



**FREE eBook**

# LEARNING netsuite

Free unaffiliated eBook created from  
**Stack Overflow contributors.**

**#netsuite**

# Table of Contents

About.....	1
<b>Chapter 1: Getting started with netsuite.....</b>	<b>2</b>
Remarks.....	2
Where to get Help.....	2
Versions.....	2
Examples.....	2
Eclipse SuiteCloud IDE Setup.....	2
Hello, World 1.0 Client Script.....	3
Hello, World 2.0 Client Script.....	4
<b>Chapter 2: Create a record.....</b>	<b>6</b>
Examples.....	6
Create new Task.....	6
Creating record in dynamic mode.....	6
<b>Chapter 3: Executing a Search.....</b>	<b>7</b>
Examples.....	7
SS 2.0 Ad Hoc Search.....	7
SS 2.0 From Saved Search.....	7
<b>Chapter 4: Executing a Search.....</b>	<b>9</b>
Examples.....	9
SS 2.0 From Saved Search.....	9
SS 2.0 Ad Hoc Search.....	9
Performing a summarized search.....	10
<b>Chapter 5: Exploiting formula columns in saved searches.....</b>	<b>11</b>
Introduction.....	11
Examples.....	11
Oracle SQL CASE statement in a Netsuite formula.....	11
Parsing a hierarchical record name using a regular expression.....	11
Build a complex string by concatenating multiple fields.....	11
Customize the CSS (stylesheet) for a column by inserting a DIV element.....	11
Protect string formulas from corruption and injection attacks.....	11

Protect field values from corruption when passing through a URL.....	12
Test the value of `mainline` in an SQL CASE statement.....	12
Complex, real-world-like example.....	12
Count records with with and without a value provided in a field (count missing and non-mis.....	12
<b>Chapter 6: Governance.....</b>	<b>14</b>
Remarks.....	14
<b>Governance.....</b>	<b>14</b>
API Usage Limit.....	14
Timeout and Instruction Count Limits.....	15
Memory Usage Limit.....	16
Examples.....	16
How many units do I have remaining?.....	16
<b>Chapter 7: Inline Editing with SuiteScript.....</b>	<b>18</b>
Introduction.....	18
Syntax.....	18
Parameters.....	18
Remarks.....	18
<b>Performance and Limitations.....</b>	<b>18</b>
<b>References:.....</b>	<b>19</b>
Examples.....	19
[1.0] Submit a Single Field.....	19
[1.0] Submit Multiple Fields.....	19
[2.0] Submit a Single Field.....	20
[2.0] Submit Multiple Fields.....	20
<b>Chapter 8: Loading a record.....</b>	<b>21</b>
Examples.....	21
SS 1.0.....	21
SS 2.0.....	21
<b>Chapter 9: Lookup Data from Related Records.....</b>	<b>22</b>
Introduction.....	22
Syntax.....	22

Parameters.....	22
Remarks.....	22
<b>Performance.....</b>	<b>22</b>
<b>Limitations.....</b>	<b>22</b>
Examples.....	23
[1.0] Lookup Single Field.....	23
[1.0] Lookup Multiple Fields.....	23
[1.0] Lookup Joined Fields.....	23
[2.0] Lookup Single Field.....	24
[2.0] Lookup Multiple Fields.....	24
[2.0] Lookup Joined Fields.....	25
<b>Chapter 10: Mass Delete.....</b>	<b>26</b>
Introduction.....	26
Examples.....	26
Delete based on Search Criteria.....	26
<b>Chapter 11: Requesting customField, customFieldList &amp; customSearchJoin with PHP API Advanc.</b>	<b>27</b>
Introduction.....	27
Examples.....	27
customField & customFieldList Usage.....	27
customSearchJoin Usage.....	27
<b>Chapter 12: RESTlet - Process external documents.....</b>	<b>29</b>
Introduction.....	29
Examples.....	29
RESTlet - store and attach file.....	29
<b>Chapter 13: RestLet - Retrieve Data (Basic).....</b>	<b>31</b>
Introduction.....	31
Examples.....	31
Retrieve Customer Name.....	31
<b>Chapter 14: Script and Script Deployment Records.....</b>	<b>32</b>
Introduction.....	32
Examples.....	32

Script Records.....	32
Script Deployment Records.....	33
<b>Chapter 15: Script Type Overview.....</b>	<b>35</b>
Introduction.....	35
Examples.....	35
The Client Script.....	35
The User Event Script.....	36
The Scheduled and Map/Reduce Scripts.....	36
The Suitelet and Portlet Scripts.....	37
The RESTlet.....	38
The Mass Update Script.....	38
The Workflow Action Script.....	38
The Bundle Installation Script.....	38
<b>Chapter 16: Scripting searches with Filter Expressions.....</b>	<b>40</b>
Introduction.....	40
Examples.....	40
Filter term.....	40
Filter expression.....	41
Filter expressions vs Filter Objects.....	42
Useful hints.....	43
<b>Chapter 17: Searches with large number of results.....</b>	<b>45</b>
Introduction.....	45
Examples.....	45
Using Search.ResultSet.each method.....	45
Using ResultSet.getRange method.....	45
Using Search.PagedData.fetch method.....	47
Using dedicated Map/Reduce script.....	47
<b>Chapter 18: Sourcing.....</b>	<b>50</b>
Parameters.....	50
Remarks.....	50
Impact of Store Value.....	50
Limitations of Sourcing.....	50

Examples.....	50
Pulling data into a custom field on Field Changed.....	50
Defining Sourcing.....	51
<b>Chapter 19: SS2.0 Suitelet Hello World.....</b>	<b>52</b>
Examples.....	52
Basic Hello World Suitelet - Plain Text Response.....	52
<b>Chapter 20: SuiteScript - Process Data from Excel.....</b>	<b>53</b>
Introduction.....	53
Examples.....	53
Update Rev Rec Dates and Rule.....	53
<b>Chapter 21: Understanding Transaction Searches.....</b>	<b>55</b>
Introduction.....	55
Remarks.....	55
Examples.....	55
Filtering only on Internal ID.....	55
Filtering with Main Line.....	58
Filtering Specific Sublists.....	60
<b>Chapter 22: User Event: Before and After Submit events.....</b>	<b>63</b>
Syntax.....	63
Parameters.....	63
Remarks.....	63
<b>beforeSubmit and afterSubmit.....</b>	<b>63</b>
<b>Typical Use Cases for beforeSubmit.....</b>	<b>64</b>
<b>Typical Use Cases for afterSubmit.....</b>	<b>64</b>
<b>User Events do not chain.....</b>	<b>64</b>
<b>Event Handlers return void.....</b>	<b>64</b>
<b>!! CAUTION !!.....</b>	<b>65</b>
Examples.....	65
Minimal: Log a message.....	65
Before Submit: Validate record before it is committed to database.....	66
After Submit: Determine whether a field was changed.....	68

<b>Chapter 23: User Event: Before Load event</b> .....	<b>70</b>
Parameters.....	70
Remarks.....	70
<b>beforeLoad</b> .....	<b>70</b>
<b>Typical Use Cases for beforeLoad</b> .....	<b>70</b>
<b>User Events do not chain</b> .....	<b>71</b>
<b>Event Handler returns void</b> .....	<b>71</b>
Examples.....	71
Minimal: Log a message on Before Load.....	71
Modifying the UI form.....	72
Restrict execution based on the action that triggered the User Event.....	73
Restrict execution based on the context that triggered the User Event.....	73
<b>Chapter 24: Using the NetSuite Records Browser</b> .....	<b>76</b>
Examples.....	76
Using the NetSuite Records Browser.....	76
Other Schema.....	76
Navigating the Records Browser.....	76
Reading the Schema.....	76
Finding a Field.....	77
Required Fields.....	77
nlapiSubmitField and Inline Editing.....	77
<b>Chapter 25: Working with Sublists</b> .....	<b>79</b>
Introduction.....	79
Remarks.....	79
Sublist Indices.....	79
Standard vs Dynamic Mode.....	79
Limitations.....	79
References:.....	80
Examples.....	80
[1.0] How many lines on a sublist?.....	80
[1.0] Sublists in Standard Mode.....	80

[1.0] Sublists in Dynamic Mode.....	81
[1.0] Find a Line Item.....	81
[2.0] How many lines on a sublist?.....	82
[2.0] Sublists in Standard Mode.....	82
[2.0] Sublists in Dynamic Mode.....	83
[2.0] Find a Line Item.....	83
<b>Credits.....</b>	<b>85</b>



---

# About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [netsuite](#)

It is an unofficial and free netsuite ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official netsuite.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to [info@zzzprojects.com](mailto:info@zzzprojects.com)

---

# Chapter 1: Getting started with netsuite

## Remarks

NetSuite is a cloud-based ERP, CRM, E-Commerce, and Professional Services management platform. It is used by over 30,000 companies to run their entire business.

NetSuite is fully customizable by administrators and developers, including via a JavaScript-based API called SuiteScript. Developers are able to write scripts that are triggered by various events throughout the NetSuite system to automate business processes.

## Where to get Help

1. Join the [NetSuite Professionals](#) Slack Community, where you have instant access to over 200 NetSuite Professionals across the globe.
2. Use the [NetSuite Records Browser](#) for the schema of all record types
3. Mozilla Developer Network's [JavaScript Reference Guide](#)

## Versions

Version	Release Date
<a href="#">2016.2</a>	2016-09-20

## Examples

### Eclipse SuiteCloud IDE Setup

1. Download and install the latest Eclipse IDE
  - Install Eclipse one of two ways:
    1. [Eclipse Installer](#)
    2. Download the [zip for your favorite package](#)
  - If you don't already have a preferred Eclipse package, *Eclipse for JavaScript Developers* is recommended
2. Install SuiteCloud IDE plugin
  1. Once installation is complete, launch Eclipse
  2. Navigate to *Help > Install New Software...*
  3. Click *Add...* to add a new Update Site
    - **Name:** SuiteCloud IDE
    - **Location:** [http://system.netsuite.com/download/ide/update\\_e4](http://system.netsuite.com/download/ide/update_e4)
      - **Note:** the location [depends on the version of NetSuite](#) you're currently on.
      - **For example:** if you're currently on Release 2017.1 then you should use this url instead: [http://system.netsuite.com/download/ide/update\\_17\\_1](http://system.netsuite.com/download/ide/update_17_1)

4. Select "SuiteCloud IDE" site in the *Work With* dropdown
5. Proceed through the install wizard
6. Restart Eclipse when prompted
3. Configure the SuiteCloud IDE plugin
  1. When Eclipse restarts, you will be prompted to set up the SuiteCloud plugin with a master password and default NetSuite account
  2. After completing this set up wizard, navigate to *Preferences > NetSuite*
    - Here you will find all of the SuiteCloud IDE preferences
  3. [Optional] If your primary use for Eclipse is NetSuite development, navigate to *Preferences > General > Perspectives* and make the "NetSuite" Perspective your default
4. Create a new NetSuite project
  1. Right-click in the *NS Explorer* window and select *New > NetSuite project*
  2. Follow the wizard for the project setup of your choosing. The project types are as follows:
    1. *Account Customization*: A project that leverages the *SuiteCloud Development Framework* for building custom objects, records, and scripts for customizing a NetSuite account.
    2. *SuiteScript*: A project used exclusively for writing scripts.
    3. *SSP Application*: A SuiteScript Server Pages application, used typically in conjunction with SiteBuilder or SuiteCommerce for NetSuite-backed E-Commerce applications.

## Hello, World 1.0 Client Script

1. Create the source file for your new Client Script
  1. Create a new JavaScript file using your favorite editor or IDE
  2. Add the following source code to your file (original source [here](#))

```

/**
 * A simple "Hello, World!" example of a Client Script. Uses the `pageInit`
 * event to write a message to the console log.
 */

function pageInit(type) {
    console.log("Hello, World from a 1.0 Client Script!");
}

```

3. Save the file as `hello-world.js` wherever you wish
2. Use the source file we just created to create a new *Script* record in NetSuite
  1. In your NetSuite account, navigate to *Customization > Scripting > Scripts > New*
  2. When prompted, select `hello-world.js` as the *Script File*
  3. Click *Create Script Record*
  4. When prompted, select *Client Script* as the Script Type
  5. Name your Script record *Hello World*
  6. Map the function named `pageInit` in our source file to the *Page Init* script event by

entering `pageInit` in the *Page Init Function* field

7. Save your new Script record
3. Deploy your new Script to the Employee record
  1. On your newly created Script record, click *Deploy Script*
  2. In the *Applies To* field, select *Employee*
  3. Make sure the *Status* field is set to *Testing*
  4. Click *Save*
4. See your script in action!
  1. Open your browser's developer/JavaScript console (typically F12 on most browsers)
  2. Create a new Employee by navigating to *Lists > Employees > Employees > New*
  3. Observe your "Hello, World" message in the browser console.

## Hello, World 2.0 Client Script

1. Create the source file for your new Client Script
  1. Create a new JavaScript file using your favorite editor or IDE
  2. Add the following source code to your file (original source [here](#))

```
define([], function () {
  /**
   * A simple "Hello, World!" example of a Client Script. Uses the `pageInit`
   * event to write a message to the console log.
   *
   * @NApiVersion 2.x
   * @NModuleScope Public
   * @NScriptType ClientScript
   */
  var exports = {};
  function pageInit(context) {
    console.log("Hello, World from a 2.0 Client Script!");
  }
  exports.pageInit = pageInit;
  return exports;
});
```

3. Save the file as `hello-world2.js` wherever you wish
2. Use the source file we just created to create a new *Script* record in NetSuite
  1. In your NetSuite account, navigate to *Customization > Scripting > Scripts > New*
  2. When prompted, select `hello-world2.js` as the *Script File*
  3. Click *Create Script Record*
  4. Name your Script record *Hello World*
  5. Save your new Script record
3. Deploy your new Script to the Employee record
  1. On your newly created Script record, click *Deploy Script*
  2. In the *Applies To* field, select *Employee*
  3. Make sure the *Status* field is set to *Testing*
  4. Click *Save*
4. See your script in action!

1. Open your browser's developer/JavaScript console (typically F12 on most browsers)
2. Create a new Employee by navigating to *Lists > Employees > Employees > New*
3. Observe your "Hello, World" message in the browser console.

Read *Getting started with netsuite online*: <https://riptutorial.com/netsuite/topic/3828/getting-started-with-netsuite>

---

# Chapter 2: Create a record

## Examples

### Create new Task

```
var record = nlapiCreateRecord('task');
record.setFieldValue('title', taskTitle);
var id = nlapiSubmitRecord(record, true);
```

### Creating record in dynamic mode

```
var record = nlapiCreateRecord('customrecord_ennveeitissuetracker', { recordmode: 'dynamic' });
nlapiLogExecution('DEBUG', 'record', record); record.setFieldValue('custrecord_name1', name);
record.setFieldValue('custrecord_empid', id); record.setFieldValue('custrecord_contactno',
contactno); record.setFieldValue('custrecord_email', email);
record.setFieldValue('custrecord_location', loc); record.setFieldValue('custrecord_incidentdate',
incidentdate); record.setFieldValue('custrecord_issuedescription', desc); //
record.setFieldValue('custrecord_reportedby', report); record.setFieldValue('custrecord_issuetype',
issuetype); record.setFieldValue('custrecord_priority', priority); //
record.setFieldValue('custrecord_replacementprovided', repl);
record.setFieldValue('custrecord_issuestatus', issuestatus); //
record.setFieldValue('custrecord_resolvedby', resolvedby);
record.setFieldValue('custrecord_remarks', remarks);
record.setFieldValue('custrecord_resolvedby', resolvedby);
record.setFieldValue('custrecord_updatedstatus', updatedstatus); var id =
nlapiSubmitRecord(record,true); var recordId = nlapiGetRecordId(); record =
nlapiLoadRecord('customrecord_ennveeitissuetracker', id);
```

Read Create a record online: <https://riptutorial.com/netsuite/topic/5127/create-a-record>

# Chapter 3: Executing a Search

## Examples

### SS 2.0 Ad Hoc Search

```
require(['N/search'], function(SEARCHMODULE){

    var type = 'transaction';
    var columns = [];
    columns.push(SEARCHMODULE.createColumn({
        name: 'internalid'
    }));
    columns.push(SEARCHMODULE.createColumn({
        name: 'formulanumeric',
        formula: '{quantity}-{quantityshiprecv}'
    }));

    var salesOrdersArray = [123,456,789];
    var filters = [];
    filters.push(['type', 'anyof', 'SalesOrd']);
    filters.push('and');
    filters.push(['mainline', 'is', 'F']);
    filters.push('and');
    filters.push(['internalid', 'anyof', salesOrdersArray]);

    var mySearchObj = {};
    mySearchObj.type = type;
    mySearchObj.columns = columns;
    mySearchObj.filters = filters;

    var mySearch = SEARCHMODULE.create(mySearchObj);
    var resultset = mySearch.run();
    var results = resultset.getRange(0, 1000);
    for(var i in results){
        var result = results[i];
        var row = {};
        for(var k in result.columns){
            log.debug('Result is ' + result.getValue(result.columns[k])); //Access result from
here
        }
    }
});
```

### SS 2.0 From Saved Search

```
require(['N/search'], function(SEARCHMODULE){
    var savedSearchId = 'customsearch_mySavedSearch';
    var mySearch = SEARCHMODULE.load(savedSearchId);
    var resultset = mySearch.run();
    var results = resultset.getRange(0, 1000);
    for(var i in results){
        var result = results[i];
        for(var k in result.columns){
            log.debug('Result is ' + result.getValue(result.columns[k])); //Access result from
```

```
here  
    }  
}  
});
```

Read Executing a Search online: <https://riptutorial.com/netsuite/topic/6081/executing-a-search>



---

# Chapter 4: Executing a Search

## Examples

### SS 2.0 From Saved Search

```
require(['N/search'], function(SEARCHMODULE){
  var savedSearchId = 'customsearch_mySavedSearch';
  var mySearch = SEARCHMODULE.load(savedSearchId);
  var resultset = mySearch.run();
  var results = resultset.getRange(0, 1000);
  for(var i in results){
    var result = results[i];
    for(var k in result.columns){
      log.debug('Result is ' + result.getValue(result.columns[k])); //Access result from
here
    }
  }
});
```

### SS 2.0 Ad Hoc Search

```
require(['N/search'], function(SEARCHMODULE){

  var type = 'transaction';
  var columns = [];
  columns.push(SEARCHMODULE.createColumn({
    name: 'internalid'
  }));
  columns.push(SEARCHMODULE.createColumn({
    name: 'formulanumeric',
    formula: '{quantity}-{quantityshiprecv}'
  }));

  var salesOrdersArray = [123,456,789];
  var filters = [];
  filters.push(['type', 'anyof', 'SalesOrd']);
  filters.push('and');
  filters.push(['mainline', 'is', 'F']);
  filters.push('and');
  filters.push(['internalid', 'anyof', salesOrdersArray]);

  var mySearchObj = {};
  mySearchObj.type = type;
  mySearchObj.columns = columns;
  mySearchObj.filters = filters;

  var mySearch = SEARCHMODULE.create(mySearchObj);
  var resultset = mySearch.run();
  var results = resultset.getRange(0, 1000);
  for(var i in results){
    var result = results[i];
    var row = {};
    for(var k in result.columns){
      log.debug('Result is ' + result.getValue(result.columns[k])); //Access result from
```

```
here
    }
}
});
```

## Performing a summarized search

```
// Assuming N/search is imported as `s`
var mySalesOrderSearch = s.create({
  type: 'salesorder'
  // Use the summary property of a Column to perform grouping/summarizing
  columns: [{
    name: 'salesrep',
    summary: s.Summary.GROUP
  }, {
    name: 'internalid',
    summary: s.Summary.COUNT
  }],
  filters: [{
    name: 'mainline',
    operator: 'is',
    values: ['T']
  }]
});

mySalesOrderSearch.run().each(function (result) {
  var repId = result.getValue({
    "name": "salesrep",
    "summary": s.Summary.GROUP
  });
  var repName = result.getText({
    "name": "salesrep",
    "summary": s.Summary.GROUP
  });
  var orderCount = parseInt(result.getValue({
    "name": "internalid",
    "summary": s.Summary.COUNT
  }), 10);

  log.debug({
    "title": "Order Count by Sales Rep",
    "details": repName + " has sold " + orderCount + " orders."
  });
});
```

Read Executing a Search online: <https://riptutorial.com/netsuite/topic/6359/executing-a-search>

---

# Chapter 5: Exploiting formula columns in saved searches

## Introduction

Formula columns in saved searches can exploit many features of Oracle SQL and HTML. The examples show how these features can be used, as well as pitfalls to avoid.

## Examples

### Oracle SQL CASE statement in a Netsuite formula

Using a CASE statement, conditionally display an expression in the column based on values found in another column, a.k.a. “my kingdom for an OR”. In the example, the result is obtained when the status of the transaction is `Pending Fulfillment` **OR** `Partially Fulfilled`:

```
CASE DECODE( {status}, 'Pending Fulfillment', 1, 'Partially Fulfilled', 1, 0 )
WHEN 1 THEN expression-1
END
```

### Parsing a hierarchical record name using a regular expression

Using a regular expression, parse a record name that might be hierarchical. The expression looks for the final colon in the name. It returns what follows the colon, or the entire name if none:

```
regexp_substr( {name} , '[^:]*$' )
```

### Build a complex string by concatenating multiple fields

The example builds a string from the name of the parent record, the name of this record, and the memo of this record.

```
{createdfrom} || ' ' || {name} || ' ' || {memo}
```

### Customize the CSS (stylesheet) for a column by inserting a DIV element

```
'<div style="font-size:11pt">' || expression || '</div>'
```

### Protect string formulas from corruption and injection attacks

In a string formula field, consider that some values might contain substrings which look to the browser like HTML. Unless this is intentional, it is important to protect the values from corruption. This is useful to avoid injection attacks: it prevents someone from entering HTML into a comment field in a web order that later gets interpreted on the desk of the customer service rep.

```
htf.escape_sc(
```

*expression* )

## Protect field values from corruption when passing through a URL

utl\_url.escape( *expression* )

## Test the value of `mainline` in an SQL CASE statement

In a saved search formula, the possible values of `mainline` are designed to be useful in an HTML context. When `mainline` is true, the value of `{mainline}` is the 1-character string \* (asterisk). When `mainline` is false, the value of `{mainline}` is the 6-character string `&nbsp;`; (non-breaking space, HTML encoded as a character entity reference). These string values can be compared with string literals in an SQL context.

```
CASE
WHEN {mainline} = '*' THEN expression-when-true
WHEN {mainline} = '&nbsp;' THEN expression-when-false
END
```

## Complex, real-world-like example

The following example combines several of the techniques covered here. It puts a hyperlink in a custom formatted column which, when clicked, opens the sales order record associated with a row. The hyperlink is designed to open the record in a new window or tab when clicked, and to display a tooltip when hovered. The `internalid` field used in the URL is protected from URL encoding. The customer name, when available, is displayed in the same column, protected from HTML encoding.

```
'<div style="font-size:11pt">'
  ||
CASE {mainline}
WHEN '*' THEN '<br>' || htf.escape_sc( regexp_substr( {name} , '[^:]*$' ) ) || '<br>'
END
  ||
  '<a alt="" title="Open the order associated with this line." '
  ||
  'href="javascript:void(0);" onClick="window.open('' '
  ||
  'https://system.na1.netsuite.com/app/accounting/transactions/transaction.nl?id='
  ||
utl_url.escape( {internalid} )
  ||
  '' , ''_blank'' )">'
  ||
{number}
  ||
'</a>'
  ||
'</div>'
```

## Count records with and without a value provided in a field (count missing

## and non-missing values)

Using Oracle SQL's `NVL2()` function, you can create a display column which contains one value if a field contains data and another value if a field does not contain data. For example, in an Entity search, turn the presence of a primary e-mail address into a text display column:

```
NVL2( {email} , 'YES' , 'NO' )
```

This lets you count records subtotaled by the presence or absence of an email address:

```
Field: Internal ID
Summary Type: Count

Field: Formula (Text)
Summary Type: Group
Formula: NVL2( {email} , 'YES' , 'NO' )
```

Read [Exploiting formula columns in saved searches](https://riptutorial.com/netsuite/topic/8298/exploiting-formula-columns-in-saved-searches) online:

<https://riptutorial.com/netsuite/topic/8298/exploiting-formula-columns-in-saved-searches>

---

# Chapter 6: Governance

## Remarks

---

### Governance

"Governance" is the name given to NetSuite's system for detecting and halting long-running, runaway, or resource-intensive scripts.

Each script type has governance limits that it cannot exceed, and there are four types of governance limits in place for each script type.

- API usage limit
- Instruction Count limit
- Timeout limit
- Memory usage limit

If a script exceeds its governance limit in **any one** of these four areas, NetSuite will **throw an *uncaughtable exception*** and terminate the script immediately.

### API Usage Limit

NetSuite limits the API usage of your scripts with a system based on "usage units". Some NetSuite API calls, particularly the ones that perform a read or write action on the database, cost a specific number of units each time they are invoked. Each script type then has a maximum number of units that can be used during each execution of the script.

*If a script exceeds its API usage limit, NetSuite terminates the script by throwing an `SSS_USAGE_LIMIT_EXCEEDED` error.*

Below are a few examples of unit costs for common operations. For an exhaustive list of Governance costs, see the article titled "API Governance" in NetSuite Help.

Operation	Unit Cost
Loading a Saved Search	5
Retrieving Search Results	10
Scheduling a task	10
Requesting a URL	10
Sending an email	10
Creating a custom record	2

Operation	Unit Cost
Creating an Employee record	5
Creating a Sales Order record	10
Saving a custom record	4
Saving a Contact record	10
Saving a Purchase Order record	20

Different operations use different amounts of units, and certain operations cost a different amount based on the record type being used. The larger the number of units a function costs, typically the longer it will take to execute.

Transactions are the largest of the record types, so working with them costs the largest amount of units. Conversely, custom records are very lightweight, and so do not cost many units. Standard NetSuite records that are *not* Transactions, like Customers, Employees, or Contacts, sit in between the two in terms of cost.

These are the usage limits by script type:

Script Type	Usage Limit
Client	1,000
User Event	1,000
Suitelet	1,000
Portlet	1,000
Workflow Action	1,000
RESTlet	5,000
Scheduled	10,000
Map/Reduce	10,000
Bundle Installation	10,000
Mass Update	10,000 per record

## Timeout and Instruction Count Limits

NetSuite also uses the governance system to detect and halt runaway scripts by using a timeout mechanism and an instruction counter.

If a script takes too much time to run, NetSuite will stop it by throwing an `SSS_TIME_LIMIT_EXCEEDED` error.

In addition, runaway scripts can be detected and halted based on their "Instruction Count". If the defined instruction count limits are exceeded, NetSuite will stop the script by throwing an `SSS_INSTRUCTION_COUNT_EXCEEDED` error.

There is, unfortunately, **no** Help documentation that defines:

- the timeout for each script type
- the instruction count limits for each script type
- what constitutes a single "instruction"

It is simply important to know that if you encounter either the `SSS_TIME_LIMIT_EXCEEDED` error or the `SSS_INSTRUCTION_COUNT_EXCEEDED` error in one of your scripts, you have processing that is taking too long. Focus your investigation on your loop structures to determine where optimizations may be made.

## Memory Usage Limit

If your script exceeds the memory usage limit, NetSuite will terminate your script by throwing a `SSS_MEMORY_USAGE_EXCEEDED` error.

Every variable declared, every function defined, every Object stored contributes to the memory usage of your script.

Both the **Scheduled Script** and the **Map/Reduce Script** have documented `50MB` memory limits. There is also a documented limit of `10MB` for the size of any String passed in to or returned from a RESTlet. There is no other documentation on the specific limits for a given script.

## Examples

### How many units do I have remaining?

In SuiteScript 1.0, use `nlobjContext.getRemainingUsage()` to retrieve the remaining units. An `nlobjContext` reference is retrieved using the global `nlapiGetContext` function.

```
// 1.0
var context = nlapiGetContext();
nlapiLogExecution("DEBUG", "Governance Monitoring", "Remaining Usage = " +
context.getRemainingUsage());

nlapiSearchRecord("transaction"); // uses 10 units
nlapiLogExecution("DEBUG", "Governance Monitoring", "Remaining Usage = " +
context.getRemainingUsage());
```

In SuiteScript 2.0, use the `getRemainingUsage` method of the `N/runtime` module's `Script` object.

```
// 2.0
```



```
require(["N/log", "N/runtime", "N/search"], function (log, runtime, s) {
  var script = runtime.getCurrentScript();
  log.debug({
    "title": "Governance Monitoring",
    "details": "Remaining Usage = " + script.getRemainingUsage()
  });

  s.load({"id": "customsearch_mysearch"}); // uses 5 units
  log.debug({
    "title": "Governance Monitoring",
    "details": "Remaining Usage = " + script.getRemainingUsage()
  });
});
```

Read Governance online: <https://riptutorial.com/netsuite/topic/7227/governance>

---

# Chapter 7: Inline Editing with SuiteScript

## Introduction

Inline editing allows users to very quickly modify and update the data for a particular record without having to load the entire record on a page, edit the form, then save the record.

NetSuite developers have a corresponding functionality called `submitFields`. The `submitFields` functionality is provided by the `nlapisubmitField` global function in SuiteScript 1.0 and the `N/record#submitFields` method in SuiteScript 2.0.

## Syntax

- `nlapisubmitField(recordType, recordId, fieldId, fieldValue);`
- `nlapisubmitField(recordType, recordId, fieldIds, fieldValues);`
- `nlapisubmitField(recordType, recordId, fieldId, fieldValue, doSourcing);`

## Parameters

Parameter	Details
<code>recordType</code>	<code>String</code> - The internal ID of the type of record being updated
<code>recordId</code>	<code>String</code> or <code>Number</code> - The internal ID of the record being updated
<code>fieldIds</code>	<code>String</code> or <code>String[]</code> - The internal ID(s) of the field(s) being updated
<code>fieldValues</code>	<code>any</code> or <code>any[]</code> - The corresponding values to be set in the given fields
<code>doSourcing</code>	<code>Boolean</code> - Whether dependent values should be sourced in upon record submission. Default is <code>false</code>

## Remarks

The `submitFields` functionality is a companion feature to the `lookupFields` functionality.

---

## Performance and Limitations

`submitFields` performs significantly faster and uses less governance than making the same changes by loading and submitting the full record.

Multiple fields can be updated at once for the same cost as updating a single field. Updating more fields with `submitFields` *does not* incur a higher governance cost.

However, you must be aware that only certain fields on each record type are inline-editable, and the performance savings *only* applies to these inline-editable fields. If you use the `submitFields` function on any non-inline-editable field, the field *will* be updated correctly, but behind the scenes, NetSuite will actually load and submit the record, thus taking more time and using more governance. You can determine whether a field is inline-editable by referring to the "nlapiSubmitField" column in the [Records Browser](#).

`submitFields` functionality is also limited to the *body* fields of a record. If you need to modify sublist data, you will need to load the record to make your changes, then submit the record.

---

## References:

- NetSuite Help: "Inline Editing and SuiteScript Overview"
- NetSuite Help: "Inline Editing Using nlapiSubmitField"
- NetSuite Help: "Consequences of Using nlapiSubmitField on Non Inline Editable Fields"
- NetSuite Help: "Field APIs"
- NetSuite Help: "record.submitFields(options)"

## Examples

### [1.0] Submit a Single Field

```
/**
 * A SuiteScript 1.0 example of using nlapiSubmitField to update a single field on a related
 record
 */

// From a Sales Order, get the Customer ID
var customerId = nlapiGetFieldValue("entity");

// Set a comment on the Customer record
nlapiSubmitField("customer", customerId, "comments", "This is a comment added by inline
editing with SuiteScript.");
```

### [1.0] Submit Multiple Fields

```
/**
 * A SuiteScript 1.0 example of using nlapiSubmitField to update multiple fields on a related
 record
 */

// From a Sales Order, get the Customer ID
var customerId = nlapiGetFieldValue("entity");

// Set a Comment and update the Budget Approved field on the Customer record
nlapiSubmitField("customer", customerId,
  ["comments", "isbudgetapproved"],
  ["The budget has been approved.", "T"]);
```

## [2.0] Submit a Single Field

```
/**
 * A SuiteScript 2.0 example of using N/record#submitFields to update a single field on a
 related record
 */

require(["N/record", "N/currentRecord"], function (r, cr) {

    // From a Sales Order, get the Customer ID
    var customerId = cr.get().getValue({"fieldId": "entity"});

    // Set a Comment on the Customer record
    r.submitFields({
        "type": r.Type.CUSTOMER,
        "id": customerId,
        "values": {
            "comments": "This is a comment added by inline editing with SuiteScript."
        }
    });
});
```

## [2.0] Submit Multiple Fields

```
/**
 * A SuiteScript 2.0 example of using N/record#submitFields to update multiple fields on a
 related record
 */

require(["N/record", "N/currentRecord"], function (r, cr) {

    // From a Sales Order, get the Customer ID
    var customerId = cr.get().getValue({"fieldId": "entity"});

    // Set a Comment and check the Budget Approved box on the Customer record
    r.submitFields({
        "type": r.Type.CUSTOMER,
        "id": customerId,
        "values": {
            "comments": "The budget has been approved.",
            "isbudgetapproved": true
        }
    });
});
```

Read Inline Editing with SuiteScript online: <https://riptutorial.com/netsuite/topic/9082/inline-editing-with-suitescript>

---

# Chapter 8: Loading a record

## Examples

### SS 1.0

```
var recordType = 'customer'; // The type of record to load. The string internal id.
var recordID = 100; // The specific record instances numeric internal id.
var initializeValues = null;
/* The first two parameters are required but the third --
 * in this case the variable initializeValues -- is optional. */
var loadedRecord = nlapiLoadRecord(recordType, recordID, initializeValues);
```

### SS 2.0

This example assumes that the record module is set to the variable RECORDMODULE, as shown below.

```
require(['N/record'], function(RECORDMODULE){

    var recordType = RECORDMODULE.Type.SALES_ORDER; //The type of record to load.
    var recordID = 100; //The internal ID of the existing record instance in NetSuite.
    var isDynamic = true; //Determines whether to load the record in dynamic mode.

    var loadedRecord = RECORDMODULE.load({
        type: recordType,
        id: recordID,
        isDynamic: isDynamic,
    });
});
```

Read Loading a record online: <https://riptutorial.com/netsuite/topic/4685/loading-a-record>

---

# Chapter 9: Lookup Data from Related Records

## Introduction

When processing a given record, you will oft need to retrieve data from one of its related records. For example, when working with a given Sales Order, you may need to retrieve data from the related Sales Rep. In SuiteScript terminology, this is called a **lookup**.

Lookup functionality is provided by the `nlapiLookupField` global function in SuiteScript 1.0 and the `N/search` module's `lookupFields` method in SuiteScript 2.0

## Syntax

- `nlapiLookupField(recordType, recordId, columns);`

## Parameters

Parameter	Details
<code>recordType</code>	<code>String</code> - The internal ID of the type of record being looked up (e.g. <code>salesorder</code> , <code>employee</code> )
<code>recordId</code>	<code>String</code> or <code>Number</code> - The internal ID of the record being looked up
<code>columns</code>	<code>String</code> or <code>String[]</code> - The list of fields to retrieve from the record. Field IDs can be referenced from the "Search Columns" section of the <a href="#">Records Browser</a> . Joined fields can be retrieved using dot syntax (e.g. <code>salesrep.email</code> )

## Remarks

---

## Performance

A Lookup is just shorthand for performing a search that filters on the internal ID of a single record for the result. Under the hood, lookups are actually performing a search, so the performance will be similar to that of a search that returns a single record.

This also means that a lookup will perform faster than loading the record to retrieve the same information.

# Limitations

Lookups can only be used to retrieve body field data. You cannot retrieve data from the sublists of a related record using a lookup. If you need sublist data, you will either need to perform a search or load the related record.

## Examples

### [1.0] Lookup Single Field

```
/**
 * An example of nlapiLookupField to retrieve a single field from a related record
 */

// Get the Sales Rep record ID
var repId = nlapiGetFieldValue("salesrep");

// Get the name of the Sales Rep
var repName = nlapiGetFieldText("salesrep");

// Retrieve the email address from the associated Sales Rep
var repEmail = nlapiLookupField("employee", repId, "email");

console.log(repEmail);
console.log(repName + "'s email address is " + repEmail);
```

### [1.0] Lookup Multiple Fields

```
/**
 * An example of nlapiLookupField to retrieve multiple fields from a related record
 */

// Get the Sales Rep record ID
var repId = nlapiGetFieldValue("salesrep");

// Retrieve multiple fields from the associated Sales Rep
var repData = nlapiLookupField("employee", repId, ["email", "firstname"]);

console.log(repData);
console.log(repData.firstname + "'s email address is " + repData.email);
```

### [1.0] Lookup Joined Fields

```
/**
 * An example of nlapiLookupField to retrieve joined fields from a related record
 */

var repId = nlapiGetFieldValue("salesrep");

// Retrieve multiple fields from the associated Sales Rep
var repData = nlapiLookupField("employee", repId, ["email", "firstname", "department.name"]);
```

```
console.log(repData);
console.log(repData.firstname + "'s email address is " + repData.email);
console.log(repData.firstname + "'s department is " + repData["department.name"]);
```

## [2.0] Lookup Single Field

```
require(["N/search", "N/currentRecord"], function (s, cr) {

  /**
   * An example of N/search#lookupFields to retrieve a single field from a related record
   */
  (function () {

    var record = cr.get();

    // Get the Sales Rep record ID
    var repId = record.getValue({
      "fieldId": "salesrep"
    });

    // Get the name of the Sales Rep
    var repName = record.getText({
      "fieldId": "salesrep"
    });

    // Retrieve the email address from the associated Sales Rep
    var repData = s.lookupFields({
      "type": "employee",
      "id": repId,
      "columns": ["email"]
    });

    console.log(repData);
    console.log(repName + "'s email address is " + repData.email);
  }) ();
});
```

## [2.0] Lookup Multiple Fields

```
require(["N/search", "N/currentRecord"], function (s, cr) {

  /**
   * An example of N/search#lookupFields to retrieve multiple fields from a related record
   */
  (function () {

    var record = cr.get();

    // Get the Sales Rep record ID
    var repId = record.getValue({
      "fieldId": "salesrep"
    });

    // Retrieve the email address from the associated Sales Rep
    var repData = s.lookupFields({
      "type": "employee",
      "id": repId,
```



```

        "columns": ["email", "firstname"]
    });

    console.log(repData);
    console.log(repData.firstname + "'s email address is " + repData.email);
}());
});

```

## [2.0] Lookup Joined Fields

```

require(["N/search", "N/currentRecord"], function (s, cr) {

    /**
     * An example of N/search#lookupFields to retrieve joined fields from a related record
     */
    (function () {

        var record = cr.get();

        // Get the Sales Rep record ID
        var repId = record.getValue({
            "fieldId": "salesrep"
        });

        // Retrieve the email address from the associated Sales Rep
        var repData = s.lookupFields({
            "type": "employee",
            "id": repId,
            "columns": ["email", "firstname", "department.name"]
        });

        console.log(repData);
        console.log(repData.firstname + "'s email address is " + repData.email);
        console.log(repData.firstname + "'s department is " + repData["department.name"]);
    }());
});

```

Read Lookup Data from Related Records online: <https://riptutorial.com/netsuite/topic/9068/lookup-data-from-related-records>

---

# Chapter 10: Mass Delete

## Introduction

This sample shows how to mass delete records in NetSuite by leveraging the Mass Update feature. Typically, we're told not to delete records, but to make records inactive, but if you must, then this small script does just that. Once the script is deployed as a 'Mass Update' script type, simply go to Lists > Mass Update > Mass Updates > Custom Updates. You should see your mass delete. Next, set up your search criteria in your mass delete and do a preview to validate your data before deleting.

## Examples

### Delete based on Search Criteria

```
/**
 * NetSuite will loop through each record in your search
 * and pass the record type and id for deletion
 * Try / Catch is useful if you wish to handle potential errors
 */

function MassDelete(record_type, record_id)
{
    try
    {
        nlapiDeleteRecord(record_type, record_id)
    }
    catch (err)
    {
        var errorMessage = err;
        if(err instanceof nlobjError)
        {
            errorMessage = errorMessage + ' ' + err.getDetails() + ' ' + 'Failed to Delete ID : '
+ record_id;
        }
        nlapiLogExecution('ERROR', 'Error', errorMessage);
        return err
    }
}
```

Read Mass Delete online: <https://riptutorial.com/netsuite/topic/9062/mass-delete>

---

# Chapter 11: Requesting customField, customFieldList & customSearchJoin with PHP API Advanced Search

## Introduction

These were some of the hardest things (and least talked about) to do with the PHP API advanced search (where you specify what fields).

I'm in the process of migrating to rest\_suite github library that uses RESTLET, and get around the PHP API user concurrency limit of 1.

But before I delete my old code I'm posting it here. Example specs for these fields can be found here:

[http://www.netsuite.com/help/helpcenter/en\\_US/srbrowser/Browser2016\\_1/schema/search/transactionsearch](http://www.netsuite.com/help/helpcenter/en_US/srbrowser/Browser2016_1/schema/search/transactionsearch)

## Examples

### customField & customFieldList Usage

```
$service = new NetSuiteService();
$search = new TransactionSearchAdvanced();
$internalId = '123'; // transaction internalId

$search->criteria->basic->internalIdNumber->searchValue = $internalId;
$search->criteria->basic->internalIdNumber->operator = "equalTo";

$field = new SearchColumnSelectCustomField();
$field->scriptId = 'custbody_os_freight_company'; // this is specific to you & found in netsuite
$search->columns->basic->customFieldList->customField[] = $field;

$field = new SearchColumnStringCustomField();
$field->scriptId = 'custbody_os_warehouse_instructions'; // this is specific to you & found in netsuite
$search->columns->basic->customFieldList->customField[] = $field;

// and so on, you can keep adding to the customField array the custom fields you want

$request = new SearchRequest();
$request->searchRecord = $search;

$searchResponse = $service->search($request);
```

### customSearchJoin Usage

```
$service = new NetSuiteService();
$search = new TransactionSearchAdvanced();
```

```
$internalId = '123'; //transaction internalId

$search->criteria->basic->internalIdNumber->searchValue = $internalId;
$search->criteria->basic->internalIdNumber->operator = "equalTo";

$CustomSearchRowBasic = new CustomSearchRowBasic();
$CustomSearchRowBasic->customizationRef->scriptId = 'custbody_os_entered_by'; //this is
specific to you & found in netsuite
$CustomSearchRowBasic->searchRowBasic = new EmployeeSearchRowBasic();
$CustomSearchRowBasic->searchRowBasic->entityId = new SearchColumnStringField();

$search->columns->customSearchJoin[] = $CustomSearchRowBasic;
//and so on, you can keep adding to the customSearchJoin array the custom fields you want

$request = new SearchRequest();

$request->searchRecord = $search;

$searchResponse = $service->search($request);
```

Read Requesting customField, customFieldList & customSearchJoin with PHP API Advanced Search online: <https://riptutorial.com/netsuite/topic/9799/requesting-customfield--customfieldlist---customsearchjoin-with-php-api-advanced-search>

---

# Chapter 12: RESTlet - Process external documents

## Introduction

When retrieving a document from an external system, it requires us to ensure the correct document extension is affixed to the document. The sample code shows how to store a document properly in NetSuite's File Cabinet as well as attaching it to its corresponding record.

## Examples

### RESTlet - store and attach file

```
/**
 * data - passed in object
 * switch - get file extension if there is one
 * nlapiCreateFile - create file in File Cabinet
 * nlapiAttachRecord - attach file to record
 */

function StoreAttachFile(data)
{
    var record_type = data.recordType;
    var record_id = data.recordId;

    if(record_id && record_type == 'vendorbill')
    {
        try
        {
            var file_type = data.fileType;
            var file_extension;

            switch (file_type)
            {
                case "pdf":
                    file_extension = "pdf";
                    break;
                case "docx":
                    file_extension = "doc";
                    break;
                case "txt":
                    file_extension = "txt";
                    break;
                case "JPGIMAGE":
                    file_extension = "jpg";
                    break;
                case "png":
                    file_extension = "png";
                    break;
                default:
                    // unknown type
                    // there should probably be some error-handling
            }
        }
    }
}
```

```

    }

    var file_name = data.fileName + "." + file_extension;
    var file = data.fileContent;

    var doc = nlapiCreateFile(file_name, file_type, file);
    doc.setFolder(115); //Get Folder ID from: Documents > File > File Cabinet

    var file_id = nlapiSubmitFile(doc);

    nlapiAttachRecord("file", file_id, record_type, record_id);
    nlapiLogExecution('DEBUG', 'after submit', file_id);
}

catch (err)
{
    var errorMessage = err;
    if(err instanceof nlobjError)
    {
        errorMessage = errorMessage + ' ' + err.getDetails();
    }
    nlapiLogExecution('DEBUG', 'Error', errorMessage)
}
}
return true;
}

```

Read RESTlet - Process external documents online:

<https://riptutorial.com/netsuite/topic/9021/restlet---process-external-documents>

---

# Chapter 13: RestLet - Retrieve Data (Basic)

## Introduction

This sample shows the basic structure of a RESTlet script that is intended to be used to retrieve data from an external system. RESTlets are endpoints that are created to allow communication with external systems.

## Examples

### Retrieve Customer Name

```
/**
 * requestdata - the data packet expected to be passed in by external system
 * JSON - data format exchange
 * stringify() convert javascript object into a string with JSON.stringify()
 * nlobjError - add in catch block to log exceptions
 */

function GetCustomerData(requestdata)
{
    var jsonString = JSON.stringify(requestdata);
    nlapiLogExecution('DEBUG', 'JSON', jsonString);

    try
    {
        var customer = requestdata.customer;
        nlapiLogExecution('DEBUG', 'customer', customer);
    }
    catch (err)
    {
        var errorMessage = err;
        if(err instanceof nlobjError)
        {
            errorMessage = errorMessage + ' ' + err.getDetails() + ' ' + errorMessage;
        }
        nlapiLogExecution('DEBUG', 'Error', errorMessage);
    }
}
```

Read RestLet - Retrieve Data (Basic) online: <https://riptutorial.com/netsuite/topic/9006/restlet---retrieve-data--basic->

---

# Chapter 14: Script and Script Deployment Records

## Introduction

In order for NetSuite to know how to utilize our source code, we need to be able to tell it which functions to call, when to call them, and who to call them for. We accomplish all of these with the *Script* and *Script Deployment* records.

## Examples

### Script Records

NetSuite uses the *Script* record to map the function(s) in your source file to specific events that occur in the system. For instance, if you need some business logic to run when a form is saved in the UI, the Script record will tell NetSuite which function to call when the `Save Record` event occurs.

You can think of the *Script* record as defining *when* our source code should run; it essentially defines something akin to:

"When a record is saved, call the `saveRecord` function in `hello-world.js`."

Here is an example of what that Script record would look like:



# Script

[Edit](#)[Back](#)[Deploy Script](#)[Actions](#) ▼

TYPE

Client

DESCRIPTION

NAME

Hello World

OWNER

Alex Wo

ID

customscript595

 INAC

API VERSION

1.0

[Scripts](#)[Parameters](#)[Unhandled Errors](#)[Execution Log](#)[Deployments](#)[System](#)

SCRIPT FILE

preview [hello-world.js](#) [download](#) [Edit](#)

PAGE INIT FUNCTION

SAVE RECORD FUNCTION

saveRecord

VALIDATE FIELD FUNCTION

FIELD CHANGED FUNCTION

POST SOURCING FUNCTION

LINE INIT FUNCTION

## Script Deployment Records

Once we have a *Script* record created, we then need to deploy that script into the system. While the *Script* record tells NetSuite which functions to call from our source file, the *Script Deployment* record lets NetSuite know which records and users our Script should execute for.

While the *Script* record defines *when* our source code should run, the *Script Deployment* defines *where* and *who* can run our script. If we have a *Script* record that says:

"When a record is saved, call the saveRecord function in hello-world.js."

then our *Script Deployment* for that record might modify that slightly to:

"When an Employee record is saved, call the saveRecord function in hello-world.js, but only for users in the Administrators group."

Again, here is an example of what that *Script Deployment* would look like:

## Script Deployment

[Edit](#) | [Back](#) | [Actions](#) ▾

SCRIPT

Hello World

APPLIES TO

Employee

ID

customdeploy\_hello\_world

DEPLOYED

<a href="#">Audience</a>	<a href="#">Scripts</a>	<a href="#">Execution Log</a>	<a href="#">System Notes</a>
ROLES			SUBSIDIARIES
Administrator			
<input type="checkbox"/> ALL ROLES			GROUPS
DEPARTMENTS			EMPLOYEES

A *Script* can have multiple *Script Deployments* associated to it. This allows us to deploy the same business logic to multiple different record types with varying audiences.

Read [Script and Script Deployment Records](#) online:

<https://riptutorial.com/netsuite/topic/8835/script-and-script-deployment-records>

---

# Chapter 15: Script Type Overview

## Introduction

You create SuiteScript customizations using an event-driven system. You define various types of Script records, each of which has its own unique set of events, and in your source file, you define functions that will be called to handle those events as they occur.

Scripts are one of the primary components with which you'll design and build your applications. The goal with this article is merely to become acquainted with the Script types and events available.

## Examples

### The Client Script

The Client Script is one of the more commonly used, and complex, script types available to you. As its name implies, the Client Script runs in the browser, i.e. on the client side. It is the only script type that runs on the client side; all others will execute on the server side of NetSuite.

The primary use of the Client Script is for responding to user interactions with record forms within the NetSuite UI.

As soon as the user loads a record form in Edit mode, a `pageInit` event is fired that we can use to run code as the form is initialized, before the user can interact with it.

Whenever the user then changes any field on the form, a series of events will fire:

1. A `validateField` event fires that allows us to validate the value the user is trying to enter in the field. We can use this to either accept or prevent the change from taking place.
2. A `fieldChanged` event then fires that allows us to respond to the new value in the field.
3. Lastly, a `postSourcing` event fires after any and all dependent fields have also sourced in their values. This allows us to respond to the change *and* make sure we are working with all of the correct data.

This series of events fires no matter whether the user is changing a body field or a sublist field.

As the user does make changes to sublist lines, another series of events will be triggered:

1. A `lineInit` event is fired whenever the user initially selects a new or existing line, before they are able to make any changes to the fields on the line.
2. Whenever the user clicks the *Add* button to add a new line, a `validateLine` event is fired that allows us to verify that the entire line is valid and can be added to the record.
3. Whenever the user clicks the *Insert* button to add a new line above an existing one, a `validateInsert` event is fired, which works exactly like the `validateLine` event.
4. Similarly, whenever the user tries to remove a line, a `validateDelete` is fired that allows to

either allow or deny the removal of the line.

5. [SuiteScript 1.0 only] Lastly, after the appropriate validation event succeeds, if the change to the line also effected a change to the total amount of a transaction, then a `recalc` event is fired that allows us to respond to the change in amount of our transaction.
6. [SuiteScript 2.0 only] Lastly, after the appropriate validation event succeeds, a `sublistChanged` event is fired to allow us to respond to the completed line change.

Finally, when the user clicks the `Save` button on the record, a `saveRecord` event is fired that allows us to validate whether the record is valid and can be saved. We can either prevent the save from occurring, or allow it to proceed with this event.

The Client script has by far the most events of any Script type, and the most complex relationship between those events.

## The User Event Script

Closely related to the Client Script is the User Event Script. The events of this Script type are again fired when a record is being loaded or saved, but it instead runs on the server side. As such, it cannot be used to respond immediately to field changes, but it also is not limited to only users interacting with the record on a form.

User Event scripts will execute no matter where the load or submit request is coming from, whether it's a user working in the UI, a third-party integration, or another internal Script making the request.

Whenever a process or user attempts to read a record out of the database, the User Event's `beforeLoad` event is triggered. We can use this to pre-process data, set default values, or manipulate the UI form before the user sees it.

Once a process or user attempts to submit a record to the database, whether it's the creation of a new record, editing of an existing record, or the deletion of a record, the following sequence occurs:

1. First, before the request actually makes its way to the database, a `beforeSubmit` event fires. We can use this event, for example, to clean up the record before it gets in the database.
2. The request is sent to the database, and the record is created/modified/deleted accordingly.
3. After the database processing is complete, an `afterSubmit` event fires. We can use this event, for example, to send out email notifications of changes, or to sync up with integrated third-party systems.

You can also watch [this series of videos](#) that help to visualize the events of this script type.

## The Scheduled and Map/Reduce Scripts

There are two types of scripts we can leverage for running background processing on a specific, regular interval; these are the *Scheduled* and the *Map/Reduce* scripts. Note that the *Map/Reduce* script type is only available in SuiteScript 2.0. The *Scheduled* script is available for both 1.0 and 2.0.

The Scheduled script only has a single `execute` event that gets triggered on whatever schedule you define. For example, you may want to run a nightly script that applies payments to invoices, or an hourly script that syncs data with an external system. When the time interval hits, NetSuite fires this `execute` event on your Scheduled script.

The Map/Reduce script works similarly, but once it is triggered, it breaks the processing into four distinct phases:

1. The `getInputData` phase is where you gather all of the input data you will need to complete the business process. You can use this phase to perform searches, read records, and package your data into a decipherable data structure.
2. NetSuite automatically passes the results of your `getInputData` phase to the second phase, called `map`. This phase is responsible for grouping your input data logically for processing. For instance, if you're applying payments to invoices, you may want to first group the invoices by Customer.
3. The results of the `map` phase are then passed to the `reduce` phase, which is where the actual processing takes place. This is where you would, keeping with our example, actually apply the Payments to the Invoices.
4. Lastly, a `summary` phase is invoked that contains data regarding the results of all your processing across the previous three phases. You can use this to generate reports or send out emails that processing is complete.

The major advantage of the Map/Reduce script is that NetSuite will automatically parallelize the processing for you across multiple queues, if available.

Both of these script types have an extremely large governance limit, so you can also use them for bulk processing or generally long-running background processes.

The shortest interval either of these script types can be configured to run is every 15 minutes.

Both of these script types can also be invoked on-demand by users or by other scripts, if necessary.

## The Suitelet and Portlet Scripts

Often we will want to build custom UI pages in NetSuite; enter the Suitelet. The Suitelet script is designed for building internal, custom UI pages. Pages can be free-form HTML, or they can utilize NetSuite's UI Builder APIs to construct forms that follow NetSuite's look and feel.

When it is deployed, a Suitelet receives its own unique URL. The Suitelet then has a single `render` event that is called whenever that URL is hit with an HTTP `GET` or `POST` request. Typically, the response to the `GET` request would be to render the form itself, and then the form would `POST` back to itself for processing the form data.

We can also leverage Suitelets to build wizard-style UI progressions using NetSuite's "Assistant" UI components.

Portlets are extremely similar to Suitelets, except that they are specifically used to build custom dashboard widgets rather than full custom pages. Other than that, the two script types function

very much alike.

## The RESTlet

RESTlets allow us to build custom REST-based endpoints into NetSuite; thus, RESTlets form the backbone of nearly any integration into NetSuite.

RESTlets provide individual event handlers for four of the most commonly used HTTP request methods:

- GET
- POST
- PUT
- DELETE

When a RESTlet receives a request, it will route the request to the appropriate event handler function based on the HTTP request method used.

Authentication to a RESTlet can be done via user session, HTTP headers, or OAuth tokens.

## The Mass Update Script

Using the Mass Update script, we can build custom Mass Updates for users to perform. This functions just like a normal Mass Update, where the user selects the type of Mass Update, builds a search that returns the records to update, and then each search result is passed individually into the custom Mass Update script.

The script provides a single `each` event handler that receives the internal ID and record type of the record that is to be updated.

Mass Update scripts must be triggered manually by users through the standard Mass Update interface.

Mass Update scripts have a massively high governance limit and are intended for commonly used, custom bulk processing.

## The Workflow Action Script

Workflows can be somewhat limited in their functionality; for example, workflows cannot interact with line items. The Workflow Action script type is intended to be invoked by a Workflow to add scripting functionality to accomplish what the workflow itself cannot.

Workflow Actions have a single `onAction` event handler that will be invoked by the Workflow.

## The Bundle Installation Script

Lastly, we have the Bundle Installation script type, which provides several events that allow us to interact with the installation, update, and uninstallation of a particular bundle. This is a rarely-encountered script type, but important to be aware of nonetheless.

The Bundle Installation includes the following event handlers, which should be fairly self-explanatory:

- `beforeInstall`
- `afterInstall`
- `beforeUpdate`
- `afterUpdate`
- `beforeUninstall`

Read Script Type Overview online: <https://riptutorial.com/netsuite/topic/7829/script-type-overview>

---

# Chapter 16: Scripting searches with Filter Expressions

## Introduction

When you create searches with Suitescript, you could provide as "filters" either array of Filter objects, or filter expression. The second option is more readable and gives you very flexible option to provide nested expressions (up to 3 levels) using not only the default "AND", but also, "OR" and "NOT" operators.

## Examples

### Filter term

To understand the filter expressions, we should start with Filter Term. This is a simple **array of strings**, containing at least 3 elements:

1. **Filter** (Field/Join field/Formula/Summary)
2. **Operator** (search.Operator)
3. **Values** (string value(or array of string values), to be used as filter parameter)

```
// Simple example:
['amount', 'equalto', '0.00']

// When the field is checkbox, use 'T' or 'F'
['mainline', 'is', 'T']

// You can use join fields
['customer.companyname', 'contains', 'ltd']

// summary filter term
['sum(amount)', 'notlessthan', '170.50']

// summary of joined fields
['sum(transaction.amount)', 'greatherthan', '1000.00']

// formula:
["formulatext: NVL({fullname},'John')", "contains", "ohn"]

// and even summary formula refering joined fields:
['sum(formulanumeric: {transaction.netamount} + {transaction.taxtotal})',
'greaterthanorequalto','100.00']

// for selection fields, you may use 'anyof'
// and put values in array
['type', 'anyof', ['CustInvc', 'VendBill', 'VendCred']]

// when using unary operator, like isempty/isnotempty
// don't forget that the filter term array contains at least 3 elements
// and put an empty string as third:
```



```

['email', 'isnotempty', '']

// you may have more than 3 elements in Filter Term array,
// when the operator requires more than one value:
['grossamount', 'between', '100.00', '200.00']

```

In some selector fields, you can use special values.

```

// When filtering the user related fields, you can use:
// Me (Current user): @CURRENT@
// My Team (somebody from the team I am leading): @HIERARCHY@
['nextapprover', 'is', '@CURRENT@']

// The same is valid for Subsidiary, Department, Location, etc.
// @CURRENT@ means MINE (Subsidiary, Department...)
// @HIERARCHY@ means MINE or DESCENDANTS
['subsidiary', 'is', '@HIERARCHY@']

// on selection fields you may use @ANY@ and @NONE@
['nextapprover', 'is', '@NONE@']

```

## Filter expression

Simple filter expression is also an **array**. It contains one or more filter terms, combined with operators - 'AND', 'OR', 'NOT'. (Operators are case insensitive):

```

[
  ['mainline', 'is', 'T'],
  'and', ['type', 'anyof', ['CustInvc', 'CustCred']],
  'and', 'not', ['amount', 'equalto', '0.00'],
  'or', ['customer.companyname', 'contains', 'ltd']
]

```

More complex filter expressions, could contain filter terms **AND** nested filter expressions, combined with operators. No more than 3 levels of nested expressions are allowed:

```

[
  ['mainline', 'is', 'T'],
  'and', ['type', 'anyof', ['CustInvc', 'CustCred']],
  'and', [ ['customer.companyname', 'contains', 'ltd'],
          'or', ['customer.companyname', 'contains', 'inc']
        ],
  'and', [ ['subsidiary', 'is', 'HQ'],
          'or', ['subsidiary', 'anyof', '@HIERARCHY@']
        ],
  'and', ['trandate', 'notbefore', 'yesterday']
]

```

And finally, let's put all this altogether in a SS2.0 sample:

```

var s = search.create({
  type    : 'transaction',
  columns : [
    'trandate',

```

```

        'trandid',
        'currency',
        'customer.companyname',
        'customer.country',
        'amount'
    ],
    filters : [
        ['mainline', 'is', 'T'],
        'and', ['type', 'anyof', ['VendBill', 'VendCred']],
        'and', [ ['customer.companyname', 'contains', 'ltd'],
                'or', ['customer.companyname', 'contains', 'inc']
            ],
        'and', [ ['subsidiary', 'is', 'HQ'],
                'or', ['subsidiary', 'anyof', '@HIERARCHY@']
            ],
        'and', ['trandate', 'notbefore', 'yesterday']
    ]
});

```

## Filter expressions vs Filter Objects

Filter expressions **cannot include** Filter Objects. This is very important. If you decide to form your filters with Filter Expression, you use array of string arrays. The following syntax is **wrong**:

```

// WRONG!!!
var f1 = search.createFilter({
    name: 'mainline',
    operator: search.Operator.IS,
    values: 'T'
});

var f2 = search.createFilter({
    name: 'type',
    operator: search.Operator.ANYOF,
    values: ['VendBill', 'VendCred']
});

// here you will receive an error message
var s = search.create({
    type : 'transaction',
    filters : [ f1, 'and', f2 ] // f1,f2 are Filter Objects, instead of string arrays
});

```

Instead, use the **correct**:

```

// CORRECT!!!
var f1 = ['mainline', search.Operator.IS, 'T'];

var f2 = ['type', search.Operator.ANYOF, ['VendBill', 'VendCred'] ];

var s = search.create({
    type : 'transaction',
    filters : [ f1, 'and', f2 ]
});

```

or if you want to keep with Filter Objects approach, pass an array of filter objects, and forget about

operators 'AND', 'OR', 'NOT'. It will be always **AND**

```
// correct, but not useful
var f1 = search.createFilter({
    name: 'mainline',
    operator: search.Operator.IS,
    values: 'T'
});

var f2 = search.createFilter({
    name: 'type',
    operator: search.Operator.ANYOF,
    values: ['VendBill', 'VendCred']
});

var s = search.create({
    type : 'transaction',
    filters : [ f1, f2 ] // here you have array of Filter Objects,
                        // filtering only when all of them are TRUE
});
```

## Useful hints

1. Here you can find the list of available search filter values for date fields:

[https://system.netsuite.com/app/help/helpcenter.nl?fid=section\\_N3010842.html](https://system.netsuite.com/app/help/helpcenter.nl?fid=section_N3010842.html)

These you can use in expressions like:

```
['trandate', 'notbefore', 'daysAgo17']
```

2. Here are the search operators:

[https://system.netsuite.com/app/help/helpcenter.nl?fid=section\\_N3005172.html](https://system.netsuite.com/app/help/helpcenter.nl?fid=section_N3005172.html)

Of course you can use **search.Operator** enum:

[https://system.netsuite.com/app/help/helpcenter.nl?fid=section\\_4345782273.html](https://system.netsuite.com/app/help/helpcenter.nl?fid=section_4345782273.html)

3. Here are the search summary types:

[https://system.netsuite.com/app/help/helpcenter.nl?fid=section\\_N3010474.html](https://system.netsuite.com/app/help/helpcenter.nl?fid=section_N3010474.html)

4. You can use ANYOF operator only on select type fields (List/Record). If you want to use it against free-text fields (like names, emails etc.), the only way is to create a nested Filter Expression with 'OR' operators:

```
[ ['email', 'startswith', 'user1@abcd.com'],
  'or', ['email', 'startswith', 'user2@abcd.com'],
  'or', ['email', 'startswith', 'user3@abcd.com'],
  'or', ['email', 'startswith', 'user4@abcd.com']
]
```

or you can write small script, doing this instead of you:

```
function stringFieldAnyOf(fieldId, listOfValues) {
    var result = [];
```

```

if (listOfValues.length > 0) {
    for (var i = 0; i < listOfValues.length; i++) {
        result.push([fieldId, 'startswith', listOfValues[i]]);
        result.push('or');
    }
    result.pop(); // remove the last 'or'
}
return result;
}

// usage: (two more filters added just to illustrate how to combine with other filters)
var custSearch = search.create({
    type: record.Type.CUSTOMER,
    columns: searchColumn,
    filters: [
        ['companyname', 'startswith', 'A'],
        'and', stringFieldAnyOf('email', ['user1@abcd.com', 'user2@abcd.com']),
        'and', ['companyname', 'contains', 'b']
    ]
});

```

### 5. Still not confident? Looking for a cheat? :)

Create a saved search in the Netsuite UI, take the search ID (lets say: customsearch1234) and log.debug the filter expression:

```

var s = search.load('customsearch1234');

log.debug('filterExpression', JSON.stringify(s.filterExpression));

```

Read Scripting searches with Filter Expressions online:

<https://riptutorial.com/netsuite/topic/10732/scripting-searches-with-filter-expressions>

---

# Chapter 17: Searches with large number of results

## Introduction

Suitescript 2.0 provides 4 methods to handle the search results.

They have different syntax, limitations and governance, and are appropriate for different situations. We will focus here on how to access **ALL** search results, using each of these methods.

## Examples

### Using Search.ResultSet.each method

This is shortest, easiest and most commonly used method. Unfortunately, it has one major limitation - cannot be used on searches with more than 4000 results (rows).

```
// Assume that 'N/search' module is included as 'search'

var s = search.create({
  type : search.Type.TRANSACTION,
  columns : ['entity','amount'],
  filters : [ ['mainline', 'is', 'T'],
             'and', ['type', 'is', 'CustInvc'],
             'and', ['status', 'is', 'open']
           ]
});

var resultSet = s.run();

// you can use "each" method on searches with up to 4000 results
resultSet.each( function(result) {

  // you have the result row. use it like this....
  var transId = result.id;
  var entityId = result.getValue('entity');
  var entityName = result.getText('entity');
  var amount = result.getValue('amount');

  // don't forget to return true, in order to continue the loop
  return true;

});
```

### Using ResultSet.getRange method

In order to use getRange for handling the large number of results, we will have to consider the following:

1. getRange has 2 parameters: **start** and **end**. Always positive, always (start < end)

2. **start** is the inclusive index of the first result to return
3. **end** is the exclusive index of the last result to return
4. If there are fewer results available than requested, then the array will contain fewer than end - start entries. For example, if there are only 25 search results, then `getRange(20, 30)` will return an array of 5 `search.Result` objects.
5. Although the above help sentence doesn't say it directly, both **start** and **end** could be outside the range of available results. In the same example - if there are only 25 search results, `getRange(100, 200)` will return an empty array [ ]
6. Maximum 1000 rows at a time.  $(end - start) \leq 1000$

```
// Assume that 'N/search' module is included as 'search'

// this search will return a lot of results (not having any filters)
var s = search.create({
  type: search.Type.TRANSACTION,
  columns : ['entity','amount'],
  filters: []
});

var resultSet = s.run();

// now take the first portion of data.
var currentRange = resultSet.getRange({
  start : 0,
  end : 1000
});

var i = 0; // iterator for all search results
var j = 0; // iterator for current result range 0..999

while ( j < currentRange.length ) {

  // take the result row
  var result = currentRange[j];
  // and use it like this....
  var transId = result.id;
  var entityId = result.getValue('entity');
  var entityName = result.getText('entity');
  var amount = result.getValue('amount');

  // finally:
  i++; j++;
  if( j==1000 ) { // check if it reaches 1000
    j=0; // reset j and reload the next portion
    currentRange = resultSet.getRange({
      start : i,
      end : i+1000
    });
  }
}
}
```

Lets calculate the Governance. We have  $1 + \text{count}/1000$  `getRange` calls taking 10 units each, so:

$$G = (1 + \text{count}/1000) * 10$$

Example: 9500 rows will take 100 units

## Using Search.PagedData.fetch method

PagedData is an object, returned by the Search.runPaged(options) method. It works exactly as the UI searches do. PagedData object contains 2 important properties, that you can see on the right side of results header in search results page in Netsuite UI:

- **count** (the total number of the results)
- **pageRanges** (list of pages, available in UI as combo-box selector)

options.pageSize parameter is limited again to 1000 result rows.

**PagedData.fetch** method is used to fetch the result portion you want (indexed by pageIndex parameter). With a little bit more code, you receive the same convenient callback function as Search.ResultSet.each, without having the 4000 rows limitation.

```
// Assume that 'N/search' module is included as 'search'

// this search will return a lot of results (not having any filters)
var s = search.create({
  type: search.Type.TRANSACTION,
  columns : ['entity','amount'],
  filters : []
});

var pagedData = s.runPaged({pageSize : 1000});

// iterate the pages
for( var i=0; i < pagedData.pageRanges.length; i++ ) {

  // fetch the current page data
  var currentPage = pagedData.fetch(i);

  // and forEach() thru all results
  currentPage.data.forEach( function(result) {

    // you have the result row. use it like this....
    var transId = result.id;
    var entityId = result.getValue('entity');
    var entityName = result.getText('entity');
    var amount = result.getValue('amount');

  });
}
```

Lets calculate the Governance. We have 5 units for runPaged(), and 1 + count/1000 pagedData.fetch calls taking 5 units each, so:

$$G = 5 + \text{ceil}(\text{count}/1000) * 5$$

Example: 9500 rows will take 55 units. Approximately half of the getRange governance units.

## Using dedicated Map/Reduce script

For really huge search results, you can use dedicated Map/Reduce script. It is much more

inconvenient, but sometimes unavoidable. And sometimes could be very handy.

The trick here is, that in Get Input Data stage, you can provide to the NS engine not the actual data (i.e. script result), but just the definition of the search. NS will execute the search for you without counting the governance units. Then each single result row will be passed to the Map stage.

Of course, there is a limitation: The total persisted size of data for a map/reduce script is not allowed to exceed 50MB. In a search result, each key and the serialized size of each value is counted towards the total size. "Serialized" means, that the search result row is converted to string with `JSON.stringify`. Thus, the value size is proportional to the number of search result columns in a result set. If you get to trouble with `STORAGE_SIZE_EXCEEDED` error, consider reducing the columns, combining to formulas, grouping the result or even splitting the search to multiple sub-searches, which could be executed in Map or Reduce stages.

```
/**
 * @NApiVersion 2.0
 * @NScriptType MapReduceScript
 */
define(['N/search'], function(search) {

function getInputData()
{
    return search.create({
        type: search.Type.TRANSACTION,
        columns : ['entity','amount'],
        filters : []
    });
}

function map(context)
{
    var searchResult = JSON.parse(context.value);
    // you have the result row. use it like this...
    var transId = searchResult.id;
    var entityId = searchResult.values.entity.value;
    var entityName = searchResult.values.entity.text;
    var amount = searchResult.values.amount.value;

    // if you want to pass some part of the search result to the next stage
    // write it to context:
    context.write(entityId, transId);
}

function reduce(context)
{
    // your code here ...
}

function summarize(summary)
{
    // your code here ...
}

return {
    getInputData: getInputData,
    map: map,
    reduce: reduce,
    summarize: summarize
}
```



```
};  
});
```

Of course the example here is simplified, without error handling and is given just to be compared with others. More examples are available at [Map/Reduce Script Type examples in NS Help Center](#)

Read Searches with large number of results online:

<https://riptutorial.com/netsuite/topic/10687/searches-with-large-number-of-results>

---

# Chapter 18: Sourcing

## Parameters

Parameter	Details
Source List	The field on the destination record which links to the source record. You must choose a source list before you can choose your source field.
Source From	The field on the source record from which data will actually be pulled. The field you choose must match the type of the destination field. For example, if you are sourcing from a <i>Phone Number</i> field, the destination field must be a <i>Phone Number</i> field as well.

## Remarks

### Impact of *Store Value*

The *Store Value* setting on the custom field definition plays a very important role in the behaviour of Sourcing:

- When *Store Value* is **checked**, data is sourced into the field *only upon initial creation* of the record. After that, NetSuite breaks the sourcing link between the fields, and they become two independent fields. This effectively allows you to leverage Sourcing as a mechanism for setting the initial or default value of your custom field.
- When *Store Value* is **unchecked**, data is sourced dynamically into the field **every time the record is loaded**. Any changes a user or script might make to the field are **never saved**. If you leave *Store Value* unchecked, it is a good idea to make your field read-only.

### Limitations of Sourcing

- Sourcing cannot be applied to *native NetSuite fields*. If you need a native field as your destination field, then you will need to either create a workflow or write a script to perform the data sourcing.
- Sourcing cannot be applied to *sublist column fields*. If you need a sublist column as your destination field, then you will need to either create a workflow or write a script to perform the data sourcing.

## Examples

### Pulling data into a custom field on Field Changed

```
// If you find yourself doing something like this...
function fieldChanged(type, name, index) {
    if (name == 'salesrep') {
        var salesRepId = nlapiGetFieldValue('salesrep');
        var salesRepEmail = nlapiLookupField('employee', salesRepId, 'email');
        nlapiSetFieldValue('custbody_salesrep_email', salesRepEmail);
    }
}
// Stop! and consider using Sourcing for your custom field instead of code
```

## Defining Sourcing

While not strictly a SuiteScript topic, *Sourcing* is an incredibly powerful feature of NetSuite, and it's an important tool in the toolbelt for any SuiteScript developer. Sourcing allows us to *pull data into a record from any of its related records*, without writing any code or building a workflow to do so.

Less code is always more maintainable code.

Sourcing is defined on the *Sourcing & Filtering* tab of a Custom Field definition.

<p><b>LABEL *</b></p> <input type="text" value="Supervisor's Phone"/>	<p><b>TYPE</b></p> <input type="text" value="Phone Number"/>
<p><b>ID</b></p> <input type="text" value="_supervisor_phone"/>	<p><b>LIST/RECORD</b></p> <input type="text" value=""/>
<p><b>OWNER</b></p> <input type="text" value="Alex Wolfe"/>	<p><input checked="" type="checkbox"/> STORE VALUE   <input type="checkbox"/> USE ENCRYPTED FORMAT</p> <p><input type="checkbox"/> SHOW IN LIST</p>
<p><b>DESCRIPTION</b></p> <input type="text" value="The Phone Number of this Employee's direct Supervisor."/>	

---

Applies To	Display	Validation & Defaulting	Sourcing & Filtering	Access	Translation
------------	---------	-------------------------	----------------------	--------	-------------

<p><b>SOURCE LIST</b></p> <input type="text" value="Supervisor"/>	<p><b>SOURCE FROM</b></p> <input type="text" value="Phone"/>
---	--

Read Sourcing online: <https://riptutorial.com/netsuite/topic/7034/sourcing>

---

# Chapter 19: SS2.0 Suitelet Hello World

## Examples

### Basic Hello World Suitelet - Plain Text Response

```
/**
 *@NApiVersion 2.x
 *@NScriptType Suitelet
 */

define([],function() { // NetSuite's AMD pattern
    function onRequest_entry(context) { // Suitelet entry function receives a context obj
        context.response.write('Hello World'); // Write a response using the context obj
    }
    return {
        onRequest: onRequest_entry // Function assigned to entry point
    };
});
```

Read SS2.0 Suitelet Hello World online: <https://riptutorial.com/netsuite/topic/6723/ss2-0-suitelet-hello-world>

---

# Chapter 20: SuiteScript - Process Data from Excel

## Introduction

Sometimes the returned search results in a Mass Update isn't the same as the results in a standard search, this is due to some limitations in a Mass Update Search. An example of this is Rev Rec Journal entries. Therefore, the workaround for this was to get the data from the standard saved search and use a script to read the excel data and update, as opposed to using the mass update feature.

## Examples

### Update Rev Rec Dates and Rule

```
/**
 * Save the results from the saved search as .csv and store in file cabinet
 * Get the file's internal id when loading the file
 * Use \n to process each row
 * Get the internal id and whatever columns that need updating
 * Create a filtered search and pass the internal id
 * If the passed in internal id finds a record match, then update the rev rec dates and rule
 */

function ProcessSearchData()
{
    var loaded_file = nlapiLoadFile(4954);//loads from file cabinet
    var loaded_string = loaded_file.getValue();
    var lines = loaded_string.split('\n');//split on newlines
    nlapiLogExecution('DEBUG', 'lines', lines);
    var values;
    for (var i = 1; i < lines.length; i++)
    {
        nlapiLogExecution('DEBUG', 'count', i);
        values = lines[i].split(',')//split by comma
        var internal_id = values[0];//first column value
        nlapiLogExecution('DEBUG', 'internal_id', internal_id);
        var start_date = values[1];
        var end_date = values[2];

        if(internal_id)
        {
            UpdateDates(internal_id, start_date, end_date)
            nlapiLogExecution('DEBUG', '""REV REC PLANs UPDATED""');
        }
    }
    return true;
}

function UpdateDates(internal_id, start_date, end_date)
{

```

```

var filters = new Array();
filters[0] = new nlobjSearchFilter('internalid', null, 'is', internal_id);

var columns = [];
columns[0] = new nlobjSearchColumn('internalid');
columns[1] = new nlobjSearchColumn('revrecstartdate');
columns[2] = new nlobjSearchColumn('revrecenddate');

var rev_rec_plan = nlapiSearchRecord('revenueplan', null, filters, columns);
if(rev_rec_plan)
{
    for (var i = 0; rev_rec_plan != null && i < rev_rec_plan.length; i++)
    {
        var record = nlapiLoadRecord('revenueplan', rev_rec_plan[0].getValue(columns[0]));
        var id = record.getId();
        record.setFieldValue('revrecstartdate', start_date);
        record.setFieldValue('revrecenddate', end_date);
        record.setFieldValue('revenuerecognitionrule', 2)//Exact days based on Arrangement
dates
        nlapiSubmitRecord(record);
    }
}
return internal_id;
}

```

Read SuiteScript - Process Data from Excel online:

<https://riptutorial.com/netsuite/topic/9034/suitescript---process-data-from-excel>

# Chapter 21: Understanding Transaction Searches

## Introduction

A deep understanding of how Transaction searches function is crucial knowledge for every NetSuite developer, but the default behaviour of these searches, and controlling that behaviour, can be quite confusing initially.

## Remarks

References:

- NetSuite Help page: "Using Main Line in Transaction Search Criteria"

## Examples

### Filtering only on Internal ID

Let's explore an example Transaction search where we define a filter for a single transaction's internal ID:

## Transaction Search

|  ▼

USE ADVANCED SEARCH

### Criteria Results

Use this tab to specify criteria that narrow down your search.


USE EXPRESSIONS

**Standard** • Summary

FILTER \* DES

Internal ID is 875

We've specified a filter to only show us results for the Transaction with the internal ID of 875; here is that Transaction:

**Sales Order** 

# SLS00000162 Alex Wolfe PENDING BILLING

[Edit](#) [Back](#) | [Next Bill](#) [Bill](#) [Authorize Return](#) [Close Order](#) [Print Labels](#)

### Primary Information

CUSTOMER Alex Wolfe	PROMISE DATE
ORDER # SLS00000162	LOCATION
PO #	CLASS
TERMS	DEPARTMENT
DATE 1/15/2017	
JOB	
EMAIL	

**Items** [Billing](#) [Shipping](#) [Gross Profit](#) [Activities](#) [History](#) [Audit Trail/Workflow](#) [Quote Approval](#)

EXCHANGE RATE  
1.00

COUPON CODE

PROMOTION

ITEM	ON HAND	AVAILABLE	QTY	UM	DESCRIPTION	PRICE LEVEL
Cable - USB 10 ft	-98	0	39		10 ft USB A/B Cable	100% Sample Pricing

We can see it is a Sales Order with a single line item.

Because internal IDs are unique across all transactions, we can expect only one search result for this search. Here is the search result:



# Transaction Search: Results

List

Return To Criteria

Save This Search

+ FILTERS

EDIT   VIEW	INTERNAL ID	*	DATE ▲	AS-OF DATE	PERIOD	TAX PERIOD	TYPE
Edit   View	875	*	1/15/2017				Sales Order
Edit   View	875		1/15/2017				Sales Order
Edit   View	875		1/15/2017				Sales Order
Edit   View	875		1/15/2017				Sales Order

Instead of the single result we expect, we get *four* results. What's more, every result has exactly the same internal ID. How is that possible?

To understand what is happening here, we need to recall that data stored in NetSuite records is divided into two categories:

1. Body Data: Data stored in standalone fields of the record (e.g. Date, Sales Rep, Document Number, Coupon Code)
2. Sublist Data: Data stored in lists within each record, usually displayed on subtabs in the UI (e.g. Items on a Sales Order)

Transactions contain multiple sublists of data, including its:

- line items
- shipping information
- tax information
- COGS (Cost of Goods Sold) details

In these search results, NetSuite is actually showing us one result for the transaction body, then other results for data on the various sublists within that same transaction.

Notice the column in our search results simply named with an asterisk (\*). Notice also that one of the results has an asterisk populated in this column while the rest are empty. This column

indicates which search result represents the body of the transaction, which is also called the transaction's Main Line.

There are times when you will want transaction searches to only show the Main Line data, and times where you will only want the line-level detail. The remaining examples show how to control what shows up in our results.

## Filtering with Main Line

When we only want one result per transaction, that means we only want the Body, or Main Line, of each transaction. To accomplish this, there is a filter named "Main Line".

By setting the *Main Line* filter to *Yes* in our search criteria, we are essentially saying "Only show me body-level data for the transactions in my results":

The screenshot shows a search criteria configuration interface. At the top, there are two tabs: 'Criteria' (selected) and 'Results'. Below the tabs, there is a instruction: 'Use this tab to specify criteria that narrow down your search.' Below this, there is a checkbox labeled 'USE EXPRESSIONS' which is unchecked. Below the checkbox, there are two tabs: 'Standard' (selected) and 'Summary'. Below the tabs, there is a list of filters. The first filter is 'Internal ID' with a value of 'is 875'. The second filter is 'Main Line' with a value of 'is true'. Below the list, there is a search input field with a dropdown arrow. Below the input field, there are four buttons: 'Add' (with a checkmark icon), 'Cancel' (with an 'x' icon), 'Insert' (with a plus icon), and 'Remove' (with a trash can icon).






Modifying our previous search criteria this way now gives us the single result we expected originally:

# Transaction Search: Results

List

[Return To Criteria](#) [Save This Search](#)

+ FILTERS

         EDIT <input type="checkbox"/>					
EDIT   VIEW	INTERNAL ID	ACCOUNT	AMOUNT (DEBIT)	AMOUNT (CREDIT)	POSTING
Edit   View	875	Sales Orders	1,408.70		No

If we reverse our *Main Line* filter to *No*, we are saying "Show me only the data from sublists in my results":

# Transaction Search: Results

List

Return To Criteria

Save This Search

+ FILTERS



EDIT



EDIT   VIEW	INTERNAL ID	ACCOUNT	AMOUNT (DEBIT)	AMOUNT (CREDIT)	POSTING
Edit   View	875	4002 Sales : Sales - Merchandise		739.05	No
Edit   View	875	8030 Shipping Income		608.68	No
Edit   View	875	2050 Sales Taxes Payable		60.97	No

To recap *Main Line's* behaviour:

- With *Main Line* set to *Yes*, we received *one* result for *only the body* of the transaction.
- With *Main Line* set to *No*, we received *three* results for *only the sublist data* of the transaction.
- With no *Main Line* filter at all, we received *four* results, essentially the combination of all the body and sublist data for the transaction.

*Note that the Main Line filter is not supported for Journal Entry searches.*

## Filtering Specific Sublists

Recall that every transaction contains multiple sublists of data. Now that we can show only sublist data using *Main Line*, we can further refine our search results to specific sublist data.

Most of the sublists included in Transaction results have a corresponding search filter to toggle whether they are included in your results:

- Use the *Shipping Line* filter to control data from the Shipping sublist
- Use the *Tax Line* filter to control data from the Tax sublist
- Use the *COGS Line* filter to control data from the COGS sublist

Each of these filters behaves like *Main Line* or any other checkbox filter: *Yes* to include this data, *No* to exclude it from your results.

Notice that there is no filter for *Item Line* to control the data from the Item sublist. Essentially, in order to say "Only show me the data from the Items sublist", we need to specify all of these aforementioned filters as *No* in our criteria:

## Transaction Search

USE ADVANCED SEARCH

Criteria   Results

Use this tab to specify criteria that narrow down your search.

USE EXPRESSIONS

Standard • Summary

FILTER *	DESCRIPTION
Internal ID	is 87
Main Line	is fal
Tax Line	is fal
Shipping Line	is fal
COGS Line	is fal

With this criteria, your search will return one result per item line on each matching transaction.

In my opinion, this missing filter is a major gap in the search functionality that should be fixed; it would be much easier and more consistent to simply have an *Item Line is Yes* filter. Until then, this is how you must specify that you only want Item data in your transaction results.

[Read Understanding Transaction Searches online:](#)

<https://riptutorial.com/netsuite/topic/9012/understanding-transaction-searches>

---

# Chapter 22: User Event: Before and After Submit events

## Syntax

- `beforeSubmit(type)` // Before Submit, 1.0
- `beforeSubmit(scriptContext)` // Before Submit, 2.0
- `afterSubmit(type)` // After Submit, 1.0
- `afterSubmit(scriptContext)` // After Submit, 2.0

## Parameters

Parameter	Details
<i>SuiteScript 2.0</i>	-
<code>scriptContext</code>	{Object}
<code>scriptContext.newRecord</code>	{N/record.Record} A reference to the record that is being read from the database. We can use it to modify the field values on the record
<code>scriptContext.oldRecord</code>	{N/record.Record} A read-only reference to the previous state of the record. We can use it to compare to the new values
<code>scriptContext.type</code>	{UserEventType} An enumeration of the type of write action being performed
<i>SuiteScript 1.0</i>	-
<code>type</code>	{String} The type of write action being performed

## Remarks

---

`beforeSubmit` **and** `afterSubmit`

These two events are triggered by any database write operation on a record. Any time a user, a script, a CSV import, or a web service request attempts to write a record to the database, the Submit events get fired.

Record actions that trigger *both* Submit events:

- Create
- Edit

- Delete
- XEdit (inline edit)
- Approve
- Reject
- Cancel
- Pack
- Ship

Record actions that trigger `beforeSubmit` only:

- Mark Complete
- Reassign (support cases)
- Edit Forecast

Record actions that trigger `afterSubmit` only:

- Dropship
- Special Order
- Order Items
- Pay Bills

---

## Typical Use Cases for `beforeSubmit`

- Validate record before it is committed to database
- Permission and restriction checks
- Last-minute changes before database commit
- Pull updates from external systems

---

## Typical Use Cases for `afterSubmit`

- Email notification of record changes
- Browser redirection
- Create/update dependent records
- Push changes to external systems

---

## User Events do not chain

Code written in User Events will not trigger any User Events on *other* records. For example, modifying the associated Customer record from the `beforeSubmit` of a Sales Order record *will not trigger* the Customer record's submit events.

NetSuite does this to avoid User Events triggering each other in an infinite loop. If you *do* need User Events to fire in a chained sequence, other script types (e.g. RESTlets, Suitelets, Scheduled Scripts) will need to be injected in between the events.



# Event Handlers return `void`

The return type of the Submit event handlers is `void`. Any data returned from our event handler has no effect on the system. We do not need to return anything from our handler function as we cannot actually do anything with its returned value.

## !! CAUTION !!

Be very cautious when comparing values between old and new records. Empty fields from the *old* record are returned as `null`, while empty fields from the *new* record are returned as an empty String. This means you cannot simply compare the old with the new, or you will get false positives. Any logic you write must handle the case where one is `null` and one is an empty String appropriately.

## Examples

### Minimal: Log a message

```
// 1.0, Revealing Module pattern
var myNamespace = myNamespace || {};

myNamespace.example = (function () {

  /**
   * User Event 1.0 example detailing usage of the Submit events
   *
   * @appliedtorecord employee
   */
  var exports = {};

  function beforeSubmit(type) {
    nlapiLogExecution("DEBUG", "Before Submit", "action=" + type);
  }

  function afterSubmit(type) {
    nlapiLogExecution("DEBUG", "After Submit", "action=" + type);
  }

  exports.beforeSubmit = beforeSubmit;
  exports.afterSubmit = afterSubmit;
  return exports;
})();

// 2.0
define(["N/log"], function (log) {

  /**
   * User Event 2.0 example showing usage of the Submit events
   *
   * @NApiVersion 2.x
   * @NModuleScope SameAccount

```

```

    * @NScriptType UserEventScript
    * @appliedtorecord employee
    */
var exports = {};

function beforeSubmit(scriptContext) {
    log.debug({
        "title": "Before Submit",
        "details": "action=" + scriptContext.type
    });
}

function afterSubmit(scriptContext) {
    log.debug({
        "title": "After Submit",
        "details": "action=" + scriptContext.type
    });
}

exports.beforeSubmit = beforeSubmit;
exports.afterSubmit = afterSubmit;
return exports;
});

```

## Before Submit: Validate record before it is committed to database

For this example, we want to make sure that any Employee who is marked as a *Project Resource* also has an appropriate *Labor Cost* defined.

```

// 1.0, Revealing Module pattern
var myNamespace = myNamespace || {};
myNamespace.example = (function () {

    /**
     * User Event 1.0 example detailing usage of the Submit events
     *
     * @appliedtorecord employee
     */
    var exports = {};

    function beforeSubmit(type) {
        if (!isEmployeeValid(nlapiGetNewRecord())) {
            throw nlapiCreateError("STOIC_ERR_INVALID_DATA", "Employee data is not valid",
true);
        }
    }

    function isEmployeeValid(employee) {
        return (!isProjectResource(employee) || hasValidLaborCost(employee));
    }

    function isProjectResource(employee) {
        return (employee.getFieldValue("isjobresource") === "T");
    }

    function hasValidLaborCost(employee) {
        var laborCost = parseFloat(employee.getFieldValue("laborcost"));

        return (Boolean(laborCost) && (laborCost > 0));
    }

```

```

    }

    exports.beforeSubmit = beforeSubmit;
    return exports;
  }) ();

  // 2.0
  define(["N/error"], function (err) {

    var exports = {};

    /**
     * User Event 2.0 example detailing usage of the Submit events
     *
     * @NApiVersion 2.x
     * @NModuleScope SameAccount
     * @NScriptType UserEventScript
     * @appliedtorecord employee
     */
    function beforeSubmit(scriptContext) {
      if (!isEmployeeValid(scriptContext)) {
        throw err.create({
          "name": "STOIC_ERR_INVALID_DATA",
          "message": "Employee data is not valid",
          "notifyOff": true
        });
      }
    }

    function isEmployeeValid(scriptContext) {
      return (!isProjectResource(scriptContext.newRecord) ||
        hasValidLaborCost(scriptContext.newRecord));
    }

    function isProjectResource(employee) {
      return (employee.getValue({"fieldId" : "isjobresource"}));
    }

    function hasValidLaborCost(employee) {
      var laborCost = employee.getValue({"fieldId" : "laborcost"});

      return (Boolean(laborCost) && (laborCost > 0));
    }

    exports.beforeSubmit = beforeSubmit;
    return exports;
  });

```

Note that we pass references to the *new* record into our validation because we do not care what the values used to be; we are only concerned with the values that are about to be written to the database. In 2.0, we do that via the `scriptContext.newRecord` reference, and in 1.0 we call the global function `nlapiGetNewRecord`.

When the data being submitted is not valid, we create and throw an error. In a `beforeSubmit` event, in order to prevent the changes from being written to the database, your function must `throw` an Exception. Often developers try to `return false` from their function, expecting that to be enough, but that is not sufficient. Error objects are created in 2.0 using the `N/error` module, and in 1.0 using the global `nlapiCreateError` function; we then raise an Exception using our created error object with

the `throw` keyword.

## After Submit: Determine whether a field was changed

After the record gets stored in the database, we want to inspect what was changed on the record. We'll do this inspection by comparing values between the old and new record instances.

```
// 1.0, Revealing Module pattern
var myNamespace = myNamespace || {};
myNamespace.example = (function () {

  /**
   * User Event 1.0 example detailing usage of the Submit events
   *
   * @appliedtorecord employee
   */
  var exports = {};

  function afterSubmit(type) {
    notifySupervisor();
  }

  function notifySupervisor() {
    // Old and New record instances are retrieved from global functions
    var employee = nlapiGetNewRecord();
    var prevEmployee = nlapiGetOldRecord();

    // If Employee Status didn't change, there's nothing to do
    if (!didStatusChange(employee, prevEmployee)) {
      return;
    }

    // Otherwise, continue with business logic...
  }

  function didStatusChange(employee, prevEmployee) {
    var status = employee.getFieldValue("employeestatus");
    var prevStatus = prevEmployee.getFieldValue("employeestatus");

    /* !! Caution !!
     * Empty fields from the Old record come back as `null`
     * Empty fields from the New record come back as an empty String
     * This means you cannot simply compare the old and new
     */
    return ((prevStatus || status) && (status !== prevStatus));
  }

  exports.afterSubmit = afterSubmit;
  return exports;
})();

// 2.0
define(["N/runtime"], function (runtime) {

  /**
   * User Event 2.0 example detailing usage of the Submit events
   *
   * @NApiVersion 2.x
   * @NModuleScope SameAccount

```

```

    * @NScriptType UserEventScript
    * @appliedtorecord employee
    */
var exports = {};

function afterSubmit(scriptContext) {
    notifySupervisor(scriptContext);
}

function notifySupervisor(scriptContext) {
    // Old and New records are simply properties on scriptContext
    var employee = scriptContext.newRecord;
    var prevEmployee = scriptContext.oldRecord;

    // If Employee Status didn't change, there's nothing to do
    if (!didStatusChange(employee, prevEmployee)) {
        return;
    }

    // Otherwise, continue with business logic...
}

function didStatusChange(employee, prevEmployee) {
    var status = employee.getValue({"fieldId" : "employeestatus"});
    var prevStatus = prevEmployee.getValue({"fieldId" : "employeestatus"});

    /* !! Caution !!
    * Empty fields from the Old record come back as `null`
    * Empty fields from the New record come back as an empty String
    * This means you cannot simply compare the old and new
    */
    return ((prevStatus || status) && (status !== prevStatus));
}

exports.afterSubmit = afterSubmit;
return exports;
});

```

Be very cautious when comparing values between old and new records. Empty fields from the *old* record are returned as `null`, while empty fields from the *new* record are returned as an empty String. This means you cannot simply compare the old with the new, or you will get false positives. Any logic you write must handle the case where one is `null` and one is an empty String appropriately.

Read User Event: Before and After Submit events online:

<https://riptutorial.com/netsuite/topic/7200/user-event--before-and-after-submit-events>

---

# Chapter 23: User Event: Before Load event

## Parameters

Parameter	Details
<i>SuiteScript 2.0</i>	-
scriptContext	{Object}
scriptContext.newRecord	{N/record.Record} A reference to the record being loaded from the database
scriptContext.type	{UserEventType} The action type that triggered this User Event
scriptContext.form	{N/ui/serverWidget.Form} A reference to the UI form that will be rendered
<i>SuiteScript 1.0</i>	-
type	{Object} The action type that triggered this User Event
form	{nlobjForm} A reference to the UI form that will be rendered
request	{nlobjRequest} the HTTP GET request; only available when triggered by browser requests

## Remarks

---

### **beforeLoad**

The `Before Load` event is triggered by any read operation on a record. Any time a user, a script, a CSV import, or a web service request attempts to read a record from the database, the `Before Load` event gets fired.

Record actions that trigger a `beforeLoad` event:

- Create
- Edit
- View / Load
- Copy
- Print
- Email
- QuickView

# Typical Use Cases for `beforeLoad`

- Modify the UI form before the user sees it
- Set default field values
- Data pre-processing

---

## User Events do not chain

Code written in User Events will not trigger any User Events on *other* records. For example, loading the associated Customer record from the `beforeLoad` of a Sales Order record *will not trigger* the Customer record's `beforeLoad`. Even if you are loading another Transaction record, its User Events will not be fired.

NetSuite does this to avoid User Events triggering each other in an infinite loop. If you *do* need User Events to fire in a chained sequence, other script types (e.g. RESTlets, Suitelets, Scheduled Scripts) will need to be injected in between the events.

---

## Event Handler returns `void`

The return type of the `beforeLoad` event handler is `void`. Any data returned from our event handler has no effect on the system. We do not need to return anything from our handler function as we cannot actually do anything with its returned value.

## Examples

### Minimal: Log a message on Before Load

```
// 1.0
function beforeLoad(type, form, request) {
    nlapiLogExecution("DEBUG", "Before Load", "type=" + type);
}

// 2.0
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 * @NModuleScope SameAccount
 */
define(["N/log"], function (log) {
    function beforeLoad(context) {
        log.debug({
            "title": "Before Load",
            "details": "type=" + context.type
        });
    }
}

return {
```

```

        "beforeLoad": beforeLoad
    };
});

```

## Modifying the UI form

```

// 1.0
// Revealing Module pattern, structures 1.0 similar to 2.0
var myNamespace = myNamespace || {};
myNamespace.example = (function () {

    /** @appliedtorecord employee */
    var exports = {};

    function beforeLoad(type, form, request) {
        showBonusEligibility(form);
    }

    function showBonusEligibility(form) {
        var field = form.addField("custpage_is_bonus_eligible",
            "checkbox", "Eligible for Bonus?");
        field.setDefaultValue(isEligibleForBonus(nlapiGetNewRecord()) ? "T" : "F");
    }

    function isEligibleForBonus(rec) {
        // Implement actual business rules for bonus eligibility here
        return true;
    }

    exports.beforeLoad = beforeLoad;
    return exports;
})();

// 2.0
/**
 * @appliedtorecord employee
 * @NScriptType UserEventScript
 * @NApiVersion 2.x
 */
define(["N/log", "N/ui/serverWidget"], function (log, ui) {
    var exports = {};

    function beforeLoad(context) {
        showBonusEligibility(context.form);
    }

    function showBonusEligibility(form) {
        var field = form.addField({
            "id": "custpage_is_bonus_eligible",
            "label": "Eligible for Bonus?",
            "type": ui.FieldType.CHECKBOX
        });
        field.defaultValue = (isEligibleForBonus() ? "T" : "F");
    }

    function isEligibleForBonus(rec) {
        // Implement actual business rules for bonus eligibility here
        return true;
    }
}

```



```

    exports.beforeLoad = beforeLoad;
    return exports;
});

```

## Restrict execution based on the action that triggered the User Event

```

// 1.0
// Utilize the type argument and raw Strings to filter your
// execution by the action
function beforeLoad(type, form, request) {
    // Don't do anything on APPROVE
    // Note that `type` is an Object, so we must use ==, not ===
    if (type == "approve") {
        return;
    }

    // Continue with normal business logic...
}

// 2.0
/**
 * @appliedtorecord employee
 * @NScriptType UserEventScript
 * @NApiVersion 2.x
 */
define([], function () {
    var exports = {};

    // Utilize context.type value and context.UserEventType enumeration
    // to filter your execution by the action
    function beforeLoad(context) {
        // Don't do anything on APPROVE
        if (context.type === context.UserEventType.APPROVE) {
            return;
        }

        // Continue with normal business logic...
    }

    exports.beforeLoad = beforeLoad;
    return exports;
});

```

## Restrict execution based on the context that triggered the User Event

In SuiteScript 1.0, we retrieve the current execution context using

`nlapiGetContext().getExecutionContext()`, then we compare the result to the appropriate raw Strings.

```

// 1.0 in Revealing Module pattern
var myNamespace = myNamespace || {};
myNamespace.example = (function () {
    var exports = {};

    function beforeLoad(type, form, request) {

```

```

        showBonusEligibility(form);
    }

    function showBonusEligibility(form) {
        // Doesn't make sense to modify UI form when the request
        // did not come from the UI
        var currentContext = nlapiGetContext().getExecutionContext();
        if (!wasTriggeredFromUi(currentContext)) {
            return;
        }

        // Continue with form modification...
    }

    function wasTriggeredFromUi(context) {
        // Current context must be compared to raw Strings
        return (context === "userinterface");
    }

    function isEligibleForBonus() {
        return true;
    }

    exports.beforeLoad = beforeLoad;
    return exports;
})();

```

In SuiteScript 2.0, we get the current execution context by importing the `N/runtime` module and inspecting its `executionContext` property. We can then compare its value to the values of the `runtime.ContextType` enumeration rather than raw Strings.

```

// 2.0
/**
 * @NScriptType UserEventScript
 * @NApiVersion 2.x
 */
define(["N/ui/serverWidget", "N/runtime"], function (ui, runtime) {
    var exports = {};

    function beforeLoad(scriptContext) {
        showBonusEligibility(scriptContext.form);
    }

    function showBonusEligibility(form) {
        // Doesn't make sense to modify the form if the
        if (!wasTriggeredFromUi(runtime.executionContext)) {
            return;
        }

        // Continue with form modification...
    }

    function wasTriggeredFromUi(context) {
        // Context can be compared to enumeration from runtime module
        return (context === runtime.ContextType.USER_INTERFACE);
    }

    exports.beforeLoad = beforeLoad;
    return exports;
});

```

```
});
```

Read User Event: Before Load event online: <https://riptutorial.com/netsuite/topic/7119/user-event-before-load-event>

---

# Chapter 24: Using the NetSuite Records Browser

## Examples

### Using the NetSuite Records Browser

The *Records Browser* defines the schema for all scriptable record types; it is an extremely critical reference tool for every SuiteScript developer. When you need to know how to reference a particular field on a specific record type in your script, the *Records Browser* is your guide.

[Direct Link](#)

## Other Schema

You may also notice tabs at the top of the *Records Browser* for *Schema Browser* and *Connect Browser*. These are very similar to the *Records Browser*, but for different NetSuite APIs.

The *Schema Browser* provides the schema for the SOAP-based Web Services API, while the *Connect Browser* provides the schema for the ODBC connector.

### Navigating the Records Browser

You browse the *Records Browser* first by Record Type, i.e. "Sales Order", "Invoice", "Employee". There is no searching capability within the *Records Browser*, so all navigation is done manually. Record Types are organized alphabetically, so you first click on the first letter of the record type you are interested in, then select the Record Type itself at the left.

For example, if you wanted to see the schema for the *Subsidiary* record type, you would first click on *S* at the top, then *Subsidiary* at the left.

### Reading the Schema

Each schema provides you with an overwhelming amount of information about each record type. It is important to know how to break down all of this information.

At the top of the schema is the name of the Record Type followed by the Internal ID of the record type; this internal ID is the programmatic reference for the record type. The schema is then broken up into several sections:

- *Fields*: The *Fields* section lists the details for all of the record's *body* fields. The fields described here can be used when you are working with the record currently in context, or with a direct reference to a record object.
- *Sublists*: The *Sublists* section shows all of the sublists on the record and every scriptable

column within each sublist. The fields in this section again apply when you are working with the record currently in context, or with a direct reference to a record object.

- *Tabs*: The *Tabs* section describes all of the native subtabs on the record type.
- *Search Joins*: The *Search Joins* section describes all of the related records through which you can build joins in your searches of this record type.
- *Search Filters*: The *Search Filters* section describes all of the fields that are available as a search filter for this record type. The internal ID when using a specific field as a search filter *does not always match* its internal ID as a body field.
- *Search Columns*: The *Search Columns* section describes all of the fields that are available as a search column for this record type. The internal ID when using a specific field as a search column *does not always match* its internal ID as a body field.
- *Transform Types*: The *Transform Types* section describes all of the record types that this one can be transformed into using the record transformation API.

## Finding a Field

As stated previously, there is no searching capability built in to the *Records Browser*. Once you've navigated to the appropriate Record Type, if you don't already know a particular field's Internal ID, the easiest way to find it is to use your browser's Find function (usually `CTRL+F`) to locate the field by its name in the UI.

## Required Fields

The *Required* column of the schema indicates whether this field is required to save the record. If this column says `true`, then you will need to provide a value for this field when saving any record of this type.

## nlapiSubmitField and Inline Editing

The `nlapiSubmitField` column is a critical piece to understand. This column indicates whether the field is available for inline editing. If `nlapiSubmitField` is `true`, then the field can be edited inline. This greatly impacts how this field is handled when trying to use the `nlapiSubmitField` or `record.submitFields` functions in your scripts.

When this column is `true`, you can safely use the Submit Fields APIs to update this field inline. When it is `false`, *you can still use these functions to update the field*, but what actually happens behind the scenes changes significantly.

When `nlapiSubmitField` is `false` for a particular field, and you utilize one of the Submit Fields APIs on it, the scripting engine behind the scenes will actually do a full load of the record, update the field, and submit the change back to the database. The end result is the same, but because the entire record is loaded and saved, your script will actually use a lot more governance than you might expect and will take longer to execute.

You can read about this in more detail on the Help page titled "Consequences of Using `nlapiSubmitField` on Non Inline Editable Fields."

Read Using the NetSuite Records Browser online: <https://riptutorial.com/netsuite/topic/7756/using-the-netsuite-records-browser>

---

# Chapter 25: Working with Sublists

## Introduction

NetSuite Records are divided into Body fields and Sublists. There are four types of sublists: Static, Editor, Inline Editor, and List.

We are able to add, insert, edit, and remove line items using Sublist APIs.

For a reference on exactly which sublists support SuiteScript, see the NetSuite Help page titled "Scriptable Sublists".

## Remarks

### Sublist Indices

Each line item in a sublist has an index that we can use to reference it.

In SuiteScript 1.0, these indices are 1-based, so the first line item has index 1, the second has index 2, and so on.

In SuiteScript 2.0, these indices are 0-based, so the first line item has index 0, the second has index 1, and so on. This of course more closely matches Array indexing in most languages, including JavaScript.

### Standard vs Dynamic Mode

The API we use for interacting with a sublist depends on whether we are working with the record in Standard or Dynamic mode.

The Standard-mode APIs simply let us provide the index of the line we want to work with as a parameter to the appropriate function.

The Dynamic-mode APIs follow a pattern:

1. Select the line we want to work with
2. Modify the selected line as desired
3. Commit the changes to the line

In Dynamic Mode, if we do not commit the changes to *each* line we modify, then those changes will not be reflected when the record is saved.

### Limitations

In order to work with sublist data via SuiteScript, we must have a reference in memory to the

record. This means the record either needs to be retrieved from the script context, or we need to load the record from the database.

We *cannot* work with sublists via either [lookup](#) or [submitFields](#) functionality.

*Static* sublists do not support SuiteScript at all.

## References:

- NetSuite Help: "What is a Sublist?"
- NetSuite Help: "Sublist Types"
- NetSuite Help: "Scriptable Sublists"
- NetSuite Help: "Working with Sublist Line Items"
- NetSuite Help: "Sublist APIs"
- NetSuite Help: "Working with Records in Dynamic Mode"

## Examples

### [1.0] How many lines on a sublist?

```
// How many Items does a Sales Order have...

// ... if we're in the context of a Sales Order record
var itemCount = nlapiGetLineItemCount("item");

// ... or if we've loaded the Sales Order
var order = nlapiLoadRecord("salesorder", 123);
var itemCount = order.getLineItemCount("item");
```

### [1.0] Sublists in Standard Mode

```
// Working with Sublists in Standard mode ...

// ... if the record is in context:

// Add item 456 with quantity 10 at the end of the item sublist
var nextIndex = nlapiGetLineItemCount("item") + 1;
nlapiSetLineItemValue("item", "item", nextIndex, 456);
nlapiSetLineItemValue("item", "quantity", nextIndex, 10);

// Inserting item 234 with quantity 3 at the beginning of a sublist
nlapiInsertLineItem("item", 1);
nlapiSetLineItemValue("item", "item", 1, 234);
nlapiSetLineItemValue("item", "quantity", 1, 3);

// Insert item 777 with quantity 2 before the end of the last item
var itemCount = nlapiGetLineItemCount("item");
nlapiInsertLineItem("item", itemCount);
nlapiSetLineItemValue("item", "item", itemCount, 777);
nlapiSetLineItemValue("item", "quantity", itemCount, 2);

// Remove the first line item
```



```

nlapiRemoveLineItem("item", 1);

// Remove the last line item
nlapiRemoveLineItem("item", nlapiGetLineItemCount("item"));

// ... or if we have a reference to the record (rec):

// Add item 456 with quantity 10 at the end of the item sublist
var nextIndex = rec.getLineItemCount("item") + 1;
rec.setLineItemValue("item", "item", nextIndex, 456);
rec.setLineItemValue("item", "quantity", nextIndex, 10);

// Insert item 777 with quantity 3 at the beginning of the sublist
rec.insertLineItem("item", 1);
rec.setLineItemValue("item", "item", 1, 777);
rec.setLineItemValue("item", "quantity", 1, 3);

// Remove the first line
rec.removeLineItem("item", 1);

// Remove the last line
rec.removeLineItem("item", rec.getLineItemCount("item"));

```

## [1.0] Sublists in Dynamic Mode

```

// Adding a line item to the end of a sublist in Dynamic Mode...

// ... if the record is in context:
nlapiSelectNewLineItem("item");
nlapiSetCurrentLineItemValue("item", "item", 456);
nlapiSetCurrentLineItemValue("item", "quantity", 10);
nlapiCommitLineItem("item");

// ... or if we have a reference to the record (rec):
rec.selectNewLineItem("item");
rec.setCurrentLineItemValue("item", "item", 456);
rec.setCurrentLineItemValue("item", "quantity", 10);
rec.commitLineItem("item");

```

## [1.0] Find a Line Item

```

// Which line has item 456 on it...

// ... if we're in the context of a record
var index = nlapiFindLineItemValue("item", "item", 456);
if (index > -1) {
    // we found it...
} else {
    // item 456 is not in the list
}

// ... or if we have a reference to the record (rec)
var index = rec.findLineItemValue("item", "item", 456);
if (index > -1) {
    // we found it on line "index"...
} else {
    // item 456 is not in the list
}

```

```
}
```

## [2.0] How many lines on a sublist?

```
// How many lines in a sublist in SuiteScript 2.0...

require(["N/record"], function (r) {
    var rec = r.load({
        "type": r.Type.SALES_ORDER,
        "id": 123
    });

    // How many lines are on the Items sublist?
    var itemCount = rec.getLineCount({"sublistId": "item"});
});
```

## [2.0] Sublists in Standard Mode

```
// Working with a sublist in Standard Mode in SuiteScript 2.0...

require(["N/record"], function (r) {
    var rec = r.create({
        "type": r.Type.SALES_ORDER,
        "isDynamic": false
    });

    // Set relevant body fields ...

    // Add line item 456 with quantity 10 at the beginning of the Items sublist
    rec.setSublistValue({"sublistId": "item", "fieldId": "item", "value": 456, "line": 0});
    rec.setSublistValue({"sublistId": "item", "fieldId": "quantity", "value": 10, "line": 0});

    // Insert line item 238 with quantity 5 at the beginning of the Items sublist
    rec.insertLine({"sublistId": "item", "line": 0});
    rec.setSublistValue({"sublistId": "item", "fieldId": "item", "value": 238, "line": 0});
    rec.setSublistValue({"sublistId": "item", "fieldId": "quantity", "value": 5, "line": 0});

    // Insert line item 777 with quantity 3 before the last line of the Items sublist
    var lastIndex = rec.getLineCount({"sublistId": "item"}) - 1; // 2.0 sublists have 0-based
    index
    rec.insertLine({"sublistId": "item", "line": lastIndex}); // The last line will now
    actually be at lastIndex + 1
    rec.setSublistValue({"sublistId": "item", "fieldId": "item", "value": 777, "line":
    lastIndex});
    rec.setSublistValue({"sublistId": "item", "fieldId": "quantity", "value": 3, "line":
    lastIndex});

    // Remove the first line
    rec.removeLine({"sublistId": "item", "line": 0});

    // Remove the last line
    rec.removeLine({"sublistId": "item", "line": rec.getLineCount({"sublistId": "item"}) -
    1});

    rec.save();
});
```

## [2.0] Sublists in Dynamic Mode

```
// Working with Sublists in Dynamic Mode in SuiteScript 2.0...

require(["N/record"], function (r) {
    var rec = r.create({
        "type": r.Type.SALES_ORDER,
        "isDynamic": true
    });

    // Set relevant body fields ...

    // Add line item 456 with quantity 10 at the end of the Items sublist
    var itemCount = rec.selectNewLine({"sublistId": "item"});
    rec.setCurrentSublistValue({"sublistId": "item", "fieldId": "item", "value": 456});
    rec.setCurrentSublistValue({"sublistId": "item", "fieldId": "quantity", "value": 10});
    rec.commitLine({"sublistId": "item"});

    // Insert line item 378 with quantity 2 at the beginning of the Items sublist
    rec.insertLine({"sublistId": "item", "line": 0});
    rec.selectLine({"sublistId": "item", "line": 0});
    rec.setCurrentSublistValue({"sublistId": "item", "fieldId": "item", "value": 378});
    rec.setCurrentSublistValue({"sublistId": "item", "fieldId": "quantity", "value": 2});
    rec.commitLine({"sublistId": "item"});

    // Insert line item 777 with quantity 3 before the last line of the Items sublist
    var lastIndex = rec.getLineCount({"sublistId": "item"}) - 1; // 2.0 sublists have 0-based
    index
    rec.insertLine({"sublistId": "item", "line": lastIndex}); // The last line will now
    actually be at lastIndex + 1
    rec.selectLine({"sublistId": "item", "line": lastIndex});
    rec.setCurrentSublistValue({"sublistId": "item", "fieldId": "item", "value": 777});
    rec.setCurrentSublistValue({"sublistId": "item", "fieldId": "quantity", "value": 3});
    rec.commitLine({"sublistId": "item"});

    // Remove the first line
    rec.removeLine({"sublistId": "item", "line": 0});

    // Remove the last line
    rec.removeLine({"sublistId": "item", "line": rec.getLineCount({"sublistId": "item"}) -
    1});

    rec.save();
});
```

## [2.0] Find a Line Item

```
// Finding a specific line item in SuiteScript 2.0...

require(["N/record"], function (r) {
    var rec = r.load({
        "type": r.Type.SALES_ORDER,
        "id": 123
    });

    // Find the line that contains item 777
    var index = rec.findSublistLineWithValue({"sublistId": "item", "fieldId": "item", "value":
    777});
```

```
// find returns -1 if the item isn't found
if (index > -1) {
    // we found it on line "index"
} else {
    // item 777 is not in the list
}
});
```

Read Working with Sublists online: <https://riptutorial.com/netsuite/topic/9098/working-with-sublists>

# Credits

S. No	Chapters	Contributors
1	Getting started with netsuite	<a href="#">4444</a> , <a href="#">Community</a> , <a href="#">erictgrubaugh</a> , <a href="#">Kirk Lampert</a>
2	Create a record	<a href="#">Deepa N</a> , <a href="#">michoel</a> , <a href="#">YNK</a> , <a href="#">Zain Shaikh</a>
3	Executing a Search	<a href="#">Adolfo Garza</a>
4	Exploiting formula columns in saved searches	<a href="#">MetaEd</a>
5	Governance	<a href="#">erictgrubaugh</a>
6	Inline Editing with SuiteScript	<a href="#">erictgrubaugh</a>
7	Loading a record	<a href="#">Adolfo Garza</a> , <a href="#">Eidolon108</a> , <a href="#">LittleZul</a> , <a href="#">michoel</a> , <a href="#">VicDid</a> , <a href="#">YNK</a>
8	Lookup Data from Related Records	<a href="#">erictgrubaugh</a>
9	Mass Delete	<a href="#">MG2016</a>
10	Requesting customField, customFieldList & customSearchJoin with PHP API Advanced Search	<a href="#">Hayden Thring</a>
11	RESTlet - Process external documents	<a href="#">MG2016</a>
12	RestLet - Retrieve Data (Basic)	<a href="#">MG2016</a>
13	Script and Script Deployment Records	<a href="#">erictgrubaugh</a>
14	Script Type Overview	<a href="#">erictgrubaugh</a>
15	Scripting searches	<a href="#">Slavi Slavov</a>

	with Filter Expressions	
16	Searches with large number of results	<a href="#">Slavi Slavov</a>
17	Sourcing	<a href="#">erictgrubaugh</a>
18	SS2.0 Suitelet Hello World	<a href="#">LittleZul</a>
19	SuiteScript - Process Data from Excel	<a href="#">MG2016</a>
20	Understanding Transaction Searches	<a href="#">erictgrubaugh</a>
21	User Event: Before and After Submit events	<a href="#">erictgrubaugh</a>
22	User Event: Before Load event	<a href="#">erictgrubaugh</a>
23	Using the NetSuite Records Browser	<a href="#">erictgrubaugh</a>
24	Working with Sublists	<a href="#">erictgrubaugh</a>