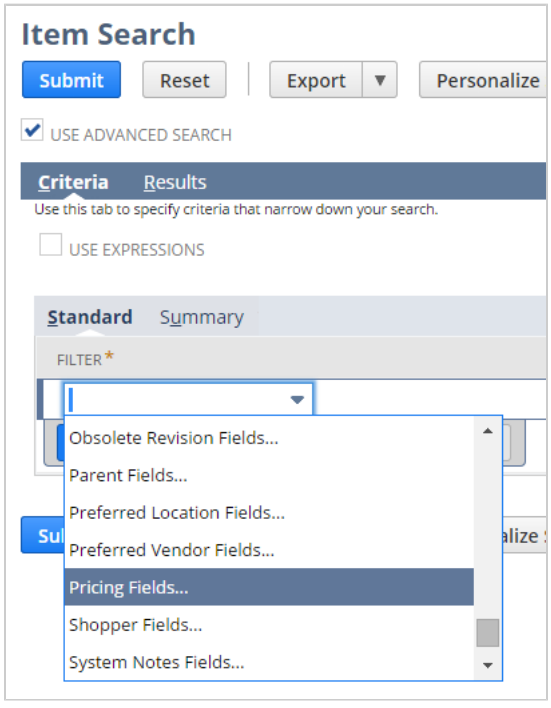
```
filters[0] = new nlobjSearchFilter('customer', 'pricing', 'is', '121');

// filter by a currency type (USA) on the Pricing record
filters[1] = new nlobjSearchFilter('currency', 'pricing', 'is', '1');

// set search return columns for Pricing search
var columns = new Array();

// return data from pricelevel and unitprice fields on the Pricing record
columns[0] = new nlobjSearchColumn('pricelevel', 'pricing');
columns[1] = new nlobjSearchColumn('unitprice', 'pricing');

// specify name as a search return column. There is no join set in this field.
// This is the Name field as it appears on Item records.
columns[2] = new nlobjSearchColumn('name');

// execute the Item search, which uses data on the Pricing record as search filters
var searchresults = nlapiSearchRecord('item', null, filters, columns);
```

The following figures show the UI equivalent of executing an Item search that uses filtering criteria pulled from the Pricing record. Note that on the Criteria tab, all available search joins for an Item search will appear at the bottom of the Filter dropdown list. Available join records are marked with the ellipsis (...) after the record name.

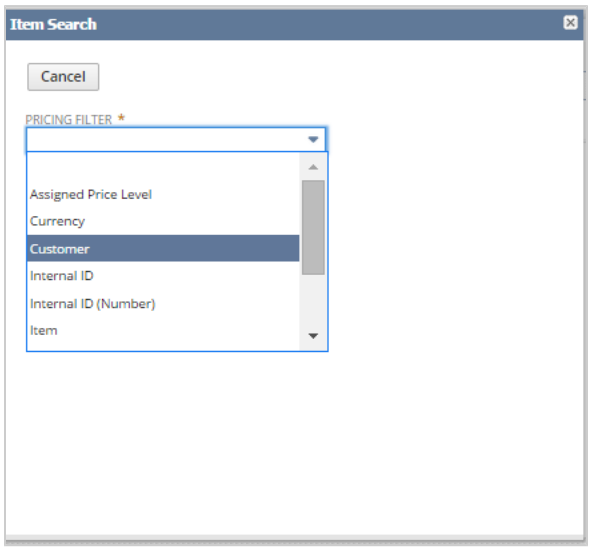
Note: Not all joins that appear in the UI are supported in SuiteScript. To see which joins are supported for a particular search, start by going to SuiteScript Supported Records. Click the record type that you want to execute the search on. Based on the example described below, you will click **Item Search** record. Then look to see which joins are supported for the record type.

The figures below show only how to set the filter values for a joined search. All of the same concepts apply when specifying search return column values.

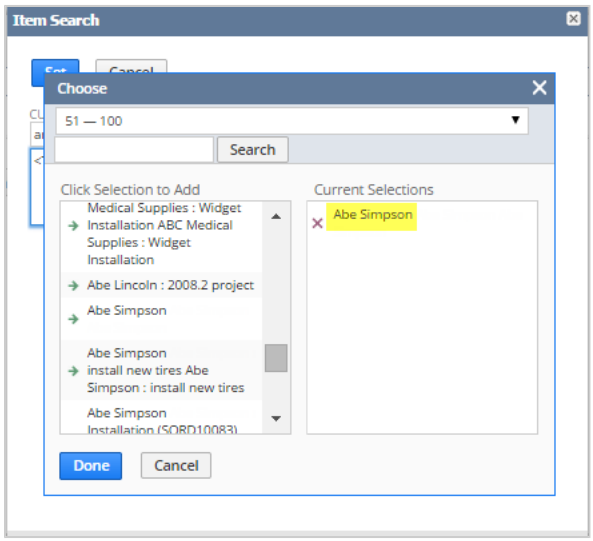
The first figure shows the Item Search record (Lists > Accounting > Items > Search).



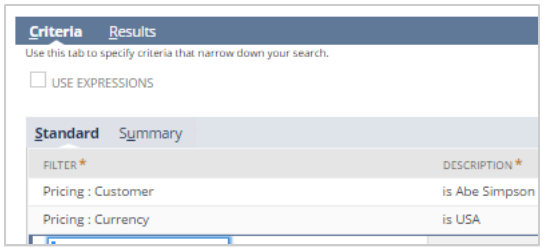
When **Pricing Fields...** is selected, a popup appears with all search fields that are available on the Pricing record (see figure below).



When you select the **Customer** field, another popup opens allowing you to select one or more customers. In the code sample, customer **121** is specified. In the UI, this customer appears as Abe Simpson (see below).



This figure shows how Item search / pricing join filtering criteria appear in the UI.



In SuiteScript, this looks like:

```
var filters = new Array();
```

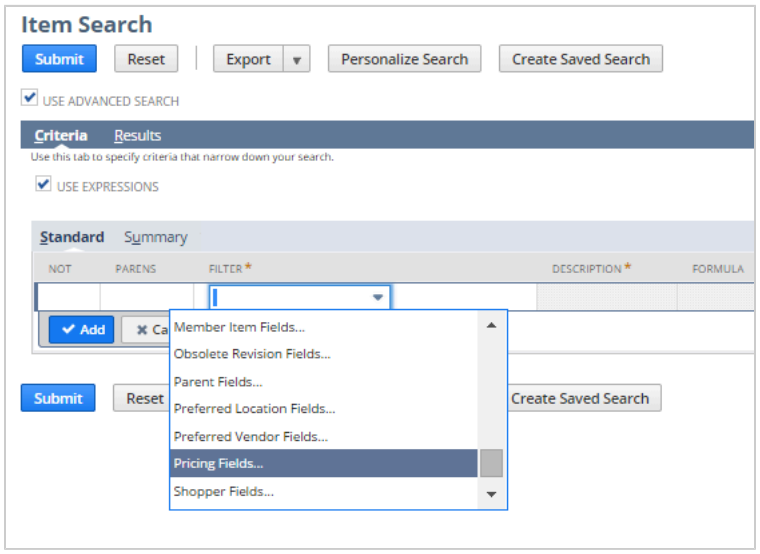
```
filters[0] = new nlobjSearchFilter('customer', 'pricing', 'is', '121');
filters[1] = new nlobjSearchFilter('currency', 'pricing', 'is', '1');
```

This example shows how you can execute joined searches using a search filter expression.

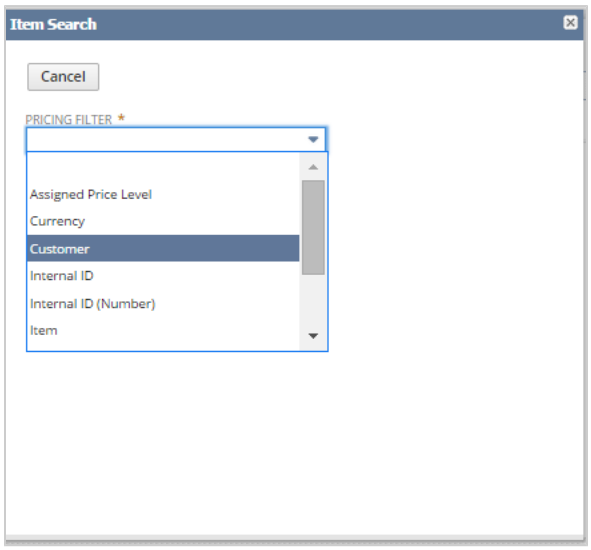
```
//Define search filter expression
var filterExpression =                [ [ 'pricing.customer', 'is', 121 ],
                'and',
                [ 'pricing.currency', 'is', 1 ] ];

//Define search columns
var columns = new Array();
//Return data from pricelevel and unitprice fields on the Pricing record
columns[0] = new nlobjSearchColumn('pricelevel', 'pricing');
columns[1] = new nlobjSearchColumn('unitprice', 'pricing');
//Specify name as a search return column. There is no join set in this field.
//This is the Name field as it appears on Item records.
columns[2] = new nlobjSearchColumn('name');

//Execute the Item search, which uses data on the Pricing record as search filter expression
var searchresults = nlapiSearchRecord('item', null , filterExpression, columns);
```

The following figures show the UI equivalent of executing the preceding example. These figures show only how to set the filter expression for a joined search. All of the same concepts apply when specifying search return column values.

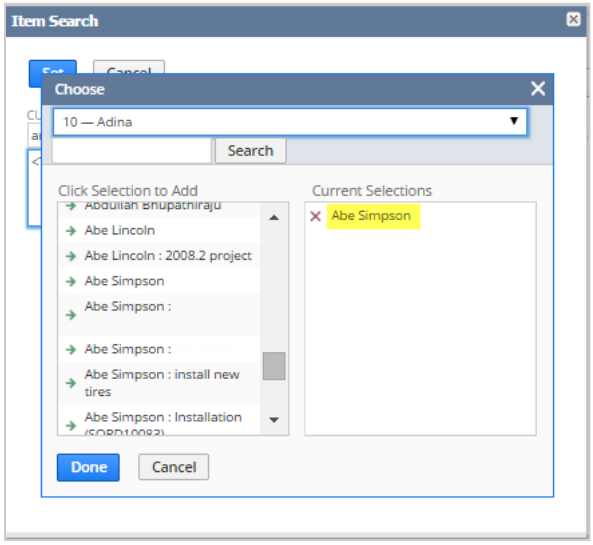
The first figure shows the Item Search record (Lists > Accounting > Items > Search).



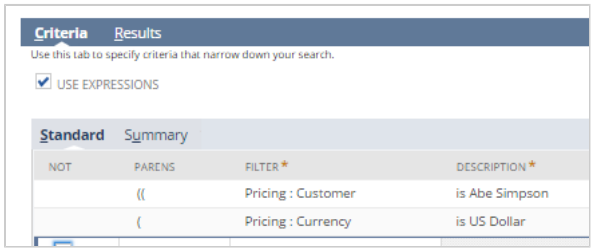
When **Pricing Fields...** is selected, a popup appears with all search fields that are available on the Pricing record. (See figure below.)



When you select the Customer field, another popup opens allowing you to select one or more customers. In the code sample, customer **121** is specified. In the UI, this customer appears as Abe Simpson. (See figure below.)



The following figure shows how Item search / pricing join filtering criteria appear in the UI.



Searching for an Item ID

The following search sample returns an array of item search results matched by the "Item ID" keyword. You can then iterate through each result and call nlobjSearchResult.getId() to get the internal item ID.

```
var x = nlapiSearchRecord('item', null, new nlobjSearchFilter('itemid', null, 'haskeywords', itemidvalue));
```

Searching for Duplicate Records

In the course of doing business, it is common to have more than one record created for the same contact, customer, vendor, or partner. In both the UI and in SuiteScript you can find all duplicate records after your NetSuite administrator has enabled the Duplicate Detection & Merge feature (Setup > Company > Enable Features, on the Company subtab, Data Management section).

In SuiteScript, a search for duplicate records is executed using the `nlapiSearchDuplicate(type, fields, id)` function. For the definition of this API, as well as a code sample, see the help topic nlapiSearchDuplicate(type, fields, id).

Note: For general information about searching for duplicate records in NetSuite, see the help topic Duplicate Record Detection in the NetSuite Help Center.

Performing Global Searches

Using global search, you can find records from anywhere in your account data. In the UI, you enter your search keywords in the **Search** field in the upper right corner of any page.

In SuiteScript, global searches are executed using the nlapiSearchGlobal(keywords) function. For the definition of this API, as well as a code sample, see the help topic nlapiSearchGlobal(keywords).

Note: For general information about global searching in NetSuite, see the help topic Global Search in the NetSuite Help Center.

Searching CSV Saved Imports

You can use SuiteScript to search the CSV saved imports in an account. Executing this kind of search may be useful if you need to access import information for CSV saved imports that were packaged and installed in a new account.

For the **type** argument in nlapiSearchRecord(type, id, filters, columns), you will set **savedcsvimport**. For example:

var search = nlapiSearchRecord('savedcsvimport', null, null, null);

Note that savedcsvimport is not a true record type in NetSuite, as it cannot be manipulated in SuiteScript or in SOAP web services. The savedcsvimport type can be used only in search. The available filter and return values for savedcsvimport are:

- internalid
- name

- description

Using Formulas, Special Functions, and Sorting in Search

```
function searchRecords()
{
   //specify a formula column that displays the name as: Last Name, First Name (Middle Name)
   var name = new nlobjSearchColumn('formulatext');
   name.setFormula("{lastname}||', '||{firstname}||case when LENGTH({middlename})=0 then '' else ' ('||{middlename}||')' end");

   //now specify a numeric formula field
   var number = new nlobjSearchColumn('formulanumeric').setFormula("1234.5678");

   //now specify a numeric formula field and format the output using a special function
   var roundednumber = new nlobjSearchColumn('formulanumeric').setFormula("1234.5678").setFunction("round");

   //now specify a sort column (sort by internal ID ascending)
   var internalid = new nlobjSearchColumn('internalid').setSort(false /* bsortdescending */);

   var columns = [name, number, roundednumber, internalid];
   var filterHasMiddleName = new nlobjSearchFilter('middlename', null, 'isNotEmpty');

   var searchresults = nlapiSearchRecord('contact', null, filterHasMiddleName, columns);
   for (var i = 0; i < searchresults.length; i++)
   {
      //access the value using the column objects
      var contactname = searchresults[i].getValue(name);
      var value = searchresults[i].getValue(number);
      var valuerounded = searchresults[i].getValue(roundednumber);
   }
}
```

Using Summary Filters in Search

```
function searchRecords()
{
   //perform a summary search: return all sales orders total amounts by customer for those with total sales > 1000
   var filter = new nlobjSearchFilter('amount', null, 'greaterThan', 1000).setSummaryType('sum');
   var entity = new nlobjSearchColumn('entity', null, 'group');
   var amount = new nlobjSearchColumn('amount', null, 'sum');

   var searchresults = nlapiSearchRecord('salesorder', null, filter, [entity, amount]);
   for (var i = 0; i < searchresults.length; i++)
   {
      //access the values this time using the name and summary type
      var entity = searchresults[i].getValue('entity', null, 'group');
      var entityName = searchresults[i].getText('entity', null, 'group');
      var amount = searchresults[i].getValue('amount', null, 'sum');
   }
}
```

SuiteScript 1.0 Supported Search Operators, Summary Types, and Date Filters

This section lists all NetSuite field types that support SuiteScript search, as well as the operators that can be used on each type. Also listed are supported search summary types and search date filters.

See the following sections for detailed reference information:

- SuiteScript 1.0 Search Operators
- SuiteScript 1.0 Search Summary Types
- SuiteScript 1.0 Search Date Filters

SuiteScript 1.0 Search Operators

Note: The content in this help topic pertains to all versions of SuiteScript. Be aware that currently it may only include links or examples for SuiteScript 1.0.

The following table lists each field type and its supported search operator.

Search Operator	List/Record	Currency, Decimal Number, Time of Day	Date	Check Box	Document, Image	Email Address, Free-Form Text, Long Text, Password, Percent, Phone Number, Rich Text, Text Area,	Multi Select
after	—	—	✓	—	—	—	—
allof	—	—	—	—	—	—	✓
any	—	✓	—	—	—	✓	—
anyof	✓	—	—	—	✓	—	✓
before	—	—	✓	—	—	—	—
between	—	✓	—	—	—	—	—
contains	—	—	—	—	—	✓	—
doesnotcontain	—	—	—	—	—	✓	—
doesnotstartwith	—	—	—	—	—	✓	—
equalto	—	✓	—	✓	—	✓	—
greaterthan	—	✓	—	—	—	—	—
greaterthanorequalto	—	✓	—	—	—	—	—
haskeywords	—	—	—	—	—	✓	—
is	—	—	—	✓	—	✓	—
					—		
					—		
isempty	—	✓	✓	—	—	✓	—
isnot	—	—	—	—	—	✓	—

Search Operator	List/Record	Currency, Decimal Number, Time of Day	Date	Check Box	Document, Image	Email Address, Free-Form Text, Long Text, Password, Percent, Phone Number, Rich Text, Text Area,	Multi Select
isnotempty	—	✓	✓	—	—	✓	—
lessthan	—	✓	—	—	—	—	—
lessthanorequalto	—	✓	—	—	—	—	—
noneof	✓	—	—	—	✓	—	✓
notafter	—	—	✓	—	—	—	—
notallof	—	—	—	—	—	—	✓
notbefore	—	—	✓	—	—	—	—
notbetween	—	✓	—	—	—	—	—
notequalto	—	✓	—	—	—	—	—
notgreaterthan	—	✓	—	—	—	—	—
notgreaterthanorequalto	—	✓	—	—	—	—	—
notlessthan	—	✓	—	—	—	—	—
notlessthanorequalto	—	✓	—	—	—	—	—
noton	—	—	✓	—	—	—	—
notonorafter	—	—	✓	—	—	—	—
notonorbefore	—	—	✓	—	—	—	—
notwithin	—	—	✓	—	—	—	—
on	—	—	✓	—	—	—	—
onorafter	—	—	✓	—	—	—	—
onorbefore	—	—	✓	—	—	—	—
startswith	—	—	—	—	—	✓	—
within	—	—	✓	—	—	—	—

Important: The onOrAfter, onOrBefore, notOnOrAfter and notOnOrBefore operators only consider the date you specify in your request, and ignore the time you specify. If you need to specify not only the date but also the time in your search, use the after or before operators. Note, however, when you use these operators, the exact time specified in your request is not part of the result set.

SuiteScript 1.0 Search Summary Types

Note: The content in this help topic pertains to all versions of SuiteScript. Be aware that currently it may only include links or examples for SuiteScript 1.0.

The following table lists the summary types that can be applied to organize your search results. Note that these summary types are available on the Results tab in the UI.

Summary Type	Type Internal ID	Purpose	Example
Group	group	Rolls up search results under the column to which you apply this type.	In a search for sales transactions, you can group the transactions found by customer name.
Count	count	Counts the number of results found that apply to this column.	In a search of items purchased by customers, you can view a count of the number of items purchased by each customer.
Sum	sum	Sums search results.	In a search of purchases this period, you can total the Amount column on your results.
Minimum	min	Shows the minimum amount found in search results.	In a search of sales transactions by sales rep, you can show the minimum amount sold in the transaction.
Maximum	max	Shows the maximum amount found in search results.	In a customer search by partner, you can show the maximum amount of sales by each partner.
Average	avg	Calculates the average amount found in your search results.	In an employee search, you can average the amounts of your employees' company contributions.

SuiteScript 1.0 Search Date Filters

Note: The content in this help topic pertains to all versions of SuiteScript. Be aware that currently it may only include links or examples for SuiteScript 1.0.

The following table lists all supported search date filters. Values are not case sensitive. These values correspond to selectors used in SuiteAnalytics. For additional information about these values, see the help topics Period Selectors, Date Range Selectors, and Date As Of Selectors.

Note: Only the yesterday, today, and tomorrow filters are available in SuiteScript.

fiscalHalfBeforeLast	lastYearToDate	previousOneQuarter	thisBusinessWeek
fiscalHalfBeforeLastToDate	monthAfterNext	previousOneWeek	thisFiscalHalf
fiscalQuarterBeforeLast	monthAfterNextToDate	previousOneYear	thisFiscalHalfToDate
fiscalQuarterBeforeLastToDate	monthBeforeLast	previousRollingHalf	thisFiscalQuarter
fiscalYearBeforeLast	monthBeforeLastToDate	previousRollingQuarter	thisFiscalQuarterToDate
fiscalYearBeforeLastToDate	nextBusinessWeek	previousRollingYear	thisFiscalYear
fiveDaysAgo	nextFiscalHalf	sameDayFiscalQuarterBeforeLast	thisFiscalYearToDate
fiveDaysFromNow	nextFiscalQuarter	sameDayFiscalYearBeforeLast	thisMonth
fourDaysAgo	nextFiscalYear	sameDayLastFiscalQuarter	thisMonthToDate
fourDaysFromNow	nextFourWeeks	sameDayLastFiscalYear	thisRollingHalf
fourWeeksStartingThisWeek	nextMonth	sameDayLastMonth	thisRollingQuarter
lastBusinessWeek	nextOneHalf	sameDayLastWeek	thisRollingYear
lastFiscalHalf	nextOneMonth	sameDayMonthBeforeLast	thisWeek
lastFiscalHalfOneFiscalYearAgo	nextOneQuarter	sameDayWeekBeforeLast	thisWeekToDate
lastFiscalHalfToDate	nextOneWeek	sameFiscalHalfLastFiscalYear	thisYear

lastFiscalQuarter	nextOneYear	sameFiscalHalfLastFiscalYearToDate	thisYearToDate
lastFiscalQuarterOneFiscalYearAgo	nextWeek	sameFiscalQuarterFiscalYearBeforeLast	threeDaysAgo
lastFiscalQuarterToDate	ninetyDaysAgo	sameFiscalQuarterLastFiscalYear	threeDaysFromNow
lastFiscalQuarterTwoFiscalYearsAgo	ninetyDaysFromNow	sameFiscalQuarterLastFiscalYearToDate	threeFiscalQuartersAgo
lastFiscalYear	oneYearBeforeLast	sameMonthFiscalQuarterBeforeLast	threeFiscalQuartersAgoToDate
lastFiscalYearToDate	previousFiscalQuartersLastFiscalYear	sameMonthFiscalYearBeforeLast	threeFiscalYearsAgo
lastMonth	previousFiscalQuartersThisFiscalYear	sameMonthLastFiscalQuarter	threeFiscalYearsAgoToDate
lastMonthOneFiscalQuarterAgo	previousMonthsLastFiscalHalf	sameMonthLastFiscalQuarterToDate	threeMonthsAgo
lastMonthOneFiscalYearAgo	previousMonthsLastFiscalQuarter	sameMonthLastFiscalYear	threeMonthsAgoToDate
lastMonthToDate	previousMonthsLastFiscalYear	sameMonthLastFiscalYearToDate	today
lastMonthTwoFiscalQuartersAgo	previousMonthsSameFiscalHalfLastFiscalYear	sameWeekFiscalYearBeforeLast	todayToEndOfThisMonth
lastMonthTwoFiscalYearsAgo	previousMonthsSameFiscalQuarterLastFiscalYear	sameWeekLastFiscalYear	tomorrow
lastRollingHalf	previousMonthsThisFiscalHalf	sixtyDaysAgo	twoDaysAgo
lastRollingQuarter	previousMonthsThisFiscalQuarter	sixtyDaysFromNow	twoDaysFromNow
lastRollingYear	previousMonthsThisFiscalYear	tenDaysAgo	weekAfterNext
lastWeek	previousOneDay	tenDaysFromNow	weekAfterNextToDate
lastWeekToDate	previousOneHalf	thirtyDaysAgo	weekBeforeLast
lastYear	previousOneMonth	thirtyDaysFromNow	weekBeforeLastToDate
			yesterday

Deprecated Search Date Filters

Important: The following search date filters are deprecated as of Version 2015 Release 1. These values are still supported in existing scripts. For new scripts, please use the arguments listed in the above table.

daysagoxx	startOfLastFiscalHalfOneFiscalYearAgo	startOfSameFiscalHalfLastFiscalYear
daysfromnowxx	startOfLastFiscalQuarter	startOfSameFiscalQuarterLastFiscalYear
lastFiscalHalfOneYearAgo	startOfLastFiscalQuarterOneFiscalYearAgo	startOfSameHalfLastFiscalYear
lastFiscalQuarterOneYearAgo	startOfLastFiscalYear	startOfSameMonthLastFiscalQuarter
lastMonthOneQuarterAgo	startOfLastHalfOneYearAgo	startOfSameMonthLastFiscalYear
lastMonthOneYearAgo	startOfLastMonth	startOfSameQuarterLastFiscalYear
lastMonthTwoQuartersAgo	startOfLastMonthOneFiscalQuarterAgo	startOfTheHalfBeforeLast
lastMonthTwoYearsAgo	startOfLastMonthOneFiscalYearAgo	startOfTheMonthBeforeLast
lastQuarterTwoYearsAgo	startOfLastMonthOneQuarterAgo	startOfTheQuarterBeforeLast
monthsagoxx	startOfLastMonthOneYearAgo	startOfTheWeekBeforeLast
monthsfromnowxx	startOfLastQuarterOneYearAgo	startOfTheYearBeforeLast
previousMonthsSameFiscalHalfLastYear	startOfLastRollingHalf	startOfThisBusinessWeek
previousMonthsSameFiscalQuarterLastFiscalYear	startOfLastRollingQuarter	startOfThisFiscalHalf
previousQuartersLastFiscalYear	startOfLastRollingYear	startOfThisFiscalQuarter
previousQuartersThisFiscalYear	startOfLastWeek	startOfThisFiscalYear
quartersagoxx	startOfMonthBeforeLast	startOfThisMonth
quartersfromnowxx	startOfNextBusinessWeek	startOfThisWeek
sameHalfLastFiscalYear		startOfThisYear
sameHalfLastFiscalYearToDate		startOfWeekBeforeLast

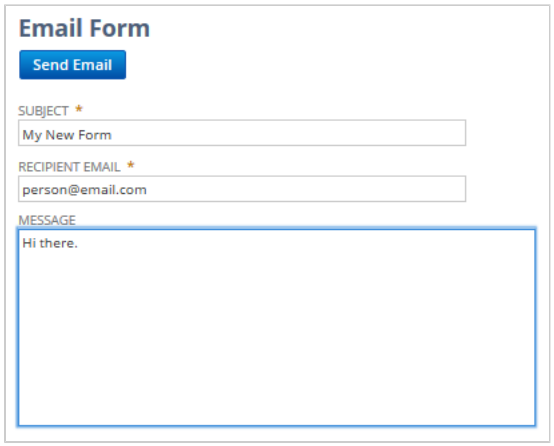
sameQuarterLastFiscalYear	startOfNextFiscalHalf	threeQuartersAgo
sameQuarterLastFiscalYearToDate	startOfNextFiscalQuarter	threeQuartersAgoToDate
startOfFiscalHalfBeforeLast	startOfNextFiscalYear	threeYearsAgo
startOfFiscalQuarterBeforeLast	startOfNextMonth	threeYearsAgoToDate
startOfFiscalYearBeforeLast	startOfNextWeek	weeksagoxx
startOfLastBusinessWeek	startOfNextYear	weeksfromnowxx
startOfLastFiscalHalf	startOfPreviousRollingHalf	yearsagoxx
	startOfPreviousRollingQuarter	yearsfromnowxx
	startOfPreviousRollingYear	

UI Objects Overview

SuiteScript UI Objects are a collection of objects that can be used as a UI toolkit for server scripts such as Suitelets and User Event Scripts. SuiteScript UI objects are generated on the server as HTML. They are then displayed in the browser and are accessible through client scripts.

Important: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). The NetSuite UI should only be accessed using SuiteScript APIs.

The following figure is an email form Suitelet built with UI objects. The form itself is represented by the nlobjForm UI object. The Subject, Recipient email, and Message fields are represented by the nlobjField UI object, and the Send Email button is represented by the nlobjButton UI object.



To learn more about working with UI objects, see these topics:

- SuiteScript 1.0 Creating Custom NetSuite Pages with UI Objects
- InlineHTML UI Objects
- Building a NetSuite Assistant with UI Objects

Email Form Code

```
/**
 * Build an email form Suitelet with UI objects. The Suitelet sends out an email
 * from the current user to the recipient email address specified on the form.
 */
function simpleEmailForm(request, response)
{
   if ( request.getMethod() == 'GET' )
   {
        var form = nlapiCreateForm('Email Form');
        var subject = form.addField('subject','text', 'Subject');
        subject.setLayoutType('normal','startcol');
        subject.setMandatory( true );
        var recipient = form.addField('recipient','email', 'Recipient email');
        recipient.setMandatory( true );
        var message = form.addField('message','textarea', 'Message');
        message.setDisplaySize( 60, 10 );
        form.addSubmitButton('Send Email');

```

```
    response.writePage(form);
    }
    else
    {
            var currentuser = nlapiGetUser();
            var subject = request.getParameter('subject')
            var recipient = request.getParameter('recipient')
            var message = request.getParameter('message')
        nlapiSendEmail(currentuser, recipient, subject, message);
    }
}
```

SuiteScript 1.0 Creating Custom NetSuite Pages with UI Objects

SuiteScript UI Objects encapsulate the UI elements necessary for building NetSuite-looking portlets, forms, fields, sublists, tabs, lists, columns, and assistant. Note that when developing a Suitelet with UI objects, you can also add custom fields with inline HTML.

Depending on the design and purpose of the custom UI, you can use either the nlobjForm UI object or nlobjList UI object as the basis. These objects encapsulate a scriptable NetSuite form and NetSuite list, respectively. You can then add a variety of scriptable UI elements to these objects to adopt the NetSuite look-and-feel. These elements can include fields (through nlobjField), buttons (through nlobjButton), tabs (through nlobjTab), and sublists (through nlobjSubList).

Important: When adding UI elements to an **existing** page, you must prefix the element name with **custpage**. This minimizes the occurrence of field/object name conflicts. For example, when adding a custom tab to a NetSuite entry form in a user event script, the name should follow a convention similar to **custpage** customtab or **custpage** mytab.

UI Objects and Suitelets

In Suitelet development, UI objects allow you to programmatically build custom NetSuite-looking pages. A blank nlobjForm object is created with nlapiCreateForm(title, hideNavbar). If you are building a custom assistant, a blank nlobjAssistant object is created with nlapiCreateAssistant(title, hideHeader). On the server, the Suitelet code adds fields, steps, tabs, buttons and sublists to the form and assistant objects.

The server defines the client script (if applicable) and sends the page to the browser. When the page is submitted, the values in these UI objects become part of the request and available to aid logic branching in the code.

For a basic example of a Suitelet built entirely of UI objects, see What Are Suitelets?. This section also provides the code used to create the Suitelet. To see examples of an assistant or "wizard" Suitelet built with UI objects, see Using UI Objects to Build an Assistant.

UI Objects and User Event Scripts

On entry forms and transaction forms, the nlobjForm object is accessed in user events scripts on which new fields are added on the server before the pages are sent to the browser. With the NetSuite nlobjForm object exposed, you can design user event scripts that manipulate most built-in NetSuite UI components (for example, fields, tabs, sublists).

Note: If you are not familiar with concept of NetSuite entry or transaction forms, see the help topic Custom Forms in the NetSuite Help Center.

The key to using user event scripts to customize a form during runtime is a second argument named `form` in the **before load** event. This optional argument is the reference to the entry/transaction form. You can use this to dynamically change existing form elements, or add new ones (see Enhancing NetSuite Forms with User Event Scripts).

Note: Sometimes the best solution for a customized workflow with multiple pages is a hybrid UI design, which encompasses both customized entry forms as well as Suitelets built with UI objects.

UI Object Descriptions and Examples

For details aboutt he UI objects supported by NetSuite, as well as code samples, see the help topic UI Objects.

InlineHTML UI Objects

SuiteScript UI objects make most of the NetSuite UI elements scriptable. However, they still may not lend themselves well to certain use cases. In these circumstances, developers can create custom UI elements by providing HTML that SuiteScript can render on a NetSuite page. These UI elements are known as "InlineHTML".

InlineHTML can be implemented in two ways:

1. Pure custom HTML with no SuiteScript UI objects
2. Hybrid of custom HTML and SuiteScript UI objects

The first approach, shown on the left side, requires you to provide all the HTML code you want to appear on the page, as if performing web designing on a blank canvas. The second approach, shown on the right, allows custom HTML to be embedded in a NetSuite page. Example code is available in the help section for Suitelets Samples.

A blog hosted within NetSuite can be created with a hybrid of inlineHTML and UI objects. A blog page can display blog entries in descending chronological order with "Read More" hyperlinks. Pagination should be used to handle the potentially large number of entries. Readers should also be able to read and leave comments. These requirements cannot be satisfied by standard NetSuite UI elements in a reader-friendly manner. However, rendering the blog entries' data (stored as custom records) in HTML and displaying it in SuiteScript UI objects as inlineHTML would satisfy this use case.

For more information, see the help topic Using Embedded Inline HTML in a Form.

Building a NetSuite Assistant with UI Objects

NetSuite UI Object Assistant Overview

You can use UI objects to build an assistant or "wizard" within NetSuite that has the same look-and-feel as other built-in NetSuite assistants. To build your own assistant, you will use objects such as nlobjAssistant, nlobjAssistantStep, nlobjFieldGroup, and nlobjField.

For examples that show some of the built-in assistants that are included with NetSuite, see the help topic Understanding NetSuite Assistants.

To learn to how programmatically construct your own assistant, see Using UI Objects to Build an Assistant.

Using UI Objects to Build an Assistant

From a UI perspective, the building blocks of most assistants you build are going to include a combination of the following: Steps, Field Groups, Fields, Buttons, Sublists.

To create this look, you will use objects such as nlobjAssistant, nlobjAssistantStep, nlobjFieldGroup, and nlobjField. The API documentation for each object and all object methods provides examples for building instances of each objects.

Also see these topics for additional information:

- Understanding the Assistant Workflow
- Using Redirection in an Assistant Workflow
- Assistant Components and Concepts
- UI Object Assistant Code Sample

Note that since your assistant is a **Suitelet**, after you have finished building the assistant, you can initiate the assistant by creating a custom menu link that contains the Suitelet URL. (See Running a Suitelet in NetSuite for details on creating custom menu links for Suitelets.)

Understanding the Assistant Workflow

If you have not done so already, please see Using UI Objects to Build an Assistant for information about the UI objects that are used to create an assistant.

In your SuiteScript code, there is an order in which many components of an assistant must be added. At a minimum you will:

1. **Create a new assistant**

a. Call nlapiCreateAssistant(title, hideHeader).
b. Add steps to the assistant. Use nlobjAssistant.addStep(name, label).
c. Define whether the steps must be completed sequentially or whether they can be completed in random order. Use nlobjAssistant.setOrdered(ordered).

2. **Build assistant pages**

   Add fields, field groups, and sublists to build assistant pages for each step.

   Note: In the context of an assistant, each step is considered a page.

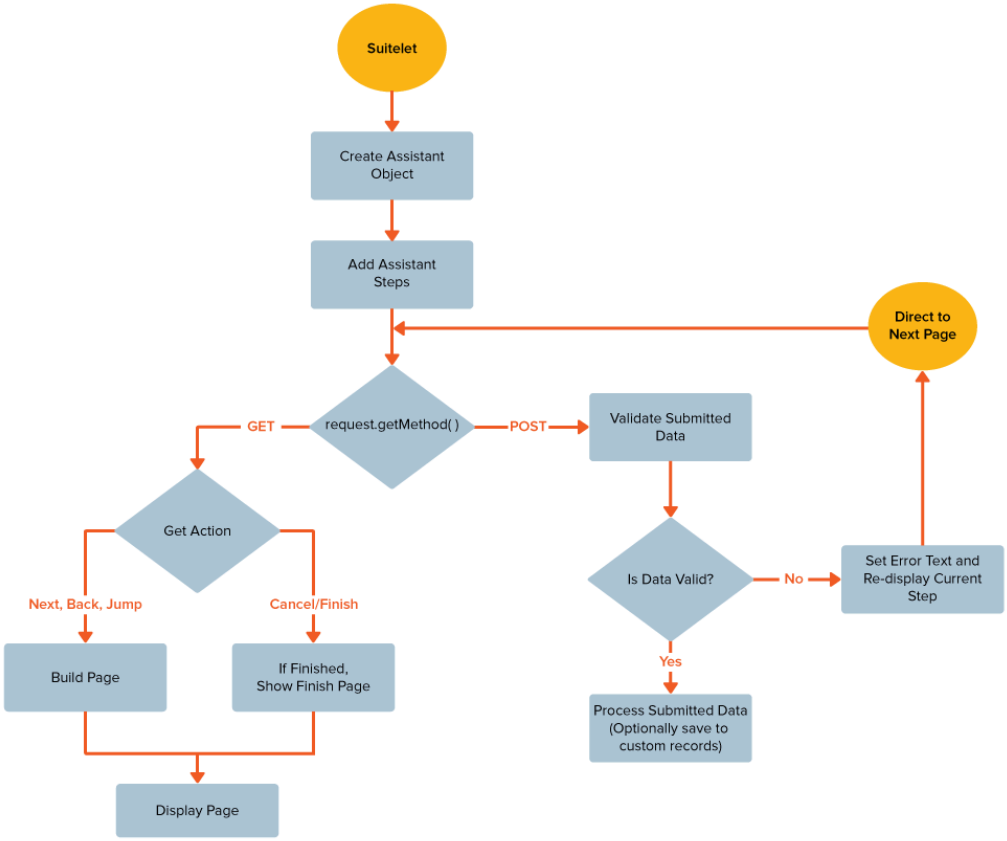
   In the assistant workflow diagram (below), see **Build Page** for a list of methods that can be used to build a page.

3. **Process assistant pages**

   In your Suitelet, construct pages in response to a user's navigation of the assistant. At a minimum you will render a specific assistant step/page using a GET request. Then you will process that page in the POST request, before then redirecting the user to another step in the assistant.

   For example, this is where you would update a user's account based on data the user has entered in the assistant.

   The following flowchart provides an overview of a suggested assistant design.



Using Redirection in an Assistant Workflow

From within a custom assistant you can redirect users to:

- a new record/page within NetSuite (for example, to a new Employee or Contact page in NetSuite)
- the start page of a built-in NetSuite assistant (for example, the Import Assistant or the Web Site Assistant)
- another custom assistant

  To link users to another NetSuite page, built-in assistant, or custom assistant, and then return them back to the originating assistant, you must set the value of the customwhence parameter in the redirect URL to originating assistant. The value of customwhence will consist of the scriptId and deploymentId of the originating custom assistant Suitelet.

Example

The following sample shows a helper function that appears at the end of the Assistant code sample (see UI Object Assistant Code Sample). Notice that in this function, the value of the customwhence parameter in the URL is the scriptId and deplomentId of the custom assistant that you originally started with. To link users out of the originating assistant, and then return them back to this assistant after they have completed other tasks, you must append the customwhence parameter to the URL you are redirecting to.

```
function getLinkoutURL( redirect, type )
{
   var url = redirect;
   if ( type == "record" )
      url = nlapiResolveURL('record', redirect);
   url += url.indexOf('?') == -1 ? '?' : '&';
   var context = nlapiGetContext();
   url += 'customwhence='+ escape(nlapiResolveURL('suitelet',context.getScriptId(),
     context.getDeploymentId()))
   return url;
}
```

Note: If you redirect users to a built-in assistant or to another custom assistant, be aware that they will not see the "Finish" page on the assistant they have been linked out to. After they complete the assistant they have been linked to, they will be redirected back to the page where they left off in the original assistant.

Assistant Components and Concepts

The following information pertains to the UI components used to build an assistant. Also described are the concepts associated with state management and error handling.

- Steps
- Field Groups
- Fields
- Sublists
- Buttons
- State Management
- Error Handling

Steps

Create a step by calling nlobjAssistant.addStep(name, label), which returns a reference to the nlobjAssistantStep object.

At a minimum, every assistant will include steps, since steps are what define each page of the assistant. Whether the steps must be completed sequentially or in a more random order is up to you. Enforced sequencing of steps be will defined by the nlobjAssistant.setOrdered(ordered) method.

The placement of your steps (vertically along the left panel, or horizontally across the top of the assistant) will also be determined by you. Also note that you can add helper text for each step using the nlobjAssistantStep.setHelpText(help) method.

Note: Currently there is no support for sub-steps.

Field Groups

Create a field group by calling nlobjAssistant.addFieldGroup(name, label), which returns a reference to the nlobjFieldGroup object.

In the UI, field group are collapsible groups of fields that can be displayed in a one-column or two-column format. The following snippet shows how to use the nlobjField.setLayoutType(type, breaktype) method to start a second column in a field group.

```
assistant.addFieldGroup("companyinfo", "Company Information");
assistant.addField("companyname", "text", "Company Name", null, "companyinfo")
assistant.addField("legalname", "text", "Legal Name", null, "companyinfo")
assistant.addField("shiptoattention", "text", "Ship To Attention", null, "companyinfo")
assistant.addField("address1", "text", "Address 1", null, "companyinfo"). setLayoutType("normal", "startcol");
assistant.addField("address2", "text", "Address 2", null, "companyinfo");
assistant.addField("city", "text", "City", null, "companyinfo");
```

Note that field groups do not have to be collapsible. They can appear as a static grouping of fields. See nlobjFieldGroup.setCollapsible(collapsible, hidden) for more information about setting collapsibility.

Fields

Create a field by calling nlobjAssistant.addField(name, type, label, source, group), which returns a reference to the nlobjField object.

Fields are added to the **current** step on a per-request basis. For example, as the sample below shows, in a GET request, if the user's current step is a step called "companyinformation", (meaning the user has navigated to a step/page with the internal ID "companyinformation"), the page that renders will include a field group and six fields within the group.

```
var step = assistant.getCurrentStep();
if (step.getName() == "companyinformation")
{
assistant.addFieldGroup("companyinfo", "Company Information");
assistant.addField("companyname", "text", "Company Name", null, "companyinfo")
assistant.addField("legalname", "text", "Legal Name", null, "companyinfo")
assistant.addField("shiptoattention", "text", "Ship To Attention", null, "companyinfo")
assistant.addField("address1", "text", "Address 1", null, "companyinfo").setLayoutType("normal", "startcol");
assistant.addField("address2", "text", "Address 2", null, "companyinfo");
assistant.addField("city", "text", "City", null, "companyinfo");
}
```

Note that all nlobjField APIs can be used with the fields returned from the addField(...) method. Also, fields marked as required are respected by the assistant. Users cannot click through to the next page if fields that are required on the current page do not contain a value.

Important: The nlobjField.setLayoutType(type, break) method can be used to place a column break in an assistant. Be aware that only the first column break that is encountered will be honored. Currently assistants support only single or two column layouts. You cannot set more than one column break.

Sublists

Create a sublist by calling nlobjAssistant.addSubList(name, type, label), which returns a reference to the nlobjSubList object.

If you want to add a sublist to an assistant, be aware that only sublists of type **inlineeditor** are supported. Also note that sublists on an assistant are always placed below all other elements on the page.

Buttons

You do not need to programmatically add button objects to an assistant. Buttons are automatically generated through the nlobjAssistant object.

Depending on which page you are on, the following buttons appear: Next, Back, Cancel, Finish. When users reach the final step in an assistant, the Next button no longer displays, and the Finish button appears. Button actions need to be communicated using the request using nlobjAssistant.getLastAction().

Important: The addition of custom buttons are not currently supported on assistants.

State Management

Assistants support data and state tracking across pages within the **same** session until the assistant is completed by the user (at which point the assistant is reset when the "Finished" page displays).

Field data tracking is automatically saved in assistants. For example, if a user revisits a page using the Back button, the previously entered data will be automatically displayed.

Every time a page is submitted, all the fields will be automatically tracked and when the page is displayed. If the user did not explicitly set a value for a field or on a sublist, then the field(s) and sublist(s) will be populated from data entered by the user the last time they submitted that page.

Note: If state/data tracking needs to be preserved across sessions, you should use custom records or tables to record this information.

Note that an SSS_NOT_YET_SUPPORTED_ERROR is thrown if the assistant is used on an "Available Without Login" (external) Suitelet. (See the help topic Setting Available Without Login for information about this Suitelet deployment option.) Session-based state tracking used in custom assistants requires a session to exist across requests.

Finally, multiple Suitelet deployments should **not** be used to manage the pages within an assistant, since data/state tracking is tied to the Suitelet instance. Developers should create one Suitelet deployment per assistant.

Error Handling

If an error occurs on a step, the assistant displays two error indicators. The first indicator is a red bar that appears directly below the step. The second indicator is the html you pass to nlobjAssistant.setError(html).

UI Object Assistant Code Sample

The following is an implementation of a setup assistant with a few basic steps. State is managed throughout the life of the user's session. In summary, this script shows you how to:

1. Create the assistant.
2. Create steps.
3. Set the user's first step.
4. Build pages for each step.
5. Process data entered by the user.

   Note that this sample can be run in a NetSuite account. To do so, you must create a .js file for the sample code below. Then you must create a new Suitelet script record and a script deployment. Do not select the Available Without Login deployment option on the Script Deployment page, otherwise the Suitelet will not run. (For general details on creating a Script record, setting values on the Script Deployment page, and creating a custom menu link for a Suitelet, see Running a Suitelet in NetSuite.)

   After the script is deployed, you can launch the assistant Suitelet by clicking the Suitelet URL on the Script Deployment page. You can also create a tasklink for the Suitelet and launch the Suitelet as a custom menu item.

   Also notice that this sample has all three types of "link outs," as defined in Using Redirection in an Assistant Workflow: one link out to add employees, one to the Import Assistant, and one to another custom assistant. Notice how the **customwhence** parameter is constructed and appended to the target URL that you are linking out to. Also notice how nlobjAssistant.sendRedirect(response) is used to ensure that customwhence is respected.

   > **Important:** If your browser is inserting scroll bars in this code sample, maximize your browser window, or expand the main frame that this sample appears is.

```
/**
* Implementation of a simple setup assistant with support for multiple setup steps and sequential or on demand step traversal.
* State is managed throughout the life of the user's session but is not persisted across sessions. Doing so would
* require writing this information to a custom record.
*
* @param request request object
* @param response response object
*/
function showAssistant(request, response)
{
   /* first create assistant object and define its steps. */
   var assistant = nlapiCreateAssistant("Small Business Setup Assistant", true);
   assistant.setOrdered( true );
   nlapiLogExecution( 'DEBUG', "Create Assistant ", "Assistant Created" );

   assistant.addStep('companyinformation', 'Setup Company Information').setHelpText("Setup your
     <b>important</b> company information in the fields below.");
   assistant.addStep('companypreferences', 'Setup Company Preferences').setHelpText("Setup your
     <b>important</b> company preferences in the fields below.");
   assistant.addStep('enterlocations', 'Enter Locations').setHelpText("Add Locations to your account.
      You can create a location record for each of your company's locations. Then you can track employees
      and transactions by location..");
   assistant.addStep('enteremployees', 'Enter Company Employees').setHelpText("Enter your
     company employees.");
   assistant.addStep('importrecords', 'Import Records').setHelpText("Import your initial company data.");
   assistant.addStep('configurepricing', 'Configure Pricing' ).setHelpText("Configure your item pricing.");
   assistant.addStep('summary', 'Summary Information').setHelpText("Summary of your Assistant
     Work.<br> You have made the following choices to configure your NetSuite account.");


   /* handle page load (GET) requests. */
   if (request.getMethod() == 'GET')
   {
```

```
    /*.Check whether the assistant is finished */
    if ( !assistant.isFinished() )
    {
       // If initial step, set the Splash page and set the intial step
        if ( assistant.getCurrentStep() == null )
                  {
          assistant.setCurrentStep(assistant.getStep( "companyinformation") );

          assistant.setSplash("Welcome to the Small Business Setup Assistant!", "<b>What
           you'll be doing</b><br>The Small Business Setup Assistant will walk you
           through the process of configuring your NetSuite account for your initial use..",
            "<b>When you finish</b><br>your account will be ready for you to use to
           run your business.");
       }
       var step = assistant.getCurrentStep();

       // Build the page for a step by adding fields, field groups, and sublists to the assistant
       if (step.getName() == "companyinformation")
       {
          assistant.addField('orgtypelabel','label','What type of organization are
             you?').setLayoutType('startrow');
          assistant.addField('orgtype', 'radio','Business To Consumer',
            'b2c').setLayoutType('midrow');
          assistant.addField('orgtype', 'radio','Business To Business','b2b').setLayoutType('midrow');
          assistant.addField('orgtype', 'radio','Non-Profit','nonprofit').setLayoutType('endrow');
          assistant.getField( 'orgtype', 'b2b' ).setDefaultValue( 'b2b' );

          assistant.addField('companysizelabel','label','How big is your organization?');
          assistant.addField('companysize', 'radio','Small (0-99 employees)', 's');
          assistant.addField('companysize', 'radio','Medium (100-999 employees)','m');
          assistant.addField('companysize', 'radio','Large (1000+ employees)','l');

          assistant.addFieldGroup("companyinfo", "Company Information");
          assistant.addField("companyname", "text", "Company Name", null,
             "companyinfo").setMandatory( true );
          assistant.addField("legalname", "text", "Legal Name", null, "companyinfo").setMandatory
              ( true );
          assistant.addField("shiptoattention", "text", "Ship To Attention", null,
             "companyinfo").setMandatory( true );
          assistant.addField("address1", "text", "Address 1", null,
            "companyinfo").setLayoutType("normal", "startcol");
          assistant.addField("address2", "text", "Address 2", null, "companyinfo");
          assistant.addField("city", "text", "City", null, "companyinfo");
          assistant.getField("legalname").setHelpText("Enter a Legal Name if it differs from
            your company name");
          assistant.getField("shiptoattention").setHelpText("Enter the name of someone
             who can sign for packages or important documents. This is important
          because otherwise many package carriers will not deliver to your corporate address");
          assistant.addFieldGroup("taxinfo", "Tax Information").setCollapsible(true /* collapsable */,
              true /* collapsed by default */);
          assistant.addField("employeeidnumber", "text", "Employee Identification Number (EIN)",
             null, "taxinfo").setHelpText("Enter the EID provided to you by the state or
           federal government");
          assistant.addField("taxidnumber", "text", "Tax ID Number", null,
          "taxinfo").setHelpText("Enter the Tax ID number used when you file your payroll
            and sales taxes");
          assistant.addField("returnmailaddress", "textarea", "Return Mail Address", null,
           "taxinfo").setHelpText("In the rare event someone returns your products, enter
             the mailing address.");

       }

       if (step.getName() == "companypreferences")
       {
          nlapiLogExecution( 'DEBUG', "Company Preferences ",  "Begin Creating Page" );

          assistant.addFieldGroup("companyprefs", "Company Preferences");
          var firstDayOfWeek = assistant.addField("firstdayofweek", "select", "First Day of Week",
             null, "companyprefs");
          var stateAbbrs = assistant.addField("abbreviatestates", "checkbox", "Use State Abbreviations
           in Addresses", null, "companyprefs");
          var customerMessage = assistant.addField("customerwelcomemessage", "text", "Customer
            Center Welcome Message", null, "companyprefs");
```

```
            customerMessage.setMandatory( true );

            assistant.addFieldGroup("accountingprefs", "Accounting Preferences").setCollapsible(true,
               true );
            var accountNumbers = assistant.addField("accountnumbers", "checkbox", "Use Account
              Numbers", null, "accountingprefs");
            var creditLimitDays = assistant.addField("credlimdays", "integer", "Days Overdue
             for Warning or Hold", null, "accountingprefs");
            var expenseAccount = assistant.addField("expenseaccount", "select", "Default
               Expense Account", 'account', "accountingprefs");
            customerMessage.setMandatory( true );

            assistant.addField('customertypelabel','label','Please Indicate Your Default Customer
                Type?' );
            assistant.addField( 'customertype', 'radio', 'Individual', 'i' );
            assistant.addField('customertype', 'radio', 'Company', 'c' );

            // get the select options for First Day of Week
            nlapiLogExecution( 'DEBUG', "Load Configuration ",  "Company Preferences" );
            var compPrefs = nlapiLoadConfiguration( 'companypreferences' );

            var firstDay = compPrefs.getField( 'FIRSTDAYOFWEEK' );
            nlapiLogExecution( 'DEBUG', "Create Day of Week Field ",  compPrefs.getFieldText(
               'FIRSTDAYOFWEEK')  );

            try
            {
               var selectOptions = firstDay.getSelectOptions();
            }
            catch( error )
            {
               assistant.setError( error );
            }

            if( selectOptions != null)
            {
               nlapiLogExecution( 'DEBUG', "Have Select Options ",  selectOptions[0].getText() );

               // add the options to the UI field
               for (var i = 0; i < selectOptions.length; i++)
               {
                  firstDayOfWeek.addSelectOption( selectOptions[i].getId(),
                     selectOptions[i].getText() );
               }
            }

            // set the default values based on the product default
            stateAbbrs.setDefaultValue( compPrefs.getFieldValue( 'ABBREVIATESTATES' ) );
            customerMessage.setDefaultValue( compPrefs.getFieldValue(
               'CUSTOMERWELCOMEMESSAGE' ) );


         }
         else if (step.getName() == "enterlocations")
         {
            var sublist = assistant.addSubList("locations", "inlineeditor", "Locations");

            sublist.addField("name", "text", "Name");
            sublist.addField("tranprefix", "text", "Transaction Prefix");
            sublist.addField("makeinventoryavailable", "checkbox", "Make Inventory Available");
            sublist.addField("makeinventoryavailablestore", "checkbox", "Make Inventory Available in
               Web Store");

            sublist.setUniqueField("name");
         }

         else if (step.getName() == "enteremployees")
         {
            // get the host
            var host = request.getURL().substring(0, ( request.getURL().indexOf('.com') + 4) );

            assistant.addFieldGroup("enteremps", "Enter Employees");
            assistant.addField("employeecount", "integer", "Number of Employees in Company", null,
```

```
            "enteremps").setMandatory( true );
         assistant.addField("enterempslink", "url", "", null, "enteremps" ).setDisplayType
            ( "inline" ).setLinkText( "Click Here to Enter Your Employees").setDefaultValue( host
            + getLinkoutURL( 'employee', 'record') );
      }

      else if (step.getName() == "importrecords")
      {
         var host = request.getURL().substring(0, ( request.getURL().indexOf('.com') + 4) );

         assistant.addFieldGroup("recordimport", "Import Data");
         assistant.addField("recordcount", "integer", "Number of Records to Import", null,
            "recordimport").setMandatory( true );
         assistant.addField("importlink", "url", "", null, "recordimport" ).setDisplayType
         ( "inline" ).setLinkText( "Click Here to Import Your Data").setDefaultValue( host +
         getLinkoutURL( "/app/setup/assistants/nsimport/importassistant.nl" ) );
      }

      else if (step.getName() == "configurepricing")
      {
         var host = request.getURL().substring(0, ( request.getURL().indexOf('.com') + 4) );

         assistant.addFieldGroup("pricing", "Price Configuration");
         assistant.addField("itemcount", "integer", "Number of Items to Configure", null,
            "pricing").setMandatory( true );


      /* When users click the 'Click Here to Configure Pricing' link, they will be taken to
      * another custom assistant Suitelet that has a script ID of 47 and a deploy ID of 1. Note
      * that the code for the "link out" assistant is not provided in this sample.
      *
      * Notice the use of the getLinkoutURL helper function, which sets the URL
      * customwhence parameter so that after users finish the with the "link out" assistant,
      * they will be redirected back to this (the originating) assistant.
      */
         assistant.addField("importlink", "url", "", null, "pricing" ).setDisplayType
            ( "inline").setLinkText( "Click Here to Configure Pricing").setDefaultValue( host +
            getLinkoutURL( "/app/site/hosting/scriptlet.nl?script=47&deploy=1" ) );
      }

      else if (step.getName() == "summary")
      {


         assistant.addFieldGroup("companysummary", "Company Definition Summary");
         assistant.addField('orgtypelabel','label','What type of organization are you?', null,
            'companysummary' );
         assistant.addField('orgtype', 'radio','Business To Consumer', 'b2c',
            'companysummary' ).setDisplayType( 'inline');
         assistant.addField('orgtype', 'radio','Business To Business','b2b',
           'companysummary' ).setDisplayType( 'inline');
         assistant.addField('orgtype', 'radio','Non-Profit','nonprofit',
         'companysummary' ).setDisplayType( 'inline');

         assistant.addField('companysizelabel','label','How big is your organization?', null,
         'companysummary' );
         assistant.addField('companysize', 'radio','Small (0-99 employees)', 's',
            'companysummary' ).setDisplayType( 'inline');;
         assistant.addField('companysize', 'radio','Medium (100-999 employees)','m',
            'companysummary' ).setDisplayType( 'inline');;
         assistant.addField('companysize', 'radio','Large (1000+ employees)','l',
             'companysummary' ).setDisplayType( 'inline');;
         assistant.addField("companyname", "text", "Company Name", null,
            "companysummary").setDisplayType( 'inline');
         assistant.addField("city", "text", "City", null, "companysummary").setDisplayType('inline');
         assistant.addField("abbreviatestates", "checkbox", "Use State Abbreviations in Addresses",
            null, "companysummary").setDisplayType( 'inline');
         assistant.addField("customerwelcomemessage", "text", "Customer Center Welcome
            Message", null, "companysummary").setDisplayType( 'inline');


         // get previously submitted steps
         var ciStep = assistant.getStep( 'companyinformation' );
```

```
            var cpStep = assistant.getStep( 'companypreferences' );

            // get field values from previously submitted steps
            assistant.getField( 'orgtype', 'b2b' ).setDefaultValue( ciStep.getFieldValue( 'orgtype' ) );
            assistant.getField( 'companysize', 's' ).setDefaultValue( ciStep.getFieldValue
               ( 'companysize' ) );
            assistant.getField( 'companyname' ).setDefaultValue( ciStep.getFieldValue
               ( 'companyname' ) );
            assistant.getField( 'city' ).setDefaultValue( ciStep.getFieldValue( 'city' ) );
            assistant.getField( 'abbreviatestates' ).setDefaultValue( cpStep.getFieldValue
               ( 'abbreviatestates' ) );
            assistant.getField( 'customerwelcomemessage' ).setDefaultValue
               ( cpStep.getFieldValue( 'customerwelcomemessage' ) );


         }

      }
      response.writePage(assistant);
   }
   /* handle user submit (POST) requests. */
   else
   {

      assistant.setError( null );

      /* 1. if they clicked the finish button, mark setup as done and redirect to assistant page */
      if (assistant.getLastAction() == "finish")
      {
         assistant.setFinished( "You have completed the Small Business Setup Assistant." );

         assistant.sendRedirect( response );
      }
      /* 2. if they clicked the "cancel" button, take them to a different page (setup tab) altogether as
         appropriate. */
      else if (assistant.getLastAction() == "cancel")
      {
         nlapiSetRedirectURL('tasklink', "CARD_-10");
      }
      /* 3. For all other actions (next, back, jump), process the step and redirect to assistant page. */
      else
      {

         if (assistant.getLastStep().getName() == "companyinformation" && assistant.getLastAction()
            == "next" )
         {
            // update the company information page
            var configCompInfo = nlapiLoadConfiguration( 'companyinformation' );

            configCompInfo.setFieldValue( 'city', request.getParameter( 'city' ) ) ;

            nlapiSubmitConfiguration( configCompInfo );
         }

         if (assistant.getLastStep().getName() == "companypreferences" && assistant.getLastAction()
             == "next" )
         {
            // update the company preferences page
            var configCompPref = nlapiLoadConfiguration( 'companypreferences' );

            configCompPref.setFieldValue( 'CUSTOMERWELCOMEMESSAGE',
               request.getParameter( 'customerwelcomemessage' ) );

            nlapiSubmitConfiguration( configCompPref );

            // update the accounting preferences pages
            var configAcctPref = nlapiLoadConfiguration( 'accountingpreferences' );

            configAcctPref.setFieldValue( 'CREDLIMDAYS', request.getParameter( 'credlimdays' ) );

            nlapiSubmitConfiguration( configAcctPref );
         }

```

```
        if (assistant.getLastStep().getName() == "enterlocations" && assistant.getLastAction() == "next" )
        {
           // create locations

           for (var i = 1; i <= request.getLineItemCount( 'locations'); i++)
           {
              locationRec = nlapiCreateRecord( 'location' );

              locationRec.setFieldValue( 'name', request.getLineItemValue( 'locations', 'name', i ) );
              locationRec.setFieldValue( 'tranprefix', request.getLineItemValue( 'locations',
                 'tranprefix', i ) );
              locationRec.setFieldValue( 'makeinventoryavailable', request.getLineItemValue
              ('locations', 'makeinventoryavailable', i ) );
              locationRec.setFieldValue( 'makeinventoryavailablestore',
                 request.getLineItemValue('locations', 'makeinventoryavailablestore', i ) );

              try
              {
                 // add a location to the account
                 nlapiSubmitRecord( locationRec );
              }
              catch( error )
              {
                 assistant.setError( error );
              }
           }

        }
        if( !assistant.hasError() )
           assistant.setCurrentStep( assistant.getNextStep() );

        assistant.sendRedirect( response );

     }
  }
}

function getLinkoutURL( redirect, type )
{
   var url = redirect;

   if ( type == "record" )
      url = nlapiResolveURL('record', redirect);

   url += url.indexOf('?') == -1 ? '?' : '&';

   var context = nlapiGetContext();
   url += 'customwhence='+ escape(nlapiResolveURL('suitelet',context.getScriptId(), context.getDeploymentId()))

   return url;

}
```

SuiteScript 1.0 API Governance

The following table lists each API and the units each consumes.

Important: The content in this help topic pertains to SuiteScript 1.0. The governance limits for each SuiteScript 2.0 API are listed in the individual topics describing SuiteScript 2.0 methods.

Notice the APIs marked with an asterisk consume a different number of units based on the type of record they are running on. This kind of governance model takes into account the NetSuite processing requirements for three categories of records: custom records, standard transaction records, and standard non-transaction records.

Custom records, for example, require less processing than standard records. Therefore, the unit cost for custom records is lower to be commensurate with the processing required for standard records. Similarly, standard non-transaction records require less processing than standard transaction records. Therefore, the unit cost for standard non-transaction records is lower to be commensurate with the processing required for standard transaction records.

Note: Standard **transaction** records include records such as Cash Refund, Customer Deposit, and Item Fulfillment. Standard **non-transaction** records include records such as Activity, Inventory Item, and Customer. In the section on SuiteScript Supported Records in the NetSuite Help Center, see the "Record Category" column. All record types not categorized as **Transaction** are considered to be standard non-transaction records. Custom List and Custom Record are considered to be custom records.

API	Unit Usage per API	Example
nlapiDeleteFile(id) nlapiInitiateWorkflow(recordtype, id, workflowid, initialvalues) nlapiTriggerWorkflow(recordtype, id, workflowid, actionid, stateid) nlapiScheduleScript(scriptId, deployId, params) nlapiSubmitConfiguration(name) nlapiSubmitFile(file) merge()	**20**	A user event script on a standard transaction record type (such as Invoice) that includes one call to nlapiDeleteRecord and one call to nlapiSubmitRecord consumes 40 units (assuming no other nlapi calls were made). In this case, the user event script consumes 40 units out of a possible 1,000 units available to user event scripts. (See the help topic Script Type Usage Unit Limits for the total units allowed for a user event script.)
nlapiDeleteRecord(type, id, initializeValues) nlapiSubmitRecord(record, doSourcing, ignoreMandatoryFields)	When used on standard transactions: **20** When used on standard non-transactions: **10** When used on custom records: **4**	
nlapiAttachRecord(type, id, type2, id2, attributes) nlapiDetachRecord(type, id, type2, id2, attributes) nlapiExchangeRate(sourceCurrency, targetCurrency, effectiveDate)	**10**	A scheduled script on a standard non-transaction record type (such as Customer) that includes one call to nlapiLoadRecord, one call to nlapiTransformRecord, and one call to nlapiSendEmail consumes 30 units.

API	Unit Usage per API	Example
nlapiGetLogin() nlapiLoadConfiguration(type) nlapiLoadFile(id) nlapiRequestURL(url, postdata, headers, callback, httpMethod) nlapiRequestURLWithCredentials(credentials, url, postdata, headers, httpsMethod) nlapiSearchGlobal(keywords) nlapiSearchRecord(type, id, filters, columns) nlapiSendCampaignEmail (campaigneventid, recipientid) nlapiSendEmail(author, recipient, subject, body, cc, bcc, records, attachments, notifySenderOnBounce, internalOnly, replyTo) nlapiVoidTransaction(transactionType, recordId) nlapiXMLToPDF(xmlstring) getResults(start, end) forEachResult(callback)		In this case, the scheduled script consumes 30 units out of a possible 10,000 units available to scheduled scripts. (See the help topic Script Type Usage Unit Limits for the total units allowed for a scheduled script.)
nlapiCreateRecord(type, initializeValues) nlapiCopyRecord(type, id, initializeValues) nlapiLookupField(type, id, fields, text) nlapiLoadRecord(type, id, initializeValues) nlapiSubmitField(type, id, fields, values, doSourcing) nlapiTransformRecord(type, id, transformType, transformValues)	When used on standard transactions: **10** When used on standard non-transactions: **5** When used on custom records: **2**	**Note:** To conserve units, only use nlapiSubmitField on fields that are available through inline edit. For more information, see Updating a field that is available through inline edit and Updating a field that is not available through inline edit.
nlapiLoadSearch(type, id) getFuture() saveSearch(title, scriptId)	**5**	
nlapiLogExecution(type, title, details)	See the help topic Governance on Script Logging.	
nlapiSetRecoveryPoint() nlapiSubmitCSVImport(nlobjCSVImport) submit(nlobjDuplicateJobRequest)	**100**	
All other SuiteScript APIs	0	

SuiteScript 1.0 Record Initialization Defaults

You can use SuiteScript to specify record initialization parameters that will default when creating, copying, loading, and transforming records. To enable this behavior, use the **initializeValues** parameter in the following APIs:

- nlapiCreateRecord(type, initializeValues)
- nlapiCopyRecord(type, id, initializeValues)
- nlapiDeleteRecord(type, id, initializeValues)
- nlapiLoadRecord(type, id, initializeValues)

In nlapiTransformRecord(type, id, transformType, transformValues), use the **transformValues** parameter to set initialization values during the record transformation process.

The initializeValues parameter is an Object that can contain an array of name/value pairs of defaults that are passed upon record initialization. The following table lists initialization types that are available to certain SuiteScript-supported records and the values they can contain. For examples, see Record Initialization Examples.

Important: In your scripts, the property type does not need to be in quotes, but the property value does, unless it is a variable, number, or boolean.

Record	Initialization Type	Values
All SuiteScript-supported records. For a list of records, see the help topic SuiteScript Supported Records. For information about scripting a record in dynamic mode, see SuiteScript 1.0 Working with Records in Dynamic Mode.	recordmode	**dynamic**
All SuiteScript-supported records that support form customization.	customform	*<customformid>*
All transactions **Important:** Use of deletionreasoncode and deletionreasonmemo is supported only for nlapiDeleteRecord. It is not supported for nlapiCopyRecord, nlapiCreateRecord, or nlapiLoadRecord.	deletionreason	*<deletionreasonid>*
	deletionreasonmemo	*<string>*
Assembly Build	assemblyitem	*<assemblyitemid>*
Cash Refund	entity	*<entityid>*
Cash Sale	entity	*<entityid>*
Check	entity	*<entityid>*
Credit Memo	entity	*<entityid>*
Customer Payment	entity	*<entityid>* | *<inv>* | *<invoices>*

Record	Initialization Type	Values
		Note: If you use the <inv> parameter, the invoice with the specified invoice ID is selected in the Apply sublist. You can use the <invoices> parameter to filter invoices from the Apply sublist. If the <invoices> parameter is used, then only the specified invoices are loaded, and all other invoices are filtered out.
Customer Refund	entity	*<entityid>*
Deposit	disablepaymentfilters	*<disablepaymentfilters>*
Estimate	entity	*<entityid>*
Expense Report	entity	*<entityid>*
Invoice	entity	*<entityid>*
Item Receipt	entity	*<entityid>*
Non-Inventory Part	subtype	sale | resale | purchase
Opportunity	entity	*<entityid>*
Other Charge Item	subtype	sale | resale | purchase
Purchase Order	entity	*<entityid>*
Return Authorization	entity	*<entityid>*
Sales Order	entity	*<entityid>*
Script Deployment	script	<scriptid>
Service	subtype	sale | resale | purchase
Tax Group	nexuscountry	*<countrycode>* See Country Codes Used for Initialization Parameters.
Tax Type	country	*<countrycode>* See Country Codes Used for Initialization Parameters.
Topic	parenttopic	*<parenttopicid>*
Vendor Bill	entity	*<entityid>*
Vendor Payment	entity	*<entityid>*
Work Order	assemblyitem	*<assemblyitemid>*

Record Initialization Examples

The following samples show multiple ways to specify record initialization values. You can specify one name/value pair at a time or an array of name/value pairs.

Example 1

```
//load a sales order in dynamic mode
var rec = nlapiLoadRecord('salesorder', {recordmode: 'dynamic'});
```

Example 2

```
//load a sales order in dynamic mode and source in all values from customer (entity) 87
var rec = nlapiLoadRecord('salesorder', 111, {recordmode: 'dynamic', entity: 87});
```

Example 2

```
//create a sales order that uses values from custom form 17
var rec = nlapiCreateRecord('salesorder', {customform: 17});
```

Example 3

```
//copy a sales order - the record object returned from the copy will be in dynamic
//mode and contain values from custom form 17
var rec = nlapiCopyRecord('salesorder', 55, {recordmode: 'dynamic', customform: 17});
```

Example 4

```
//create an Array to set multiple initialization values
var initvalues = new Array();
initvalues.customform= 17;
initvalues.recordmode = 'dynamic';
initvalues.entity = 355;

// create sales order and pass values stored in the initvalues array
var rec = nlapiCreateRecord('salesorder', initvalues);
```

Example 5

```
//create a script deployment record in dynamic mode and attach a script record to it
var rec = nlapiCreateRecord('scriptdeployment', {recordmode: 'dynamic', script: 68});
rec.setFieldValue('status', 'NOTSCHEDULED');
```

Country Codes Used for Initialization Parameters

If you are scripting the Tax Group or Tax Type records, you can initialize the record to source all values related to a specific country. In your script, use the country code for the < *countrycodeid* > value, for example:

nlapiCreateRecord('taxgroup', {nexuscountry: 'AR'});

Country Code	Country Name
AD	Andorra
AE	United Arab Emirates
AF	Afghanistan
AG	Antigua and Barbuda
AI	Anguilla
AL	Albania

Country Code	Country Name
AM	Armenia
AN	Netherlands Antilles
AO	Angola
AQ	Antarctica
AR	Argentina
AS	American Samoa
AT	Austria
AU	Australia
AW	Aruba
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BD	Bangladesh
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BH	Bahrain
BI	Burundi
BJ	Benin
BL	Saint Barthélemy
BM	Bermuda
BN	Brunei Darussalam
BO	Bolivia
BR	Brazil
BS	Bahamas
BT	Bhutan
BV	Bouvet Island
BW	Botswana
BY	Belarus
BZ	Belize
CA	Canada
CC	Cocos (Keeling) Islands

Country Code	Country Name
CD	Congo, Democratic People's Republic
CF	Central African Republic
CG	Congo, Republic of
CH	Switzerland
CI	Cote d'Ivoire
CK	Cook Islands
CL	Chile
CM	Cameroon
CN	China
CO	Colombia
CR	Costa Rica
CU	Cuba
CV	Cap Verde
CX	Christmas Island
CY	Cyprus
CZ	Czech Republic
DE	Germany
DJ	Djibouti
DK	Denmark
DM	Dominica
DO	Dominican Republic
DZ	Algeria
EC	Ecuador
EE	Estonia
EG	Egypt
EH	Western Sahara
ER	Eritrea
ES	Spain
ET	Ethiopia
FI	Finland
FJ	Fiji
FK	Falkland Islands (Malvina)

Country Code	Country Name
FM	Micronesia, Federal State of
FO	Faroe Islands
FR	France
GA	Gabon
GB	United Kingdom (GB)
GD	Grenada
GE	Georgia
GF	French Guiana
GG	Guernsey
GH	Ghana
GI	Gibraltar
GL	Greenland
GM	Gambia
GN	Guinea
GP	Guadeloupe
GQ	Equatorial Guinea
GR	Greece
GS	South Georgia
GT	Guatemala
GU	Guam
GW	Guinea-Bissau
GY	Guyana
HK	Hong Kong
HM	Heard and McDonald Islands
HN	Honduras
HR	Croatia/Hrvatska
HT	Haiti
HU	Hungary
ID	Indonesia
IE	Ireland
IL	Israel
IM	Isle of Man

Country Code	Country Name
IN	India
IO	British Indian Ocean Territory
IQ	Iraq
IR	Iran (Islamic Republic of)
IS	Iceland
IT	Italy
JE	Jersey
JM	Jamaica
JO	Jordan
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KH	Cambodia
KI	Kiribati
KM	Comoros
KN	Saint Kitts and Nevis
KP	Korea, Democratic People's Republic
KR	Korea, Republic of
KW	Kuwait
KY	Cayman Islands
KZ	Kazakhstan
LA	Lao People's Democratic Republic
LB	Lebanon
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
LY	Libyan Arab Jamahiriya

Country Code	Country Name
MA	Morocco
MC	Monaco
MD	Moldova, Republic of
ME	Montenegro
MF	Saint Martin
MG	Madagascar
MH	Marshall Islands
MK	Macedonia
ML	Mali
MM	Myanmar
MN	Mongolia
MO	Macau
MP	Northern Mariana Islands
MQ	Martinique
MR	Mauritania
MS	Montserrat
MT	Malta
MU	Mauritius
MV	Maldives
MW	Malawi
MX	Mexico
MY	Malaysia
MZ	Mozambique
NA	Namibia
NC	New Caledonia
NE	Niger
NF	Norfolk Island
NG	Nigeria
NI	Nicaragua
NL	Netherlands
NO	Norway
NP	Nepal

Country Code	Country Name
NR	Nauru
NU	Niue
NZ	New Zealand
OM	Oman
PA	Panama
PE	Peru
PF	French Polynesia
PG	Papua New Guinea
PH	Philippines
PK	Pakistan
PL	Poland
PM	St. Pierre and Miquelon
PN	Pitcairn Island
PR	Puerto Rico
PS	Palestinian Territories
PT	Portugal
PW	Palau
PY	Paraguay
QA	Qatar
RE	Reunion Island
RO	Romania
RS	Serbia
RU	Russian Federation
RW	Rwanda
SA	Saudi Arabia
SB	Solomon Islands
SC	Seychelles
SD	Sudan
SE	Sweden
SG	Singapore
SH	St. Helena
SI	Slovenia

Country Code	Country Name
SJ	Svalbard and Jan Mayen Islands
SK	Slovak Republic
SL	Sierra Leone
SM	San Marino
SN	Senegal
SO	Somalia
SR	Suriname
ST	Sao Tome and Principe
SV	El Salvador
SY	Syrian Arab Republic
SZ	Swaziland
TC	Turks and Caicos Islands
TD	Chad
TF	French Southern Territories
TG	Togo
TH	Thailand
TJ	Tajikistan
TK	Tokelau
TM	Turkmenistan
TN	Tunisia
TO	Tonga
TP	East Timor
TR	Türkiye
TT	Trinidad and Tobago
TV	Tuvalu
TW	Taiwan
TZ	Tanzania
UA	Ukraine
UG	Uganda
UM	US Minor Outlying Islands
US	United States
UY	Uruguay

Country Code	Country Name
UZ	Uzbekistan
VA	Holy See (City Vatican State)
VC	Saint Vincent and the Grenadines
VE	Venezuela
VG	Virgin Islands (British)
VI	Virgin Islands (USA)
VN	Vietnam
VU	Vanuatu
WF	Wallis and Futuna Islands
WS	Western Samoa
YE	Yemen
YT	Mayotte
ZA	South Africa
ZM	Zambia
ZW	Zimbabwe

SuiteScript 1.0 Supported File Types

This section provides a list of all files types that can be defined in the *type* argument in the following APIs:

- nlapiCreateFile(name, type, contents) in SuiteScript Functions .
- nlobjResponse.setContentType(type, name, disposition) in SuiteScript Objects

  As a best practice, you should always use this method to set the content type for response objects in Suitelets. Setting the content type ensures that the content of the response is handled correctly. For example, if your Suitelet sends a response in JSON format, set the content type to application/json.

When referencing a file type in the *type* argument, use the **file type ID**. See the following example:

```
nlapiCreateFile('helloworld.txt', ' PLAINTEXT ', 'Hello World\nHello World');
```

Important: Be aware that the nlapiCreateFile function does not support the creation of non-text file types such as PDFs, unless the contents argument is base-64 encoded.

File Type ID	Name	Extension	Content Type
AUTOCAD	AutoCad	.dwg	application/x-autocad
BMPIMAGE	BMP Image	.bmp	image/x-xbitmap
CSV	CSV File	.csv	text/csv
EXCEL	Excel File	.xls	application/vnd.ms-excel
FLASH	Flash Animation	.swf	application/x-shockwave-flash
GIFIMAGE	GIF Image	.gif	image/gif
GZIP	GNU Zip File	.gz	application/x-gzip-compressed
HTMLDOC	HTML File	.htm	text/html
ICON	Icon Image	.ico	image/ico
JAVASCRIPT	JavaScript File	.js	text/javascript
JPGIMAGE	JPEG Image	.jpg	image/jpeg
JSON	JSON File	.json	application/json
MESSAGERFC	Message RFC	.eml	message/rfc822
MP3	MP3 Audio	.mp3	audio/mpeg
MPEGMOVIE	MPEG Video	.mpg	video/mpeg
MSPROJECT	Project File	.mpp	application/vnd.ms-project
PDF	PDF File	.pdf	application/pdf
PJPGIMAGE	PJPEG Image	.pjpeg	image/pjpeg
PLAINTEXT	Plain Text File	.txt	text/plain
PNGIMAGE	PNG Image	.png	image/x-png
POSTSCRIPT	PostScript File	.ps	application/postscript

File Type ID	Name	Extension	Content Type
POWERPOINT	PowerPoint File	.ppt	application/vnd.ms-powerpoint
QUICKTIME	QuickTime Video	.mov	video/quicktime
RTF	RTF File	.rtf	application/rtf
SMS	SMS File	.sms	application/sms
STYLESHEET	CSS File	.css	text/css
TIFFIMAGE	TIFF Image	.tiff	image/tiff
VISIO	Visio File	.vsd	application/vnd.visio
WORD	Word File	.doc	application/msword
XMLDOC	XML File	.xml	text/xml
ZIP	Zip File	.zip	application/zip

SuiteScript 1.0 Olson Values

The values below are used in SuiteScript when working with alternate time zones (see the DateTime Time Zone APIs and the DateTime Time Zone methods for nlobjRecord). The time zones used with these APIs are known as Olson Values. These values are maintained by the International Assigned Numbers Authority (IANA) in an international standard time zone database. This database defines all valid time zone names in a format similar to "America/Denver" and includes daylight savings time rules for each time zone. The values used to populated the Time Zone dropdown list found at Home > Set Preferences are also based on these values.

Key	Olson Value	Description
1	Etc/GMT+12	(GMT-12:00) International Date Line West
2	Pacific/Samoa	(GMT-11:00) Midway Island, Samoa
3	Pacific/Honolulu	(GMT-10:00) Hawaii
4	America/Anchorage	(GMT-09:00) Alaska
5	America/Los_Angeles	(GMT-08:00) Pacific Time (US & Canada)
6	America/Tijuana	(GMT-08:00) Tijuana, Baja California
7	America/Denver	(GMT-07:00) Mountain Time (US & Canada)
8	America/Phoenix	(GMT-07:00) Arizona
9	America/Chihuahua	(GMT-07:00) Chihuahua, La Paz, Mazatlan - New
10	America/Chicago	(GMT-06:00) Central Time (US & Canada)
11	America/Regina	(GMT-06:00) Saskatchewan
12	America/Guatemala	(GMT-06:00) Central America
13	America/Mexico_City	(GMT-06:00) Guadalajara, Mexico City, Monterrey - Old
14	America/New_York	(GMT-05:00) Eastern Time (US & Canada)
15	US/East-Indiana	(GMT-05:00) Indiana (East)
16	America/Bogota	(GMT-05:00) Bogota, Lima, Quito
17	America/Caracas	(GMT-04:30) Caracas
18	America/Halifax	(GMT-04:00) Atlantic Time (Canada)
19	America/La_Paz	(GMT-04:00) Georgetown, La Paz, San Juan
20	America/Manaus	(GMT-04:00) Manaus
21	America/Santiago	(GMT-04:00) Santiago
22	America/St_Johns	(GMT-03:30) Newfoundland
23	America/Sao_Paulo	(GMT-03:00) Brasilia
24	America/Buenos_Aires	(GMT-03:00) Buenos Aires
25	Etc/GMT+3	(GMT-03:00) Cayenne

Key	Olson Value	Description
26	America/Godthab	(GMT-03:00) Greenland
27	America/Montevideo	(GMT-03:00) Montevideo
28	America/Noronha	(GMT-02:00) Mid-Atlantic
29	Etc/GMT+1	(GMT-01:00) Cape Verde Is.
30	Atlantic/Azores	(GMT-01:00) Azores
31	Europe/London	(GMT) Greenwich Mean Time : Dublin, Edinburgh, Lisbon, London
32	GMT	(GMT) Casablanca
33	Atlantic/Reykjavik	(GMT) Monrovia, Reykjavik
34	Europe/Warsaw	(GMT+01:00) Sarajevo, Skopje, Warsaw, Zagreb
35	Europe/Paris	(GMT+01:00) Brussels, Copenhagen, Madrid, Paris
36	Etc/GMT-1	(GMT+01:00) West Central Africa
37	Europe/Amsterdam	(GMT+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna
38	Europe/Budapest	(GMT+01:00) Belgrade, Bratislava, Budapest, Ljubljana, Prague
39	Africa/Cairo	(GMT+02:00) Cairo
40	Europe/Istanbul	(GMT+02:00) Athens, Bucharest, Istanbul
41	Asia/Jerusalem	(GMT+02:00) Jerusalem
42	Asia/Amman	(GMT+02:00) Amman
43	Asia/Beirut	(GMT+02:00) Beirut
44	Africa/Johannesburg	(GMT+02:00) Harare, Pretoria
45	Europe/Kiev	(GMT+02:00) Helsinki, Kyiv, Riga, Sofia, Tallinn, Vilnius
46	Europe/Minsk	(GMT+02:00) Minsk
47	Africa/Windhoek	(GMT+02:00) Windhoek
48	Asia/Riyadh	(GMT+03:00) Kuwait, Riyadh
49	Europe/Moscow	(GMT+03:00) Moscow, St. Petersburg, Volgograd
50	Asia/Baghdad	(GMT+03:00) Baghdad
51	Africa/Nairobi	(GMT+03:00) Nairobi
52	Asia/Tehran	(GMT+03:30) Tehran
53	Asia/Muscat	(GMT+04:00) Abu Dhabi, Muscat
54	Asia/Baku	(GMT+04:00) Baku
55	Asia/Yerevan	(GMT+04:00) Caucasus Standard Time
56	Etc/GMT-3	(GMT+04:00) Tbilisi
57	Asia/Kabul	(GMT+04:30) Kabul

Key	Olson Value	Description
58	Asia/Karachi	(GMT+05:00) Islamabad, Karachi
59	Asia/Yekaterinburg	(GMT+05:00) Ekaterinburg
60	Asia/Tashkent	(GMT+05:00) Tashkent
61	Asia/Calcutta	(GMT+05:30) Chennai, Kolkata, Mumbai, New Delhi
62	Asia/Katmandu	(GMT+05:45) Kathmandu
63	Asia/Almaty	(GMT+06:00) Novosibirsk
64	Asia/Dhaka	(GMT+06:00) Astana, Dhaka
65	Asia/Rangoon	(GMT+06:30) Yangon (Rangoon)
66	Asia/Bangkok	(GMT+07:00) Bangkok, Hanoi, Jakarta
67	Asia/Krasnoyarsk	(GMT+07:00) Krasnoyarsk
68	Asia/Hong_Kong	(GMT+08:00) Beijing, Chongqing, Hong Kong, Urumqi
69	Asia/Kuala_Lumpur	(GMT+08:00) Kuala Lumpur, Singapore
70	Asia/Taipei	(GMT+08:00) Taipei
71	Australia/Perth	(GMT+08:00) Perth
72	Asia/Irkutsk	(GMT+08:00) Irkutsk
73	Asia/Manila	(GMT+08:00) Manila
74	Asia/Seoul	(GMT+09:00) Seoul
75	Asia/Tokyo	(GMT+09:00) Osaka, Sapporo, Tokyo
76	Asia/Yakutsk	(GMT+09:00) Yakutsk
77	Australia/Darwin	(GMT+09:30) Darwin
78	Australia/Adelaide	(GMT+09:30) Adelaide
79	Australia/Sydney	(GMT+10:00) Canberra, Melbourne, Sydney
80	Australia/Brisbane	(GMT+10:00) Brisbane
81	Australia/Hobart	(GMT+10:00) Hobart
82	Pacific/Guam	(GMT+10:00) Guam, Port Moresby
83	Asia/Vladivostok	(GMT+10:00) Vladivostok
84	Asia/Magadan	(GMT+11:00) Magadan, Solomon Is., New Caledonia
85	Pacific/Kwajalein	(GMT+12:00) Fiji, Marshall Is.
86	Pacific/Auckland	(GMT+12:00) Auckland, Wellington
87	Pacific/Tongatapu	(GMT+13:00) Nuku'alofa

Multiple Shipping Routes and SuiteScript

The following topics are covered in this section. If you are unfamiliar with the Multiple Shipping Routes feature, you should read each topic sequentially.

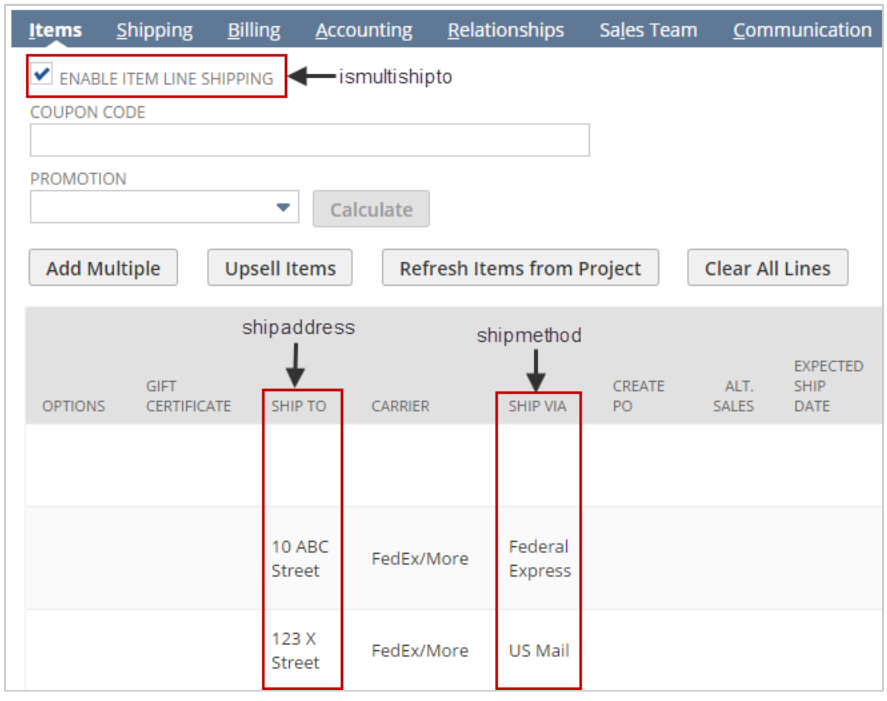
- What is the Multiple Shipping Routes feature?
- How does MSR work in SuiteScript?
- Which fields are associated with MSR?
- Does MSR work on existing custom forms?
- Multiple Shipping Routes Sample Code for SuiteScript
- Do I need to make code changes to existing SuiteScript code?

What is the Multiple Shipping Routes feature?

With the Multiple Shipping Routes (MSR) feature, you can associate several items with one transaction, and then set different shipping addresses and shipping methods for each item. Transaction types such as sales order, cash sale, invoice, estimate, and item fulfillment all support MSR.

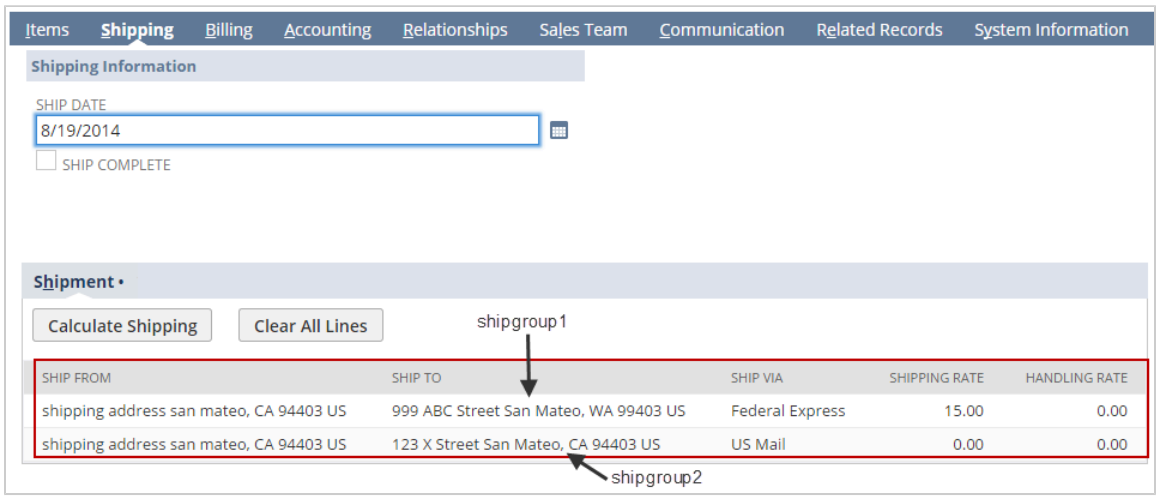
Note: For additional information about this feature, as well as steps for enabling MSR in your account, see the help topic Multiple Shipping Routes in the NetSuite Help Center.

The following figure shows a sales order with three items. When MSR is enabled in your account, **and** the *Enable Item Line Shipping* box is selected on the transaction, each item can have its own ship-to address and shipping method. The ship-to address is specified in the *Ship To* column; the shipping method is specified in the *Ship Via* column. The SuiteScript internal IDs for each field are shown the figure below.



In the UI, after all items have been added to the transaction (a sales order in this example), you must then create individual shipping groups by clicking the **Calculate Shipping** button on the Shipping tab.

A shipping group includes all details related to where the item is being shipped from, where it is being shipped to, the item's shipping method, and its shipping rate (see figure).



Although there is no UI label called "Shipping Group," SuiteScript developers can verify that shipping groups have been created by either looking in the UI after the record has first been submitted or by searching on the record and specifying *shipgroup* as one of the search return columns. See the code sample called Get the shipping groups created for the sales order for details.

The previous figure shows the UI equivalent of two separate shipping groups on a sales order. These groups are *ship group* 1 and *ship group* 2.

Note that although the sales order included three items, only two shipping groups were generated. This is because the shipping information for two of the items is the same (123 Main Street for ship-to, and shipping method). The third item contains shipping details that are not like the previous two items. Therefore, this order contains three items, but only two different shipping groups.

How does MSR work in SuiteScript?

When working with MSR-enabled transactions in SuiteScript, developers should be aware of the following:

- In SuiteScript, at the time of creating a sales order, you cannot override the default shipping rate that has been set for an item. SuiteScript developers should be aware of this when creating **user event** and **scheduled** scripts.
- There is no SuiteScript equivalent of the Calculate Shipping button that appears on the Shipping tab. In SuiteScript, shipping calculations are handled by the NetSuite backend when the transaction is submitted.
- The nlapiTransformRecord(...) API includes an optional *shipgroup* setting. For example:

```
var itemFulfillment = nlapiTransformRecord('salesorder', id, 'itemfulfillment', { ' shipgroup ' : 50 });
var fulfillmentId = nlapiSubmitRecord(itemFulfillment, true);
```

When working with MSR-enabled transactions, you must specify a value for *shipgroup* during your transforms. If you do not specify a value, the value **1** (for the first shipping group) is defaulted. This means that only the items belonging to the first shipping group will be fulfilled when the sales order is transformed.

For a code sample that shows how to transform a sales order to an item fulfillment, see Transform the sales order to create an item fulfillment.

- In both the UI and in SuiteScript, if you make any update to any item on MSR-enabled transactions, this action may result in changes to the shipping cost. Every time an item is updated and the record is submitted, NetSuite re-calculates the shipping rate. NetSuite calculates all orders based on "real-time" shipping rates.
- In both the UI and in SuiteScript, the only *transformation* workflow that is impacted by MSR is the sales order to fulfillment workflow. Invoicing and other transaction workflows are not impacted.
- After MSR is enabled, and the Enable Item Line Shipping box is checked, the Sales Order displays a shipaddress, shipcarrier, and shipmethod field for each item line.
- Selecting an item line sends a system request that reloads the available Carrier (shipcarrier) and Ship Via (shipmethod) lists. Reloading these fields prompts post-sourcing to trigger twice.

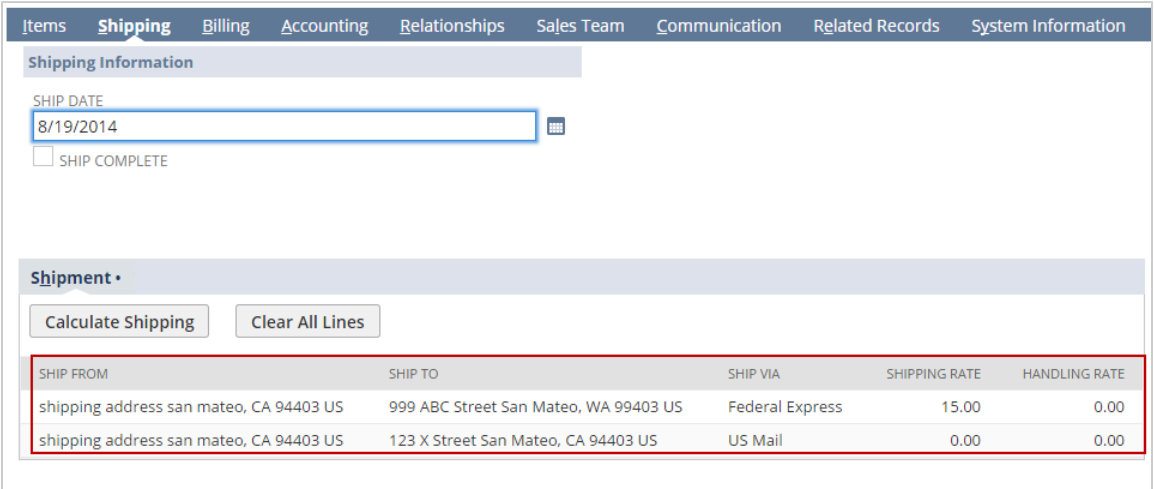
  To learn more, see the help topic Post Sourcing Sample.

Which fields are associated with MSR?

The following table lists UI field labels and their corresponding SuiteScript internal IDs for all MSR-related fields.

UI Label	Element Name	Note
Enable Item Line Shipping	ismultishipto	Set to ' **T** ' to allow for multiple items with separate shipping address/methods
Ship To	shipaddress	References the internal ID of the customer's shipping address. You can get the internal ID by clicking the Address tab on the customer's record. The address ID appears in the ID column. **Note:** The Show Internal ID preference must be enabled for address IDs to show.
Ship Via	shipmethod	References the internal ID of the item. Go to the Items list to see all item IDs. **Note:** The Show Internal ID preference must be enabled for address IDs to show.
Shipping sublist	shipgroup	Each item that has a separate shipping address/shipping method will have a unique shipgroup number. When you transform a sales order to create a fulfullment, and the sales order has MSR enabled, you will need to specify a shipgroup value (1, 2, 3, etc) for each shipgroup you want fulfilled. See figure below. **Important:** If no shipgroup value is specified, only the items associated with the first shipgroup (1) will be fulfilled.

This figure shows two different shipping groups: ship group 1 and ship group 2. When transforming the transaction using nlapiTransformRecord(...), you must specify each item you want fulfilled based on its shipgroup value.



Does MSR work on existing custom forms?

Yes. However, after you enable Multiple Shipping Routes in your account, you must also enable MSR on your custom form by adding the Enable Line Item Shipping box to the Items sublist. For steps on adding this box to a custom form, see *Multiple Shipping Routes* in the NetSuite Help Center.

Note: After MSR is enabled in your account, the Enable Line Item Shipping box is automatically added to the Items sublist on standard forms.

Multiple Shipping Routes Sample Code for SuiteScript

The following samples show a typical workflow for MSR-enabled transactions. Note that these samples reference the sales order as the transaction type.

1. Create a sales order and set your items
2. Get the shipping groups created for the sales order
3. Transform the sales order to create an item fulfillment
4. Search for MSR-enabled sales orders that have not been fulfilled

Create a sales order and set your items

```
// Create Sales order with two items and two different shipping addresses
var record = nlapiCreateRecord('salesorder');
record.setFieldValue('entity', 87); //set customer ID

// Set values for the first item
record.setLineItemValue('item', 'item', 1, 380);
record.setLineItemValue('item', 'quantity', 1, 1);
record.setLineItemValue('item',  'shipaddress', 1, 84);
record.setLineItemValue('item', 'shipmethod', 1, 37);

// Set values for the second item
record.setLineItemValue('item', 'item', 2, 440);
```

```
record.setLineItemValue('item', 'quantity', 2, 1);
record.setLineItemValue('item', 'shipaddress', 2, 275);
record.setLineItemValue('item','shipmethod', 2, 37);

var id = nlapiSubmitRecord(record, true);
```

Get the shipping groups created for the sales order

```
var columns = new Array();
var filters = new Array();
filters[0] = new nlobjSearchFilter('internalid', null, 'is', id);
filters[1] = new nlobjSearchFilter('shipgroup', null, 'greaterthan', 0);
columns[0] = new nlobjSearchColumn('shipgroup');
var searchresults = nlapiSearchRecord('salesorder', null, filters, columns);
var shipgroups = new Array();
for( i=0; i< searchresults.length ; i++ )
{
    shipgroups[i] = searchresults[i].getValue('shipgroup');
}
```

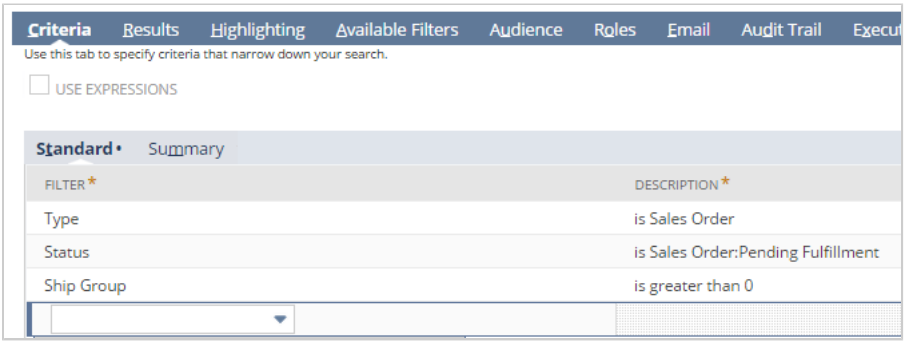
Transform the sales order to create an item fulfillment

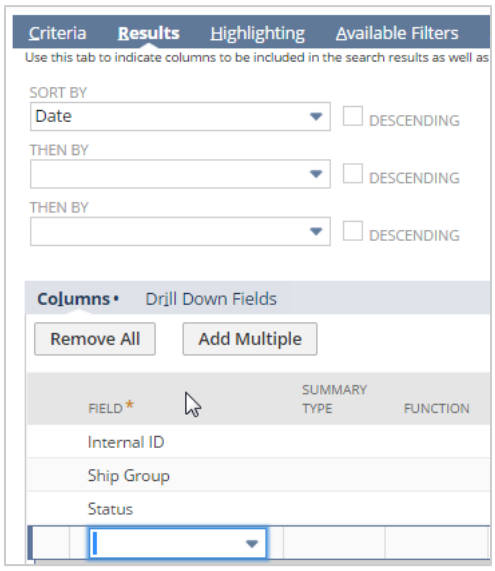
```
// Transform the sales order and pass each of the shipping groups
for(i=0; i< shipgroups.length ; i ++ )
{
var itemFulfillment = nlapiTransformRecord('salesorder', id, 'itemfulfillment', { 'shipgroup' : shipgroups[i]})
var fulfillmentId = nlapiSubmitRecord(itemFulfillment, true)
}
```

Important: If you do not pass a value for *shipgroup*, only the first item on the sales order is fulfilled.

Search for MSR-enabled sales orders that have not been fulfilled

1. Create a saved search to obtain shipping group IDs. The following figures show the criteria and results column values to set.





2. Next, run your saved search in SuiteScript to verify the same results are returned as in the saved searched performed in the UI. Then transform all unfulfilled orders based on shipgroup ID. For example,

```
var results = nlapiSearchRecord('salesorder', 17); // where 17 is the internal ID of the previous saved search
for ( var i = 0; results != null && i < results .length; i++)
{
    var id =  results[i].getValue('internalid');
    var shipgroup= results[i].getValue('shipgroup');

// Transform
   var itemFulfillment = nlapiTransformRecord('salesorder', id, 'itemfulfillment', { 'shipgroup' : shipgroup })
   var fulfillmentId = nlapiSubmitRecord(itemFulfillment)
}
```

Do I need to make code changes to existing SuiteScript code?

Simply enabling the MSR feature in your NetSuite account does not require any code changes. However, after you enable MSR on individual transactions (by selecting the *Enable Item Line Shipping* box on a transaction's Items sublist), you may need to make the following updates to your code:

1. When you first enable the MSR feature, you will be prompted to enable the *Per-line taxes* feature. Therefore, when you add items to a transaction that has MSR enabled, AND you want set a tax for your items, you will need to set a *taxcode* value for each line item. For example,

   ```
   nlapiSetLineItemValue('item', 'taxcode', 1, '230' ) // where 230 is the tax code internal ID
   ```

   Note that if you do not want to add taxes to an item, you are not required to set a value for taxcode. In other words, if you want to add taxes, you must add them on a per-line basis.

   If you have never set *taxcode* values on any of your transactions, and you do not want to add *taxcode* values, no code changes are required.

2. If MSR is enabled on the transaction, you can search for the transaction, get the ID, and then fulfill the order. With MSR enabled, your existing search will now have to include a *shipgroup* column in your search.

3. Any sales order to item fulfillment transformation code will now have to include *shipgroup* as a transaction value. For example:

```
var itemFulfillment = nlapiTransformRecord('salesorder', id, 'itemfulfillment', { 'shipgroup' : 5 })/
var fulfillmentId = nlapiSubmitRecord(itemFulfillment, true);
```

A transformation default for salesorder->itemfulfillment was added to the SuiteScript API so that the shipping route (shipgroup) can be defaulted in during transformations.

SuiteScript 1.0 Referencing the currencyname Field in SuiteScript

Note: The content in this help topic pertains to all versions of SuiteScript. Be aware that currently it may only include links or examples for SuiteScript 1.0.

The **currencyname** field in SuiteScript is intended to be read-only and not a submittable field (in other words, this field should not be accessible from user event scripts). For example, using the nlapiGetFieldValue(...) or nlapiLookupField(...) functions on **currencyname** will either return no value or will return an error.

To return currency information, you should instead reference the **currency** field.

If you must return currency name information, you will need to load the transaction and call getFieldValue(). For example,

```
nlapiLoadRecord().getFieldValue('currencyname');
```